CURSO DE PROGRAMACIÓN

INSTITUTO TECNOLÓGICO DE CELAYA

DEPARTAMENTO DE INGENIERÍA QUÍMICA

DR. VICENTE RICO RAMÍREZ

vicente@iqcelaya.itc.mx

http://www.iqcelaya.itc.mx/~vicente/

Página de Internet del Curso:

http://www.iqcelaya.itc.mx/~vicente/Programacion/MainProgramacion.html

UNIDAD I

TEMA I

INTRODUCCIÓN A LA COMPUTACIÓN

ARQUITECTURA DE LAS COMPUTADORAS

¿QUE ES UNA COMPUTADORA (UN ORDENADOR)?

Existen numerosas definiciones de una computadora, entre ellas las siguientes:

1) Una computadora es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades hasta miles de millones de veces más rápidas que las alcanzables por los seres humanos.

2) Un ordenador es una máquina capaz de aceptar datos a través de un medio de entrada, procesarlos automáticamente bajo el control de un programa previamente almacenado, y proporcionar la información resultante a través de un medio de salida.

3) Una computadora es cualquier dispositivo en el que la información se representa en forma numérica y que, mediante el recuento, comparación y manipulación de estos números (de acuerdo con un conjunto de instrucciones almacenadas en su memoria), puede realizar una multitud de tareas: Realizar complejos cálculos matemáticos, reproducir una melodía, etc.

4) Una computadora es un dispositivo electrónico capaz de recibir un conjunto de instrucciones y ejecutarlas realizando cálculos sobre los datos numéricos.

COMPONENTES DE UNA COMPUTADORA

Una computadora de cualquier forma que se vea tiene dos tipos de componentes: El Hardware y el Software.

Hardware

Llamamos Hardware a la parte física de la computadora; corresponde a las partes que podamos percibir con el sentido del tacto. En español la traducción más cercana es la de "soporte físico". El hardware que compone a una computadora es muy complejo, pues una pequeña pieza puede contener millones de transistores.

Software

Para que el ordenador trabaje se necesita que le suministren una serie de instrucciones que le indiquen qué es lo que queremos que haga. Estas órdenes se le suministran por medio de programas. El software está compuesto por todos aquellos programas necesarios para que el ordenador funcione apropiadamente. El software dirige de forma adecuada a los elementos físicos o hardware.

LAS PARTES DEL HARDWARE

El Hardware esta compuesto por cinco unidades o secciones básicas: Entrada, Salida, CPU, Memoria y Almacenamiento Secundario. Estas unidades se describen a continuación:

Unidades de Entrada y Salida

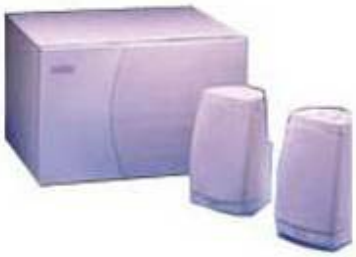
Es la parte del ordenador que le sirve para comunicarse con el exterior; es decir, para recibir y emitir información. A las unidades de entrada y salida se le conoce también como **periféricos**:

El monitor nos muestra la información.

El Teclado y el Mouse sirven para introducir los datos a la computadora.

El lector de CD-ROM sirve para leer la información almacenada en un CD.

Mediante la impresora se obtiene una versión en papel de la información procesada por la computadora.



Las bocinas sirven para escuchar los sonidos que emite la computadora a través de una tarjeta de sonido.

Unidad Central de Procesamiento (CPU)

La unidad central de proceso o CPU es la parte más importante de un ordenador. Esta unidad se encarga de realizar las tareas fundamentales y es por ello el elemento principal de un sistema computarizado. Si hacemos un símil entre un ordenador y el cuerpo humano, la CPU haría el papel del cerebro: atender las solicitudes, mandar y hacer controlar la ejecución.

Un **microprocesador** es un circuito integrado o chip que contiene a la CPU. Su tamaño es algo menor que el de una caja de cerillos.

La unidad central de procesamiento se divide en dos partes: una parte en la en la que se realizan las operaciones aritméticas y lógicas (**unidad aritmético-lógica**) y otra parte que controla todo los proceso de ejecución (**unidad de control**) en la computadora.

La **unidad de control** dirige todas las actividades del ordenador. Actúa como el corazón del sistema, enviando impulsos eléctricos (señales de control) para secuenciar (poner en orden) y sincronizar (establecer tiempos sucesivos de ejecución) el funcionamiento de los componentes restantes.

Unidad de Memoria

La Memoria Principal o Memoria Central es el dispositivo que sirve para almacenar los programas (instrucciones) que se quieran ejecutar (cuando haya que cargar el programa) y para almacenar los datos, los cálculos intermedios y los resultados (cuando el programa ya se esté ejecutando). Sólo los datos almacenados en la memoria son procesables por la CPU. Los datos que estén contenidos en algún dispositivo de almacenamiento externo deben ser

previamente introducidos a la memoria, por medio de una unidad periférica. Dentro de la memoria principal, existen dos divisiones en función de las posibilidades de lectura/escritura o solamente lectura: RAM y ROM.

Memoria RAM (Random Access Memory)

Es la memoria destinada a contener los programas cambiantes del usuario y los datos que se vayan necesitando durante la ejecución de dichos programas. Es la memoria flexible y reutilizable. La memoria RAM se llama también memoria de usuario, por ser la memoria con la que trabaja el sistema para ejecutar los programas. Cuando se hace referencia a la capacidad de memoria de un ordenador se está hablando de la memoria RAM del sistema.

Memoria ROM (Read Only Memory)

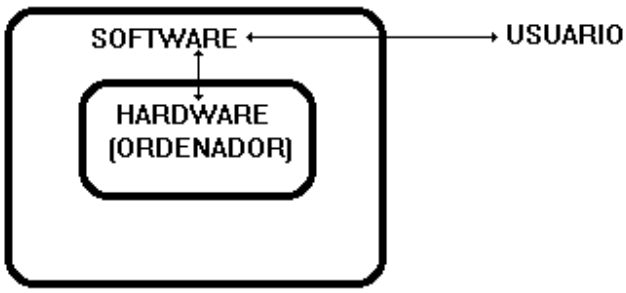
Memoria de solo lectura, llamada también memoria residente o permanente. Son memorias que sólo permiten la lectura y no pueden ser re-escritas. Su contenido viene grabado por el fabricante de la computadora y no puede ser cambiado. Debido a estas características es que esta memoria se usa para almacenar información vital para el funcionamiento del sistema. La gestión del proceso de arranque, la verificación inicial del sistema, la carga del sistema operativo y diversas rutinas de control de dispositivos de entrada/salida suelen ser las tareas encargadas a los programas grabados en ROM. Los programas que constituyen la información vital de una computadora forman la llamada BIOS (Basic Input Output System).

Unidad de Almacenamiento Secundario

Esta es el almacén de largo plazo y de alta capacidad de la computadora. Los programas y datos que no están siendo utilizados por las otras unidades normalmente se colocan en dispositivos de almacenamiento secundario hasta que necesiten, posiblemente horas, días, meses o incluso años después. Ejemplo: Disco duro.

EL SOFTWARE

El ordenador, por sí mismo, no puede realizar ninguna función; es necesario que algo le dirija y organice. Este "algo" son las instrucciones que el programador escribe. Estas instrucciones, agrupadas en forma de programas que son depositados en la memoria del ordenador, forman lo que se denomina "**software**". El software es el nexo de unión entre el hardware y el hombre.



Tal y como hemos definido el software, éste es un conjunto de programas. La pregunta ahora es: ¿Qué es un programa? Un **programa** es una secuencia de instrucciones que pueden ser interpretadas por un ordenador, obteniendo como fruto de esa interpretación un determinado resultado.

Podemos clasificar en software en dos grandes grupos: software de sistema (Sistema Operativo) y software de aplicación.

Software del Sistema o Sistema Operativo

El sistema operativo es aquel conjunto de programas cuyo objeto es facilitar el uso eficiente de la computadora. Este conjunto de programas administra los recursos del sistema (hardware).

El sistema operativo se puede dividir en ***programas de control*** y **programas de servicio**. Los programas de control son los que van orientados a facilitar, automatizar y mejorar el rendimiento de los procesos en el ordenador (simultaneidad de operación de periféricos, tratamiento de errores, etc.); como ejemplo se tiene al administrador de tareas de windows. Los programas de servicio o de proceso son los que van orientados a proporcionar facilidades de comunicación con el usuario (Ejemplo: aplicaciones como el explorador de windows)

Software de Aplicación

El software de aplicación está constituido por aquello programas que hacen que el ordenador coopere con el usuario en la realización de tareas típicamente humanas, tales como gestionar una contabilidad, escribir un texto, hacer gráficos y diagramas, realizar cálculos repetitivos, etc. Algunos ejemplos de software de aplicación son: procesadores de texto (Word), hojas de cálculo (Excel), sistemas de bases de datos (Access), etc.

La diferencia principal entre los programas de aplicación y el sistema operativo estriba en que los del sistema operativo suponen una ayuda al usuario para relacionarse con el ordenador y hacer un uso más cómodo del mismo, mientras que los de aplicación son programas que cooperan con el usuario para la realización de tareas que anteriormente habían de ser llevadas a cabo únicamente por el hombre (sin ayuda de ordenador). Es en estos programas de aplicación donde se aprecia de forma más clara la ayuda que puede suponer un ordenador en las actividades humanas, ya que la máquina se convierte en un auxiliar del hombre, liberándole de las tareas repetitivas.

LOS SISTEMAS DE REPRESENTACIÓN NUMÉRICA

Es común escuchar que las computadoras utilizan el sistema binario para representar cantidades e instrucciones. En esta sección se describen las ideas principales en este sentido. Los sistemas de representación numérica más usados son el sistema binario, el sistema octal, el sistema hexadecimal y, por supuesto, el sistema decimal.

Sistema decimal

En cuanto al problema de representar cantidades numéricas, el sistema decimal es el sistema tradicional. Recordemos que en este sistema, una cantidad cualquiera, por ejemplo 1798, en realidad se lee como:

$1 \times 10^3 + 7 \times 10^2 + 9 \times 10^1 + 8 \times 10^0 \quad == \quad 1 \times 1000 + 7 \times 100 + 9 \times 10 + 8 \times 1$

Observe la diferencia entre los diversos conceptos involucrados: La representación de la cantidad en el sistema decimal (1798) y las "cifras" que componen su representación en este "sistema de numeración" (en este caso cuatro cifras: 1, 7, 9 y 8 colocadas en un cierto orden). En este sistema, el valor de las cifras viene complementado por su posición en el conjunto (se dice que el sistema es posicional), de forma que el valor total de una expresión viene representado por el producto de su valor-base (0 a 9) multiplicado por la potencia de 10 que corresponda según su posición. Al final se suman los resultados parciales. Resulta así que, en el sistema decimal, la cantidad más alta que se puede representar mediante una cantidad de cuatro cifras (como ejemplo), nnnn, es como máximo:

$9 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 9 \times 10^0 \quad == \quad 9999$

Es fácil verificar que un número decimal de n dígitos puede representar como máximo a 10^n números (en el caso del ejemplo, para 4 dígitos, $10^4 = 10000$).

Ejemplo:1534

Valor posicional	**10^3**	**10^2**	**10^1**	**10^0**		
4 en 10^0				**4**	4 x 10^0	4
3 en 10^1			**3**		3 x 10^1	30
5 en 10^2		**5**			5 x 10^2	500
1 en 10^3	**1**				1 x 10^3	1000

Sistema binario

El sistema binario puede representar también cualquier cantidad basándose en cifras que solo pueden tener dos valores, 0 y 1. Es importante resaltar que las características físicas de los dispositivos electrónicos (como las computadoras) hacen que sea fácil representar con ellos a cantidades en el sistema binario. Para lograr esto se hace corresponder los dos posibles valores de la variable binaria con los dos estados físicos de un circuito o dispositivo. Por ejemplo, con los estados de circuito abierto (1) o cerrado (0); magnetizado (1) no magnetizado (0); con luz (1), sin luz (0); etc. Esta es la razón por la cual las computadoras utilizan el sistema binario. En cambio, el sistema decimal no es muy adecuado para los dispositivos electrónicos, puesto que aquí, al ser un sistema de base 10, cada cifra pueden tener diez valores distintos (0 al 9) y sería complicada asignar cada valor a un estado físico de algún dispositivo electrónico.

El sistema binario es exactamente análogo al decimal, con la diferencia de la base de las potencias y de los posibles valores para cada cifra (0 ó 1). En el sistema binario la nueva base es 2 (en vez de 10 como en aquel caso). Por ejemplo, la cantidad binaria 11100000110 se lee:

$$1x2^{10} + 1x2^{9} + 1x2^{8} + 0x2^{7} + 0x2^{6} + 0x2^{5} + 0x2^{4} + 0x2^{3} + 1x2^{2} + 1x2^{1} + 0x2^{0}$$

Se puede probar que el número anterior resulta igualmente en el número "mil setecientos noventa y ocho". La capacidad de representación en el sistema

binario es, sin embargo, menor que en el decimal. Así, una cantidad binaria de 4 dígitos puede representar como máximo al número:

$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 == 15$

En el caso de la numeración binaria, también es fácil verificar que un número de n dígitos puede representar como máximo a 2^n números (en el caso del ejemplo, para 4 dígitos, $2^4 = 16$).

Ejemplo: el número binario 1011 es igual al número decimal 11

Valor posicional	**2^3**	**2^2**	**2^1**	**2^0**		**Valor decimal**
1 en 2^0				**1**	1×2^0	1
1 en 2^1			**1**		1×2^1	2
0 en 2^2		**0**			0×2^2	0
1 en 2^3	**1**				1×2^3	8

Otros sistemas de representación numérica

Además del binario (sistema "natural" del ordenador), en la literatura informática y en los programas se utilizan otras formas de numeración (sobre todo para representar valores constantes). La más común de ellas es el sistema **hexadecimal**.

El sistema hexadecimal, como los anteriores, también es posicional. En este caso el valor de la posición viene dado por potencias de 16 (16^0, 16^1, 16^2,…).

Como sólo poseemos 10 caracteres para representar los posibles dígitos, se añaden las letras A, B, C, D, E y F.

Por tanto en base 16 disponemos de los siguientes caracteres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, **A** = 10, **B** = 11, **C** = 12, **D** = 13, **E** = 14, y **F** = 15.

Como ejemplo, el número A52F corresponde al número 42287 en base decimal.

Valor posicional	**16^3**	**16^2**	**16^1**	**16^0**		**Valor decimal**
F en 16^0				**F**	F x 16^0	15
2 en 16^1			**2**		2 x 16^1	32
5 en 16^2		**5**			5 x 16^2	1280
A en 16^3	**A**				A x 16^3	40960

Conversión de Números en el Sistema Decimal a los Sistemas Binario y Hexadecimal

Al Sistema Binario

Si lo que queremos es convertir un número decimal a binario, dividiremos sucesivamente el valor decimal por 2 hasta llegar a 1. Los restos de las divisiones nos indicarán el valor binario. El siguiente ejemplo corresponde a la conversión del número 52 en base decimal al sistema binario

División	**Cociente**	**Resto**
52 / 2	26	**0**
26 / 2	13	**0**
13 / 2	6	**1**
6 / 2	3	**0**
3 / 2	1	**1**
1		1

110100

Sistema Hexadecimal

Para realizar la conversión al sistema decimal se sigue un método similar al anterior, sólo que ahora se realiza divisiones entre 16. Véase el ejemplo de la Tabla para el número 332_{10} = 14C $_{16}$

División	**Cociente**	**Resto**
332 / 16	20	12 = **C**
20 / 16	1	**4**
1		**1**

Almacenamiento interno

Una magnitud que solo puede tener dos valores se denomina bit (abreviatura de "Binary digit"). Un bit es la menor cantidad de información que puede concebirse en una computadora, y su abreviatura es b, por ejemplo: 10 Kb son 10000 bits. Resulta así que un interruptor que puede estar encendido o apagado es (puede ser) un almacenamiento de 1 bit de información (basta con hacer corresponder "encendido" con el valor 1 y "apagado" con el valor 0).

Tradicionalmente el almacenamiento interno de los ordenadores se realiza en grupos de 8 (o múltiplos de 8) cifras binarias (bits); estos conjuntos (**octetos**) son la menor cantidad de información que trata el ordenador con entidad propia y reciben el nombre de **bytes** (suele abreviarse con B). Por ejemplo, 16 Kb y 16 KB. Se refieren a 16000 bits y 128000 bits respectivamente.

Podemos usar un símil para entenderlo: Aunque nuestro alfabeto tiene 26 caracteres no se utilizan aislados, para que tengan significado se utilizan en grupos (palabras), los ordenadores utilizan también palabras, solo que estas son siempre de la misma longitud (8, 16, 32 o 64 bits según el modelo de procesador).

Según lo anterior, un octeto (una "palabra" de 8 bits), 1 Byte, puede almacenar un número tan grande como 2^8 = 256, lo que deriva en que si, por ejemplo,

reservamos una palabra de 8 bits para "describir" (contener) una variable, sabemos de antemano que dicha variable no va a poder adoptar nunca mas de 256 estados distintos. Esto unido al hecho de que la forma de representación interna utiliza el sistema binario y los elementos y circuitos físicos son igualmente binarios (pueden adoptar solo dos estados) hacen que las potencias de dos: 8, 16, 32, 64 etc. son "números mágicos" en el mundo de las computadoras.

EJERCICIOS

1 Determine a qué números en base decimal son equivalentes los siguientes números en base binaria, octal y hexadecimal (respectivamente):

$\mathbf{10010101}_2 = 1x2^7 + 0x2^6 + 0x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 149$

$\mathbf{745}_8 = 7x8^2 + 4x8^1 + 5x8^0 = 485$

$\mathbf{A3F2}_{16} = 10x16^3 + 3x16^2 + 15x16^1 + 2x16^0 = 41970$

2 El número **130** en base decimal a que números es equivalente en:

Base binaria ?	130/2	65	0	
	65/2	32	1	
	32/2	16	0	
	16/2	8	0	$\mathbf{10000010}_2$
	8/2	4	0	
	4/2	2	0	
	2/2	1	0	
	½	0	1	
Base Octal ?	130/8	16	2	
	16/8	2	0	$\mathbf{202}_8$
	2/8	0	2	
Base Hexadecimal?				
	130/16	8	2	
	8/16	0	8	$\mathbf{82}_{16}$

LENGUAJES

Lenguaje es el empleo de notaciones, señales y vocales (voz, palabras) para expresar ideas, comunicarse, y establecer relaciones entre los seres humanos. Un lenguaje no sólo consta de "palabras", sino también de su pronunciación y los métodos para combinar las palabras en frases y oraciones; los lenguajes se forman mediante combinaciones de palabras definidas en un diccionario terminológico previamente establecido. Las combinaciones posibles deben respetar un conjunto de reglas sintácticas establecidas, a ello se le conoce con el nombre de ***Sintaxis***. Además, las palabras deben tener determinado sentido, deben ser comprendidas por un grupo humano en un contexto dado, a ello se le denomina ***Semántica***.

TIPOS DE LENGUAJES

Aunque existen muchas clasificaciones, en general se puede distinguir entre dos clases de lenguajes: los **lenguajes naturales** (ingles, alemán, español, etc.) y los **lenguajes artificiales** o formales (matemático, lógico, computacional, etc.). Tanto el lenguaje natural como el lenguaje artificial son humanos. El primero es natural porque se aprende (o adquiere) inconsciente e involuntariamente. Ningún bebé decide aprender o no la lengua que hablan sus padres, y ningún padre sienta a su hijo y le enseña las reglas sintácticas de su lengua. Las personas hablan y se entienden, pero generalmente no se cuestionan las reglas que utilizan al hablar. Por otra parte, los lenguajes artificiales sí se aprenden de manera voluntaria y conscientemente. Un ejemplo de lenguaje artificial son los **lenguajes de programación** utilizados para desarrollar programas informáticos.

LOS LENGUAJES DE PROGRAMACIÓN

Un **Lenguaje de Programación** es un conjunto de reglas, notaciones, símbolos y/o caracteres que permiten a un programador poder expresar el procesamiento de datos y sus estructuras en la computadora. Cada lenguaje posee sus propias sintaxis. También se puede decir que un programa es un conjunto de órdenes o instrucciones que resuelven un problema específico basado en un Lenguaje de Programación.

Existen varias clasificaciones para los lenguajes de programación.

Clasificación de los Lenguajes de Programación

Los programadores escriben instrucciones en diversos lenguajes de programación. La computadora puede entender directamente algunos de ellos, pero otros requieren pasos de traducción intermedios. Hoy día se utilizan cientos de lenguajes de computadora.

Los Lenguajes de Programación pueden clasificarse **de acuerdo con su uso** en:

1. Lenguajes desarrollados para el **cálculo numérico**. Tales como **FORTRAN**, Mathematica y Matlab.
2. Lenguajes para **sistemas**. Como **C**, **C++** y ensamblador.
3. Lenguajes para aplicaciones de **Inteligencia Artificial**. Tales como Prolog, y Lisp.

También se pueden clasificar de acuerdo con el **tipo de instrucciones de que constan**. En esta clasificación se tiene al **lenguaje máquina**, al **lenguaje ensamblador** y al **lenguaje de alto nivel**. Se presenta a continuación una descripción de cada uno de ellos.

Lenguaje máquina (Binario)

Una computadora sólo puede entender el lenguaje máquina. El lenguaje de máquina ordena a la computadora realizar sus operaciones fundamentales una por una. Dicho lenguaje es difícil de usar para lar persona porque trabajar con números no es muy cómodo además de que estos números están en formato binario. ¿Cómo es que se representan las operaciones como números? John Von Neumann desarrolló el modelo que lleva su nombre para esta representación. Ya se estudió que representar números usando el sistema binario no es complicado, pero se tenía luego el problema de representar las acciones (o instrucciones) que iba a realizar la computadora también en el sistema binario; pues la memoria, al estar compuesta por bits, solamente permite almacenar números binarios. La solución que se tomó fue la siguiente: a cada acción que sea capaz de realizar la computadora, se le asocia un número, que corresponde a su código de operación (**opcode**). Por ejemplo, una calculadora programable simple podría asignar los siguientes opcodes :

1 = SUMA, 2 = RESTA, 3 = MULTIPLICA, 4 = DIVIDE

Supóngase entonces que se quiere realizar la operación 5 * 3 + 2, en la calculadora descrita arriba. En la memoria de la calculadora se podría representar el programa de la siguiente forma:

Posición	Opcode	Significado	Comentario
0	5	5	Primer número de la fórmula
1	3	*	3 es el **opcode** que representa la multiplicación.
2	3	3	Segundo número de la fórmula
3	1	+	1 es el **opcode** para la suma.
4	2	2	Último número de la fórmula

y en código binario:

5 3 3 1 2

101 011 011 001 010

Podemos ver que, con esta representación, es simple expresar las operaciones de las que es capaz de realizar el hardware en la memoria. La descripción y uso de los opcodes es lo que se llama lenguaje de máquina. El lenguaje máquina es el lenguaje más primitivo y depende directamente del hardware.

Lenguajes de bajo nivel (ensamblador)

Para facilitar y agilizar su labor a los programadores, se buscaron nuevos lenguajes. Cuando abstraemos los opcodes y los sustituimos por una palabra que sea una clave de su significado, se tiene el concepto de Lenguaje Ensamblador. Así, el lenguaje ensamblador representa las acciones del ordenador mediante pequeñas abreviaturas de palabras en inglés. Podemos entonces definir al Lenguaje Ensamblador de la siguiente forma:

Lenguaje Ensamblador consiste en asociar a los opcodes palabras clave que faciliten su uso por parte del programador

No obstante, el lenguaje ensamblador requiere de muchas instrucciones para realizar simples operaciones.

Lenguajes de alto nivel

Para acelerar aun más el proceso de programación se desarrollaron los lenguajes de alto nivel, en los que se puede escribir un sólo enunciado para realizar tareas sustanciales. Los lenguajes de alto nivel permiten a los programadores escribir instrucciones que asemejan al inglés cotidiano y contiene notaciones matemáticas de uso común. El concepto de lenguaje de alto nivel nació con el lenguaje FORTRAN (FORmula TRANslation) que, como su nombre indica, surgió como un intento de traducir fórmulas matemáticas al lenguaje ensamblador y por consiguiente al lenguaje de máquina. A partir de FORTRAN, se han desarrollado innumerables lenguajes que siguen el mismo concepto: buscar la mayor abstracción posible y facilitar la vida al programador, aumentando la productividad. Entre estos lenguajes de alto nivel se encuentra el ***lenguaje C++*** que servirá de base para el desarrollo del curso.

EJEMPLO DE TIPOS DE LENGUAJES

Lenguaje Máquina

```
100001010101010
100100101010100
100011100101110
```

Lenguaje de Nivel Bajo (Ensamblador)

```
LOAD R1, (B)
LOAD R2, (C)
ADD R1, R2
STORE (A), R1
```

Lenguajes de Alto Nivel

```
A = B + C;
```

HISTORIA DE LOS LENGUAJES DE PROGRAMACIÓN

Se presentan a continuación datos relevantes de algunos de los lenguajes de programación de mayor importancia histórica.

FORTRAN

FORTRAN es el más viejo de los lenguajes de alto nivel. Fue diseñado por IBM en 1950. El idioma se hizo tan popular en los 60´s que otros desarrolladores empezaron a producir sus propias versiones y esto llevó a una gran cantidad de dialectos (en 1963 había 40 compiladores de FORTRAN diferentes). En 1972 se creó *FORTRAN66*, como una forma de estandarizar la estructura del lenguaje. Luego, en 1980, se estableció una norma oficial para el lenguaje avalada por la Organización de Normas Internacionales (ISO). Tal versión es normalmente conocida como *FORTRAN 77* (dado que el proyecto final se completó en 1977). En 1991 surge *FORTRAN90*, un desarrollo mayor del idioma pero que incluye todos los elementos de FORTRAN77 para facilitar la compatibilidad. Finalmente, en 1997, surge *FORTRAN95* o High Performance Fortran (HPF).

BASIC

BASIC es la abreviación de *Beginners All-purpose Symbolic Instruction Code*. Basic fue desarrollado en la Universidad de Dartmouth en 1964 bajo la dirección de J. Kemeny y T. Kurtz. Surgió como un idioma simple de aprender y fácil de traducir. En los 70´s, cuando se creó la computadora personal Altair, Bill Gates y Paul Allen implementaron su propia versión de Basic en dicha computadora. Con ello comenzó el futuro de BASIC y de la PC. En ese tiempo, Gates era estudiante de Harvard y Allen era un empleado de Honeywell. La versión BASIC de Gates ocupaba un total de 4KB de memoria incluyendo el código y los datos que se usaron para el código fuente. Luego Gates implementó BASIC en otras plataformas (Apple, Comodor y Atari) y fue a partir de entonces que la corporación de Microsoft empezó su reinado en el mundo de las PC. Más tarde en los 70's, surgió el sistema operativo MS-DOS de Bill Gates que incluía un intérprete de BASIC. La versión distribuida con MS-DOS era GW-BASIC y podía ser ejecutada en cualquier máquina que pudiera ejecutar DOS.

C

El lenguaje C reúne características de programación tanto de los lenguajes ensambladores como de los lenguajes de alto nivel; este lenguaje posee gran poderío basado en sus operaciones a nivel de bits (propias de ***ensambladores***) y la mayoría de los elementos de la programación estructurada de los lenguajes de ***alto nivel***. Por ello es que C ha sido el lenguaje preferido para el desarrollo de software de sistemas y aplicaciones profesionales de la programación de computadoras.

En 1970 Ken Thompson de los laboratorios Bell creó la primera versión del lenguaje, la cual podía ejecutarse en el sistema operativo UNIX; a este lenguaje se le llamó lenguaje B y tenía la desventaja de ser lento. En 1971 Dennis Ritchie, con base en el lenguaje B, desarrolló NB que luego cambió su nombre por C. Su diseño incluyó una sintaxis simplificada, la aritmética de direcciones de memoria (permite al programador manipular bits, bytes y direcciones de memoria) y el concepto de apuntador. Además, al ser diseñado para mejorar software de sistemas, se buscó que generase códigos eficientes y uno portabilidad total, es decir el que pudiese correr en cualquier máquina. Logrados los objetivos anteriores, C se convirtió en el lenguaje preferido de los programadores profesionales.

C++

En 1980 Bjarne Stroustrup, también de los laboratorios Bell, adicionó al lenguaje C las características de la *programación orientada a objetos* (incluyendo la ventaja de una biblioteca de funciones orientada a objetos) y lo denominó C con clases. Para 1983 dicha denominación cambió a la de C++.

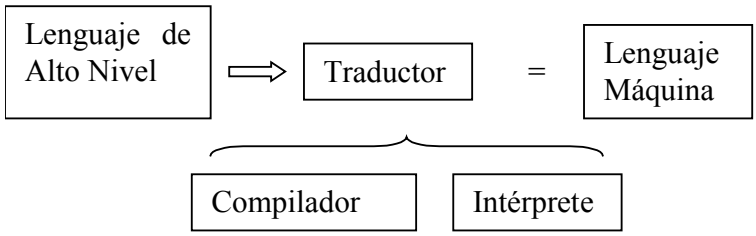
TRADUCTORES Y COMPILADORES

Código Fuente

Se le da el nombre de código fuente a los programas escritos en un determinado lenguaje de programación y que está compuesto por instrucciones escritas por un programador. El código fuente no constituye software propiamente dicho pero es una instancia mediante la cual se logra el software.

Traductores de un Lenguaje de Programación

Los traductores son programas que traducen los programas en código fuente, escritos en lenguajes de alto nivel, a programas escritos en lenguaje máquina. Los traductores pueden ser de dos tipos: compiladores e intérpretes



Compilador

Un compilador es un programa que lee el código escrito en un lenguaje (lenguaje origen), y lo traduce en un programa equivalente escrito en otro lenguaje (lenguaje objetivo). Como una parte fundamental de este proceso de traducción, el compilador le hace notar al usuario la presencia de errores en el código fuente del programa. Vea la siguiente figura.

Los lenguajes C y C++ son lenguajes que utiliza un compilador. El trabajo del compilador y su función es llevar el código fuente escrito en C/C++ a un programa escrito en lenguaje máquina. Entrando en más detalle, un programa en código fuente es compilado obteniendo un archivo parcial (un objeto) que tiene extensión obj. Luego el compilador invoca al "*linker*" que convierte al archivo objeto en un ejecutable con extensión exe; este último archivo es un archivo en formato binario (ceros y unos) y puede funcionar por sí sólo.

Además, el compilador al realizar su tarea realiza también una comprobación de errores en el programa; es decir, revisa que todo esté en orden. Por ejemplo, variables y funciones bien definidas, todo lo referente a cuestiones sintácticas, etc. Está fuera del alcance del compilador que, por ejemplo, el algoritmo utilizado en el problema funcione bien.

La siguiente figura muestra los pasos para tener un programa ejecutable desde el código fuente:

Intérprete

Los intérpretes no producen un lenguaje objetivo como en los compiladores. Un intérprete lee el código como está escrito e inmediatamente lo convierte en acciones; es decir, lo ejecuta en ese instante.

Existen lenguajes que utilizan un intérprete (como por ejemplo JAVA) que traduce en el instante mismo de lectura el código en lenguaje máquina para que pueda ser ejecutado. La siguiente figura muestra el funcionamiento de un intérprete.

Diferencia entre compilador e intérprete

Los compiladores difieren de los intérpretes en varios aspectos: Un programa que ha sido compilado puede correr por sí sólo, pues en el proceso de compilación se lo transformo en otro lenguaje (lenguaje máquina). Un intérprete traduce el programa cuando lo lee, convirtiendo el código del programa directamente en acciones. La ventaja del intérprete es que dado cualquier programa se puede interpretar en cualquier plataforma (sistema operativo). En cambio, el archivo generado por el compilador solo funciona en la plataforma en donde se le ha creado. Sin embargo, hablando de la velocidad de ejecución, un archivo compilado es de 10 a 20 veces más rápido que un archivo interpretado.

CÓDIGO ASCII

Existe una equivalencia en informática entre los números naturales entre 0 y 255 (posibles valores de un byte) y los caracteres, de forma que a cada número le corresponde una letra, símbolo o código. La equivalencia más utilizada es la tabla **ASCII** (American Standard Code for Information Interchange). Cada caracter (por ejemplo la letra A) tiene asignado un número por el ordenador de forma que podemos referenciarlo mediante dicho número. Por ejemplo, la letra A tiene por código de identificación el número 65, la B el 66, el 2 el 50, etc. Se proporciona aquí una copia de esta tabla para su referencia.

Para obtener un carácter a través del teclado, presione la tecla **Alt** y, simultáneamente (sin soltar la tecla), presione el número de código correspondiente. Observe que la primer columna y el primer renglón de la tabla sirven para indicar a qué número en el sistema hexadecimal sería equivalente cada uno de los códigos de los caracteres.

Tabla de códigos ASCII

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32 space	33 !	34 "	35 #	36 $	37 %	38 &	39 '	40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
3	48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
4	64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O

5	80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
6	96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
7	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 blank
8	128 ™	129 ¡	130 ¢	131 £	132 ¤	133 ¥	134 ¦	135 §	136 ¨	137 ©	138 ª	139 «	140 ¬	141	142 ®	143 ¯
9	144 °	145 ±	146 ²	147 ³	148 ´	149 µ	150 ¶	151 ·	152 ¸	153 ¹	154 º	155 »	156 ¼	157 ½	158 ¾	159 ¿
A	160 À	161 Á	162 Â	163 Ã	164 Ä	165 Å	166 Æ	167 Ç	168 È	169 É	170 Ê	171 Ë	172 Ì	173 Í	174 Î	175 Ï
B	176 Ð	177 Ñ	178 Ò	179 Ó	180 Ô	181 Õ	182 Ö	183 ×	184 Ø	185 Ù	186 Ú	187 Û	188 Ü	189 Ý	190 Þ	191 ß
C	192 à	193 á	194 â	195 ã	196 ä	197 å	198 æ	199 ç	200 è	201 é	202 ê	203 ë	204 ì	205 í	206 î	207 ï
D	208 ð	209 ñ	210 ò	211 ó	212 ô	213 õ	214 ö	215 ÷	216 ø	217 ù	218 ú	219 û	220 ü	221 ý	222 þ	223 ÿ
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Note que los códigos entre 0 y 31 son caracteres de control y se suelen llamar "no imprimibles", pues su equivalencia no es un caracter sino una determinada acción. Por ejemplo, el código 13 es equivalente a la pulsación de ENTER. Brevemente, los más interesantes son:

código	equivale	código	equivale
07	beep (pitido del altavoz del PC)	27	ESC (tecla escape)
08	backspace	28	cursor a la derecha
09	Tab (tabulación)	29	cursor a la izquierda
10	line feed (avance de línea)	30	cursor arriba
13	CR (retorno de carro)	31	cursor abajo

NOTA:

Los códigos ASCII a partir del 127 son definibles y dependen de cada máquina. Los representados aquí corresponden con los que son imprimibles desde HTML. En una IBM PC en MS-DOS, por ejemplo, estos códigos pueden ser distintos a los aquí representados.

UNIDAD I

TEMA II

INTRODUCCIÓN A LA PROGRAMACIÓN

RESOLUCIÓN DE PROBLEMAS

Pasos para la solución de problemas

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es esencialmente un proceso creativo, se pueden considerar también como una serie de fases o pasos comunes que generalmente deben seguir todos los programadores.

Las siguientes son las etapas que se deben cumplir para resolver con éxito un problema de programación:

1. Definición del problema
2. Análisis del problema
3. Selección de la mejor alternativa
4. Crear Diagrama de Flujo
5. Codificación
6. Compilación
7. Pruebas
8. Documentación externa

Definición del Problema

Está dada por el enunciado del problema, el cual debe ser claro y completo. Es importante que conozcamos exactamente que se desea de la computadora; mientras que esto no se comprenda no tiene caso pasar a la siguiente etapa.

Análisis del Problema

Entendido el problema (que se desea obtener de la computadora), para resolverlo es preciso analizar:

- Los datos o resultados que se esperan.
- Los datos de entrada que se deben suministrar.

- El proceso al que se requiere someter dichos datos a fin de obtener los resultados esperados.
- Fórmulas, ecuaciones y otros recursos necesarios.

Una recomendación muy práctica es el que nos pongamos en el lugar de la computadora, y analizar que es necesario que me ordenen y en que secuencia para poder producir los resultados esperados.

Selección de la Mejor Alternativa

Analizado el problema posiblemente tengamos varias formas de resolverlo; lo importante es determinar cual es la mejor alternativa. Esto es, la que produce los resultados esperados en el menor tiempo y al menor costo.

Crear Diagrama de Flujo

Una vez que sabemos como resolver el problema, pasamos a dibujar gráficamente la lógica de la alternativa seleccionada. Eso es precisamente un Diagrama de Flujo: la representación gráfica de una secuencia lógica de pasos a cumplir por la computadora para producir un resultado esperado.
La experiencia nos ha demostrado que resulta muy útil trasladar esos pasos lógicos planteados en el diagrama a frases que indiquen lo mismo; es decir, hacer una codificación del programa pero utilizando instrucciones en Español, como si le estuviéramos hablando a la computadora. Esto es lo que se denomina Pseudocódigo. Cuando logremos habilidad para desarrollar programas, es posible que no sea necesario elaborar ni el diagrama de flujo ni el pseudocódigo del programa.

Codificación

Una vez que hayamos elaborado el diagrama, codificamos el programa en el lenguaje de programación seleccionado. Esto es, colocamos cada paso del diagrama en una instrucción o sentencia utilizando un lenguaje que la computadora reconoce. Este programa es el que se conoce como Código Fuente (Source Code).

Todos los lenguajes de programación proveen facilidades para incluir líneas de comentarios en los programas. Estos comentarios aclaran lo que se ordena a la computadora y facilitan la compresión del programa. Puesto que estos comentarios no se toman cuenta como instrucciones y aparecen en los listados del programa, resulta muy conveniente agregar abundantes comentarios a todo programa que codifiquemos. Esto es lo que se denomina **Documentación Interna**.

Compilación

Utilizamos ahora un programa Compilador, el cual analiza todo el programa fuente y detecta errores de sintaxis ocasionados por fallas en la codificación. Las fallas de lógica que pueda tener nuestro programa fuente no son detectadas por el compilador. Cuando no hay errores graves en la compilación, el compilador traduce cada instrucción del código fuente a instrucciones propias de la máquina (Lenguaje de Maquina), creando el Programa Objeto. Cuando hay errores, éstos se deben corregir sobre el mismo programa fuente. El paso de compilación se repite hasta eliminar todos los errores y obtener el programa ejecutable.

Pruebas

Cuando tenemos el programa ejecutable (en lenguaje de maquina) ordenamos al computador que lo ejecute, para lo cual suministramos datos de prueba. Los resultados obtenidos se analizan para identificar cualquiera de las siguientes situaciones:

- La lógica del programa esta bien, pero hay errores sencillos, los cuales se corrigen modificando algunas instrucciones o incluyendo unas nuevas; el proceso debemos repetirlo desde el paso 5.
- Hay errores muy graves ocasionados por fallas en la lógica, y lo más aconsejable es que regresemos al paso 2 para analizar nuevamente el problema y repetir todo el proceso.

- No hay errores y los resultados son los esperados. En este caso, el programa lo podemos guardar permanentemente para usarlo cuando necesitemos ejecutarlo nuevamente.

Documentación Externa

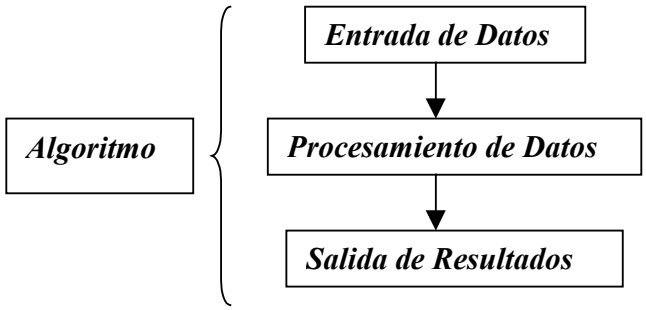
Cuando el programa ya se tiene listo para ejecutar, es conveniente que hagamos su documentación externa. Una buena documentación externa incluiría, por ejemplo:

- Enunciado del problema
- Narrativo con la descripción de la solución
- Descripción de las variables utilizadas en el programa, cada una con su respectiva función
- Resultados de la ejecución del programa

ALGORITMOS Y DIAGRAMAS DE FLUJO

Algoritmo

Un algoritmo es un conjunto de acciones que determinan la secuencia de los pasos a seguir para resolver un problema específico. Sus pasos deben estar definidos con precisión de forma que no existan ambigüedades que den origen a elegir una opción equivocada. Los algoritmos son finitos; es decir, su ejecución termina en un número determinado de pasos. La mayoría de los algoritmos de utilidad al programador poseen 3 partes principales:



Los algoritmos pueden representarse a través de un conjunto de palabras por medio de las cuales se puede representar la lógica de un programa. Este conjunto de palabras constituyen lo que se conoce como **pseudocódigo**. Además, los algoritmos se pueden representar gráficamente a través de un **diagrama de flujo**. Ambas herramientas se describen a continuación.

Diagramas de flujo

Un diagrama de flujo es una representación gráfica de un algoritmo o de una parte del mismo. La ventaja de utilizar un diagrama de flujo es que se le puede construir independientemente del lenguaje de programación, pues al momento de llevarlo a código se puede hacer en cualquier lenguaje. Dichos diagramas se construyen **utilizando ciertos símbolos** de uso especial como son rectángulos, óvalos, pequeños círculos, etc.; estos símbolos están conectados entre sí por flechas conocidas como **líneas de flujo**. A continuación se presentan estos símbolos y su significado.

Símbolos y su Significado

Terminal. Representa el inicio y fin de un programa.

Proceso. Son acciones que el programa tiene que realizar

Decisión. Indica operaciones lógicas o de comparación.

Entrada. Nos permite ingresar datos.

Salida. Es usado para indicar salida de resultados

Selector múltiple. Representa una decisión con múltiples alternativas.

Conector. Enlaza dos partes cualesquiera de un programa

Línea de flujo. Indica dirección de flujo del diagrama. Las flechas de flujo no deben cruzarse. Los diagramas se leen de arriba hacia abajo y de izquierda a derecha.

Conector fuera de página. Representa conexión entre partes del algoritmo representadas en páginas diferentes.

Pseudocódigo

A continuación se muestran algunos ejemplos de palabras utilizadas para construir algoritmos en pseudocódigo.

PALABRA	UTILIZACIÓN
ABRE	Abre un archivo
CASO	Selección entre múltiples alternativas
CIERRA	Cierra un archivo
ENTONCES	Complemento de la selección SI - ENTONCES
ESCRIBE	Visualiza un dato en pantalla
FIN	Finaliza un bloque de instrucciones
HASTA	Cierra la iteración HAZ - HASTA
HAZ	Inicia la iteración HAZ - HASTA
INICIO	Inicia un bloque de instrucciones
LEER	Leer un dato del teclado
MIENTRAS	Inicia la iteración mientras
NO	Niega la condición que le sigue
O	Disyunción lógica
O - BIEN	Complemento opcional de la selección SI - ENTONCES
PARA	Inicia un número fijo de iteraciones
SI	Inicia la selección SI-ENTONCES
USUAL	Opcional en la instrucción CASO
Y	Conjunción lógica
{	Inicio de comentario
}	Fin de comentario
<=	Asignación

PROGRAMACIÓN ESTRUCTURADA

La programación estructurada es un estilo con el cual el se busca que el programador elabore programas sencillos y fáciles de entender. Para ello, la programación estructurada hace uso de tres estructuras básicas de control. Éstas son:

Estructura Secuencial

Estructura Selectiva

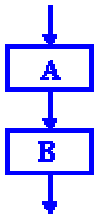
Estructura Repetitiva (ó Iterativa)

La programación estructurada se basa un teorema fundamental, el cual afirma que cualquier programa, no importa el tipo de trabajo que ejecute, puede ser elaborado utilizando únicamente las tres estructuras básicas (secuencia, selección, iteración).

DEFINICIÓN DE LAS TRES ESTRUCTURAS BÁSICAS

Estructura Secuencial

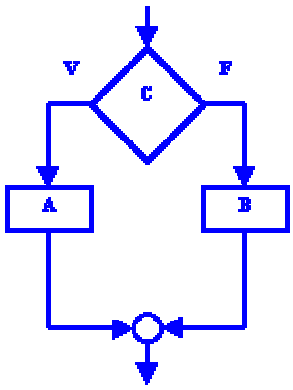
Indica que las instrucciones de un programa se ejecutan una después de la otra, en el mismo orden en el cual aparecen en el programa. Se representa gráficamente como una caja después de otra, ambas con una sola entrada y una única salida.



Las cajas A y B pueden ser definidas para ejecutar desde una simple instrucción hasta un módulo o programa completo, siempre y cuando éstos también sean programas apropiados.

Estructura Selectiva

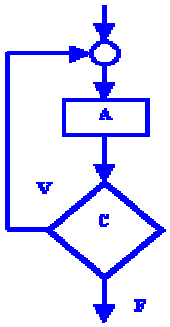
También conocida como la estructura SI-VERDADERO-FALSO, plantea la selección entre dos alternativas con base en el resultado de la evaluación de una condición; equivale a la instrucción IF de todos los lenguajes de programación y se representa gráficamente de la siguiente manera:



En el diagrama de flujo anterior, C es una condición que se evalúa; A es la acción que se ejecuta cuando la evaluación de esta condición resulta verdadera y B es la acción ejecutada cuando el resultado de la evaluación indica falso. La estructura también tiene una sola entrada y una sola salida; y las funciones A y B también pueden ser cualquier estructura básica o conjunto de estructuras.

Estructura Repetitiva (Iterativa)

También llamada la estructura HACER-MIENTRAS-QUE, corresponde a la ejecución repetida de una instrucción mientras que se cumple una determinada condición. El diagrama de flujo para esta estructura es el siguiente:



Aquí el bloque A se ejecuta repetidamente mientras que la condición C se cumpla o sea cierta. También tiene una sola entrada y una sola salida; igualmente A puede ser cualquier estructura básica o conjunto de estructuras.

Ventajas de la Programación Estructurada

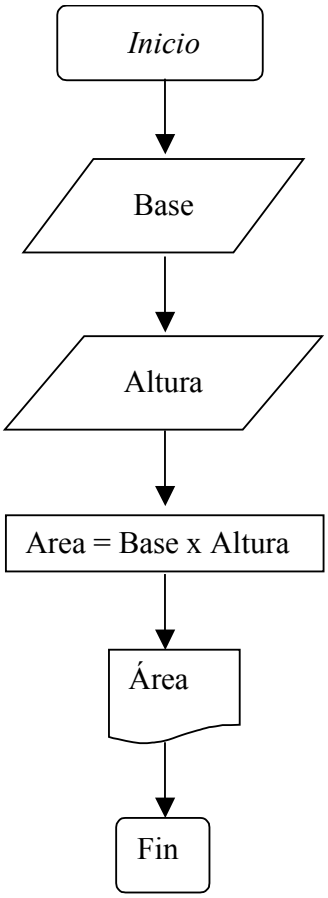
Con la programación estructurada, elaborar programas de computadora sigue siendo una labor que demanda esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este nuevo estilo podemos obtener las siguientes ventajas:

1. Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación.
2. Se logra una reducción del esfuerzo en las pruebas. El seguimiento de las fallas o depuración (debugging) se facilita debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente.
3. Se crean programas más sencillos y más rápidos.

EJERCICIOS SOBRE DIAGRAMAS DE FLUJO

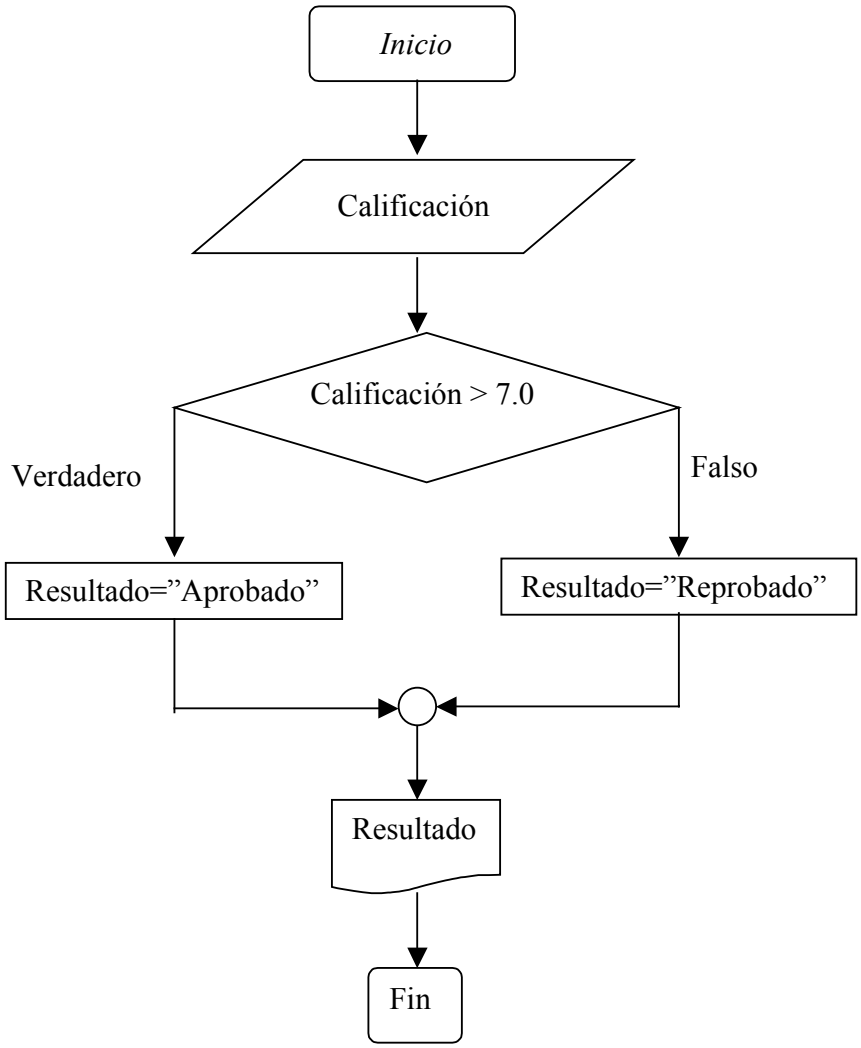
Estructura Secuencial

Calcular el área de un rectángulo a partir de su altura y su base



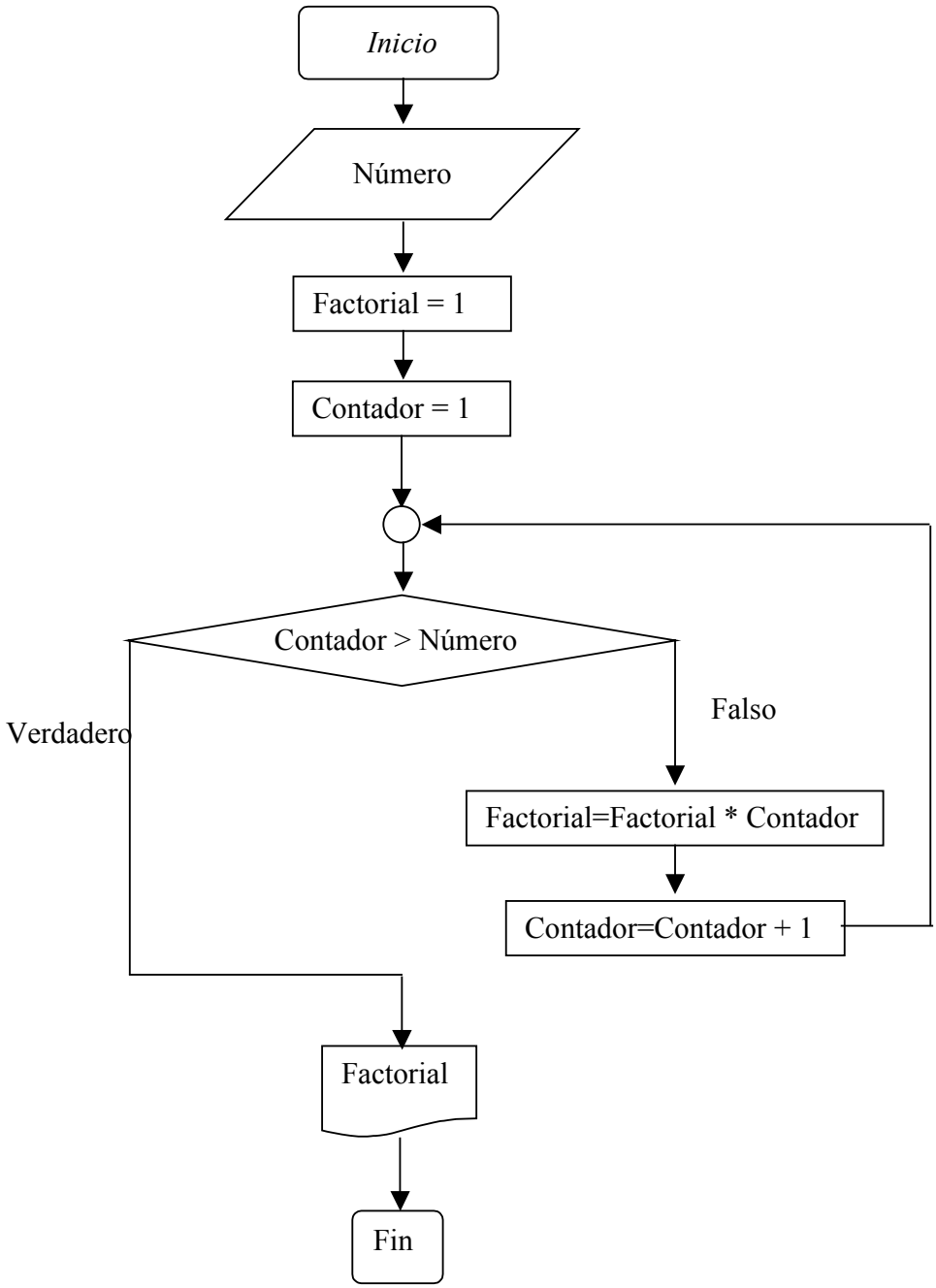
Estructura Selectiva

Convertir calificaciones numéricas (0 al 10) a calificaciones de "Aprobado" ó "Reprobado", siendo 7.0 la calificación mínima aprobatoria.



Estructura Repetitiva (Iterativa)

Calcular el factorial de un número entero positivo



Ejercicios

Estructura Secuencial

Convertir una cantidad dada en metros a pies y pulgadas.

Estructura Selectiva

Dados dos números, ordenarlos ascendentemente

Estructura Repetitiva

Multiplicar entre sí todos los números enteros entre n (el menor) y m (incluyéndolos) si tales números están dados.

PARADIGMAS DE PROGRAMACIÓN

Existe una infinidad de definiciones de lo que es un paradigma. Un paradigma es un determinado marco desde el cual miramos el mundo, lo comprendemos, lo interpretamos e intervenimos sobre él. Abarca desde el conjunto de conocimientos científicos que imperan en una época determinada hasta las formas de pensar y de sentir de la gente en un determinado lugar y momento histórico.

Adam Smith define paradigma, en su libro "Los poderes de la mente", como "*un conjunto compartido de suposiciones. Es la manera como percibimos el mundo: agua para el pez. El paradigma nos explica el mundo y nos ayuda a predecir su comportamiento*".

En nuestro contexto, el paradigma debe ser concebido como una forma aceptada de resolver un problema en la ciencia, que más tarde es utilizada como modelo para la investigación y la formación de una teoría. También, el paradigma debe ser concebido como un conjunto de métodos, reglas y generalizaciones utilizadas conjuntamente por aquellos entrenados para realizar el trabajo científico de investigación.

En nuestro contexto, ***los paradigmas de programación nos indican las diversas formas que, a lo largo de la evolución de los lenguajes, han sido aceptadas como estilos para programar y para resolver los problemas por medio de una computadora.***

Se muestran a continuación un resumen de los paradigmas de uso más extendido en programación.

PROGRAMACIÓN POR PROCEDIMIENTOS

Es el paradigma original de programación y quizá todavía el de uso más común. En él, el programador se concentra en el procesamiento, en el algoritmo requerido para llevar a cabo el cómputo deseado.

Los lenguajes apoyan este paradigma proporcionando recursos para pasar argumentos a las funciones y devolviendo valores de las funciones. FORTRAN es

el lenguaje de procedimientos original, Pascal y C son inventos posteriores que siguen la misma idea. La ***programación estructurada*** se considera como el componente principal de la ***programación por procedimientos***.

PROGRAMACIÓN MODULAR

Con los años, en el diseño de programas se dio mayor énfasis al diseño de procedimientos que a la organización de la información. Entre otras cosas esto refleja un aumento en el tamaño de los programas. La programación modular surge como un remedio a esta situación. A menudo se aplica el término módulo a un conjunto de procedimientos afines junto con los datos que manipulan. Así, el paradigma de la programación modular consiste en:

a) Establecer los módulos que se requieren para la resolución de un problema.

b) Dividir el programa de modo que los procedimientos y los datos queden ocultos en módulos.

Este paradigma también se conoce como principio de ocultación de procedimientos y datos. Aunque C++ no se diseño específicamente para desarrollar la programación modular, su concepto de clase proporciona apoyo para el concepto de módulo.

ABSTRACCIÓN DE DATOS

Los lenguajes como ADA y C++ permiten que un usuario defina tipos que se comporten casi de la misma manera que los tipos definidos por el lenguaje. Tales tipos de datos reciben a menudo el nombre de tipos abstractos o tipos definidos por el usuario. El paradigma de programación sobre este tipo de datos consiste en:

a) Establecer las características de los tipos de datos abstractos se desean definir.

b) Proporcionar un conjunto completo de operaciones válidas y útiles para cada tipo de dato.

Cuando no hay necesidad de más de un objeto de un tipo dado, no es necesario este estilo y basta con el estilo de programación de ocultamiento de datos por medio de módulos.

PROGRAMACIÓN ORIENTADA A OBJETOS (OOP)

El problema con la abstracción de datos es que no hay ninguna distinción entre las propiedades generales y las particulares de un conjunto de objetos. Expresar esta distinción y aprovecharla es lo que define a la OOP a través del concepto de herencia. El paradigma de la programación orientada a objetos es, entonces,

a) Definir que clases se desean
b) Proporcionar un conjunto completo de operaciones para cada clase
c) Indicar explícitamente lo que los objetos de la clase tienen en común empleando el concepto de herencia

En algunas áreas las posibilidades de la OOP son enormes. Sin embargo, en otras aplicaciones, como las que usan los tipos aritméticos básicos y los cálculos basados en ellos, se requiere únicamente la abstracción de datos y/o programación por procedimientos, por lo que los recursos necesarios para apoyar la OOP podrían salir sobrando.

LENGUAJE C++: PRIMER PROGRAMA

Veamos nuestro primer programa en C++. Esto nos ayudará a sentar una base útil para el desarrollo de los siguientes ejemplos que irán apareciendo.

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

int main()

{
    //Esto es un comentario y no tiene efecto alguno en el programa.
    //El siguiente tambien es un comentario.
    /* Observe que hay dos formas de escribir un comentario. Una es
       utilizando dos diagonales y la otra utilizando asterisco y una diagonal.
       Cuando se usan dos diagonales es necesario escribirlas al
       principio de cada linea, pero solo es necesario al principio de la linea.
       Cuando se usan un asterisco y una diagonal, no se requiere que
       se escriban al principio de cada linea, pero se requiere que
       que se escriban al principio y al final del comentario. Observe
       que el orden del asterisco y la diagonal cambia segun se abra
       o se cierre el comentario */

    cout<<"Este es el programa mas simple que puede haber.\n"
    <<"Posee unicamente comentarios y mensajes de salida a pantalla.\n"
    <<"Haz la prueba escribiendo mensajes como este.\n";

    /* Observa que \n es equivalente a teclear enter. También observa que
       un sólo cout puede servir para enviar varias líneas de mensajes.
       Sin embargo, también puedes usar un cout por cada línea. Nota que
       para cada cout debes de usar un semicolon (;), como en el siguiente
       ejemplo */

    cout<<"\n";
    cout<<"Como se vera durante el curso, los mensajes a pantalla y \n";
    cout<<"los comentarios son muy utiles para que el programa sea\n";
    cout<<"claro y para facilitar la comunicacion con el usuario.\n";
    cout<<"\n";

    system("PAUSE");
    return 0;
}
```

La función main

Iremos repasando muy someramente el programa. Observe la estructura que ha sido escrita en negritas:

```
int main()
{
return 0;
}
```

Se trata de instrucciones muy especiales que se van a encontrar en TODOS los programas de C y C++.

En principio, se puede pensar en dichas instrucciones como la definición de una función. Todas las funciones C++ toman unos valores de entrada, llamados parámetros o argumentos, y devuelven un valor de retorno. La primera palabra, "int", nos dice el tipo del valor de retorno de la función, en este caso un número entero. La función "main" siempre devuelve un entero. "main" es el nombre de la función; en general será el nombre que usaremos cuando queramos usar o llamar a la función. Sin embargo, en este caso "main" es una función especial, ya que nosotros no la usaremos nunca explícitamente; es decir, nunca encontrarás en ningún programa una línea que invoque a la función "main". Esta función será la que tome el control automáticamente cuando se ejecute nuestro programa. Observe también que las llaves **{ }** encierran el cuerpo o definición de la función. Más adelante veremos que también tiene otros usos. Posteriormente en el curso se analizarán todos los conceptos relacionados con funciones.

Comentarios

Un comentario no es propiamente un tipo de sentencia dado que no es ejecutado por el programa. En C++ pueden introducirse comentarios en cualquier parte del programa. Estos comentarios ayudarán a seguir el funcionamiento del programa durante la depuración o en la actualización del programa, además de documentarlo. Los comentarios en C++ se delimitan

entre /* y */, cualquier cosa que escribamos en su interior será ignorada por el compilador. En C++ se ha incluido otro tipo de comentarios, que empiezan con //. Estos comentarios no tienen marca de final, sino que terminan cuando termina la línea. Por ejemplo:

```
// Esto es un comentario
```

cout

Sin entrar en detalle el comando **cout** es un elemento que permitirá que nuestros programas se comuniquen con nosotros. Sirve para indicar salida estándar. Este elemento nos permite enviar a la pantalla cualquier variable o constante, incluidos literales. Lo veremos más detalladamente en una sesión en conjunto con la declaración de variables, de momento sólo nos interesa cómo usarlo para mostrar cadenas de caracteres (mensajes a pantalla).

Nota: en realidad **cout** es un objeto de C++ pero los conceptos de clase y objeto no se revisarán en este curso.

El uso es muy simple:

```
#include <iostream.h>
cout << texto o nombre de variable <<_texto o nombre de variable …    ;
```

Librerías externas

Aunque es este momento no entraremos en detalle, la línea

```
#include <iostream.h>
```

es necesaria porque las funciones que permiten el acceso a **cout** están definidas en una librería externa. Con todos los elementos incluidos aquí ya podemos escribir algunos ejemplos en C++.

INSTRUCCIONES BÁSICAS PARA LA ESCRITURA Y COMPILACIÓN DE UN PROGRAMA CON DEV-C++

En este tutorial el símbolo → será utilizado para indicar la selección de alguna opción de un menú.

1. Vaya a ***Inicio*** → ***Programas*** → ***BloodShed Dev-C++*** → ***Dev-C++***

Dev-C++.lnk

2. Una vez que aparezca al ventana principal del programa, vaya al menú ***File*** → ***New*** → ***Source File***

3. Aparecerá un archivo con el nombre de ***Untitled1***. Lo primero que se sugiere escribir en dicho archivo es la estructura básica del programa (todo programa tiene un inicio y un fin; en otras palabras, todo programa debe contener dicha estructura básica):

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

int main()
{

    system("PAUSE");
    return 0;
}
```

4. Si el estudiante lo desea puede llevar consigo al centro de cómputo un disco de 3 ½ " para guardar en él sus programas, aunque es posible también guardarlos en el disco duro de la computadora (que se almacena solo de forma temporal y puede ser borrado por el personal del centro).
   Si selecciona la opción del disco, insértelo en el drive apropiado de la computadora.
   Vaya al menú ***File*** → ***Save As*** . Aparecerá la ventana ***Save File***. En dicha ventana, en la línea correspondiente a ***Guardar en***, utilice el cursor para

seleccionar el disco de 3 ½ (drive A) o la carpeta apropiada en el disco duro de la computadora. En la línea correspondiente a ***Nombre***, escriba el nombre con el cual quiere guardar su programa. Presione el botón ***Guardar***.

5. Modifique el archivo añadiéndole las instrucciones apropiadas que debería tener el programa para resolver su problema. Es decir, teclee las instrucciones de su diagrama de flujo utilizando la sintaxis del lenguaje C++.

6. Terminado su programa, seleccione el menú ***File*** → ***Save***

7. El programa escrito en lenguaje de alto nivel (en este caso C++) debe ser traducido a lenguaje máquina para que el procesador de la computadora pueda ejecutarlo. El traductor en este caso es un compilador.

   Vaya al menú ***Execute*** → ***Compile***.

8. Aparece una ventana (***Compile Progress***) que indicará si su programa tiene errores de sintaxis.
   a) Si el programa tiene errores, la lista de errores se muestra en la parte inferior de la ventana del programa. Corrija los errores y compile las veces que sea necesario hasta que el programa ya no tenga errores de sintaxis.
   b) Si el programa no tiene errores, el archivo ejecutable correspondiente a su programa será guardado en su disco o en el disco duro, según su selección. Usando las ventanas principales de Dev-C++ puede ejecutar el programa. Vaya al menú ***Execute*** → ***Run***.

   Los pasos de compilación y ejecución pueden también realizarse a través de ir al menú ***Execute*** → ***Compile and Run***. En este caso el programa sólo será ejecutado si no hay errores de compilación.

9. Vea los resultados obtenidos con el programa. Si no es lo esperado, corrija el programa hasta obtener los resultados esperados. Haga también modificaciones en el código fuente para que observe como se modifica la ejecución del programa.

TIPOS DE DATOS: CLASIFICACIÓN DE VARIABLES

Durante la creación de los diagramas de flujo en los ejercicios que se han realizado, han aparecido nombres de **variables** que nos han servido para representar cantidades que se van a calcular o que van a ir cambiando durante la ejecución del algoritmo (Recordar, por ejemplo, base, altura, contador, producto, factorial, etc). Como se verá en su momento, los lenguajes de programación requieren que el programador defina (o **declare**) todos las datos o variables del programa antes de su utilización.
Básicamente, declarar una variable servirá para indicar a la computadora a que **tipo** de datos corresponden cada una de las variables del problema.

Los tipos de datos determinan cómo se manipulará la información contenida en las variables que se definan. No olvidar que la información en el interior de la memoria del ordenador es siempre binaria. El mismo valor puede usarse para codificar una letra, un número, una instrucción de programa, etc. Saber que tipo de información se tiene almacenada permite a la computadora interpretar y manipular la información de forma apropiada.

En forma general (ya se verá posteriormente la clasificación para el lenguaje C++ en particular), las variables las podemos clasificar como:

- **Tipo Carácter** Si la variable es de tipo carácter, la información que contiene involucrará letras, dígitos, símbolos etc. (Ejemplo "Lunes", "Agente 007", etc)

- **Tipo Numérico** Como su nombre lo indica, si la variable es de tipo numérico, la información que contiene involucrará sólo números. A su vez, los caracteres de tipo numérico pueden clasificarse como:

    1. **Enteros** Ejemplo: 0, 1, 2, 5,678, etc.

2. **Punto Flotante** Ejemplo: 3.454, 0.568954, 2.3, etc.

- **Tipo Booleano** Son aquellas variables que sólo pueden tener "Valer" Falso o Verdadero. Este tipo de variables son importantes en problemas con operaciones lógicas.

ELEMENTOS BÁSICOS DE C++: DECLARACIÓN DE VARIABLES

A través de operaciones aritméticas (suma, resta, etc.) y lógicas (por ejemplo, comparaciones) los programas manipulan datos tales como números y caracteres. C++ y prácticamente todos los lenguajes de programación utilizan una estructura de lenguaje conocida como "**variable**" para almacenar e identificar la información. El término de variable es muy utilizado en matemáticas para representar una cantidad que debe ser calculada o cuyo valor puede cambiar durante algún procedimiento de cálculo. El concepto es muy similar en programación, la diferencia principal, sin embargo, es que en programación las variables pueden representar no sólo números, sino también caracteres o listas de números y caracteres. De esta forma, a la cantidad o símbolo que una variable representa se le conoce como el "**valor**" de una variable. Por ejemplo, si una variable sirve para almacenar información de tipo numérico un valor *válido* para esa variable pudiera ser el número 2, pero si la variable almacena una cadena de caracteres, un *valor* válido para la variable puede ser cualquier palabra ("libro", "manzana", etc.)

El compilador (en este caso el compilador de C++) asigna una posición de memoria a cada variable de programa. Así, el valor de la variable, en formato binario, se guarda en la posición de memoria asignada a la variable.

Identificadores

Al nombre de una variable se le conoce como **identificador**. Es deseable que, para que los programas sean fáciles de entender, los nombres de las variables estén asociados con su significado. Un identificador debe comenzar con una letra o con el guión bajo (_), mientras que el resto del identificador pueden ser cualquier combinación de letras, dígitos y el guión bajo. Por ejemplo, los siguientes son identificadores válidos para las variables:

x x1 x_1 _abc ABC123z7 suma datos2 area Producto

aunque sólo tres de ellos son nombres que adecuadamente indican su significado. Los siguientes no son identificadores válidos ¿Por qué?

12 3x %cambio data-1 miprimer.c programa.cpp

Hay una clase especial de identificadores conocido como **palabras reservadas** (*keywords*). Estas palabras reservadas tienen ya un significado definido por el lenguaje de programación (C++), de forma que el programador no puede utilizarlos para identificar variables. Las palabras reservadas en el lenguaje C++ están dadas en la siguiente lista:

asm	and	and_eq	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	
reinterpret_cast		return	short	signed	sizeof
static	static_cast	struct	switch	template	this
throw	true	try	typedef	typeid	typename
union	unsigned	using	virtual	void	volatile
wchar_t	while	xor	xor_eq		

Declaración de Variables

Todas las variables de un programa deben de ser **declaradas**. Cuando se declara una variable se está indicando al compilador (y a la computadora) que tipo de información se está almacenando en esa variable. Por ejemplo, las siguientes dos declaraciones sirven para indicar el tipo de las 3 variables de un programa:

```
int     numero_de_cajas;
double  peso_por_caja, peso_total;
```

La palabra *int* en la primera declaración es una abreviación de *integer*, de forma que la primera declaración indica que la variable *numero_de_cajas* es de tipo entero. Observe que en la segunda declaración se declaran dos variables. Cuando se declaran más de una variable en una declaración, las variables se separan por **comas**. Observe también que cada declaración termina con un **punto y coma**. En la segunda declaración la palabra double indica que las variables *peso_por_caja* y *peso_toal* son de tipo *doble* (punto flotante). Posteriormente se indica el significado de doble. La clase de información que una variable almacena se le conoce como su **tipo**. A las palabras int y double se les conoce como **nombre del tipo**.

En general, la sintaxis de un declaración de variables es:

Nombre_del_tipo *Identificador_1*, *Identificador_2*, ... ;

Recordar que la *sintaxis* de un lenguaje de programación es el conjunto de reglas gramaticales definidas para ese lenguaje.

Tipos de Variables

Tipo Numérico

Como se describió anteriormente, las variables numéricas en un programa pueden ser de dos tipos, *enteras* y de *punto flotante*. Por ejemplo, conceptualmente los números 2 y 2.0 son iguales, pero los lenguajes de programación como C++ les consideran que tiene tipo distinto. El número 2 es un entero, mientras que el número 2.0 es de punto flotante porque contiene una parte fraccionaria (aun cuando este parte fraccionaria vale cero en este caso). Existen muchos tipos de variables numéricas en C++. La clasificación depende, entre otras cosas, del rango de valores que una variable puede poseer. Recuerde que una computadora tiene limitaciones en su capacidad de memoria, de forma que los números son almacenados utilizando un número limitado de bytes. Así, hay un límite para el valor que una variable numérica

puede tener, y ese valor depende del tipo. Es decir, la computadora utiliza diferente número de bytes para guardar cada variable y ese número de bytes depende de su tipo.

La siguiente tabla proporciona información a cerca de los tipos numéricos del lenguaje C++

Nombre del Tipo	Memoria usada	Rango de Valores	Precisión
short	2 btytes	-32767 a 32767	
int	**4 bytes**	**-2147483647 a 2147483647**	
long	4 bytes	-2147483647 a 2147483647	
float	4 bytes	10^{-38} a 10^{38}	7 dígitos
double	**8 bytes**	**10^{-308} a 10^{308}**	**15 dígitos**
long double	10 bytes	10^{-4932} a 10^{4932}	19 dígitos

En la tabla, los tipos *short*, *int* y *long* representan tipos de variables enteras. Los tipos *float*, *double* y *long double* son variables de punto flotante (poseen un punto decimal). Aun cuando C++ te permite dicha clasificación, para la mayoría de los programas que se utilizarán los únicos tipos que se necesitan son los tipos *int* y *double*.

Tipo Caracter

Debido a que las computadoras no son usadas únicamente para hacer cálculo numéricos, mostramos aquí un tipo de variable no numérica. Las variables del tipo ***char*** son variables cuyo valor consiste de letras, dígitos o signos de puntuación. Una variable de tipo char puede contener un solo caracter. Hay un tipo de cadenas de caracteres (*string*) pero éste se analizará posteriormente en el curso por varias razones. Ejemplo de uso:

char simbolo_1;

Tipo Booleano

Variables de tipo booleano (nombradas así en honor a George Boole quien creó las reglas de la lógica matemática) únicamente pueden tener los valores de *true* (verdadero) o *false* (falso). El tipo booleano en lenguaje C++ se representa por *bool*. La siguiente es una declaración de la variable *verdad* como de tipo booleano:

bool verdad;

SENTENCIAS DE ASIGNACIÓN E INICIALIZACIÓN

A cada instrucción de un programa se le conoce como **Sentencia**. En este documento se discuten las sentencias de asignación.

Sentencias de Asignación

La forma más directa de cambiar el valor de una variable es a través de una sentencia de asignación o simplemente **asignación**. Un asignación es una orden que se da a la computadora y que en palabras se interpretaría como:

"*Evalúa la expresión escrita al lado derecho del signo de igualdad y asígnale tal valor a la variable colocado al lado izquierdo del signo igual*"

La sintaxis de una asignación es:

nombre_de_variable **=** *expresion*;

Observe que al igual que otras sentencias en C++, una asignación termina con un **punto y coma** y que los dos términos de la asignación están separados por un **signo igual**. El término *expresion* colocado al lado derecho del signo igual se evalúa y el resultado se asigna a la variable colocado en el lado izquierdo del signo igual.

El término *expresion* puede ser muy simple, como un valor numérico o el nombre de otra variable, pero también puede contener operaciones algebraicas (suma, resta, etc.) entre variables del programa y funciones matemáticas (seno, coseno, raíz cuadrada, etc.)

Los siguientes son ejemplos de asignaciones:

```
peso_por_caja = 10;
```

*peso_total = peso_por_caja * numero_de_cajas;*

peso_total = peso_total + 1;

*distancia = velocidad * tiempo;*

Expresiones Usando Operadores Aritméticos

En un programa de C++ es posible combinar variables entre sí (o con números constantes) a través de operadores aritméticos:

+ Suma

- Resta

***** Multiplicación

/ División

También es posible incorporar funciones matemáticas más complicadas para relacionar variables, pero esas se analizarán un poco después en el curso.

Observa que en los tres últimos ejemplos de asignación se utilizan los operadores *multiplicación* y *suma*.

Inicialización de Variables

Una variable de un programa no tiene un valor con sentido hasta que se le asigna uno en el programa. Pongamos el siguiente ejemplo. Suponga que se tiene un programa como el siguiente:

```
int main ()
{
  int   numero1, numero2;
  numero1 = numero2 + 1;
 return 0;
}
```

¿Qué valor se le asigna a la variable *numero1* ? La respuesta es "*no se sabe*". El problema del programa anterior es que la variable *numero2* no ha sido **inicializada**, es decir, no se le ha asignado un valor que tenga significado dentro del contexto del programa. Cuando esto ocurre el compilador posiblemente no marque error (algunos compiladores marcan este tipo de errores pero otros no lo hacen), pero el resultado que el programa proporciona para la variable numero2 seguramente carecerá de sentido. El valor de una variable no inicializada corresponde al valor binario que la computadora tenía almacenada en el espacio de memoria que ahora le ha asignado a la variable. Generalmente este será un valor "*basura*" dado que no tendrá significado en el programa.

Hay dos formas de inicializar una variable, una es a través de alguna asignación en el programa; la otra es inicializarla al mismo tiempo que se declara. Los dos casos siguientes son equivalentes:

```
int contador, indice;
contador = 1;
indice = 2;
```

o bien

```
int contador = 1, indice=2;
```

La sintaxis de este último caso sería:

```
Nombre_del_Tipo    identificador_1 = valor_1,
                   identificador_2 = valor_2, ... ;
```

SENTENCIAS DE ENTRADA Y SALIDA DE INFORMACIÓN

La forma básica de un programa C++ para enviar mensajes o valores de salida del programa es a través de la instrucción **cout**. De la misma forma, la forma básica de recibir valores de entrada para una variable es a través de la instrucción **cin**.

Salida Usando cout

Los valores de una variable o mensajes formados por caracteres pueden enviarse a pantalla a través de la instrucción cout. La sintaxis es la siguiente:

cout<<" *Se escribe aqui el mensaje* ***";***
cout<< *nombre_de_variable* ***;***

Observe que en ambos casos se utiliza *cout*<< y ambos casos terminan con punto y coma. Note que cuando se envía sólo una cadena de caracteres (frases, palabras, etc.), ésta se escribe entre comillas. Por otra parte, cuando se quiere desplegar el valor de una variable, sólo se escribe el nombre de la variable, sin comillas. Al símbolo << se le conoce como operador *insertar* u operador de inserción.

Es posible combinar varios mensajes y valores de variables a través de una sola instrucción cout, pero para ello es necesario utilizar varios operadores de inserción, como en el siguiente ejemplo:

cout<< "El valor de la primera variable es "<<var1 << " y el de la segunda es " << var2;

Dentro de los mensajes que se insertan entre las comillas de la instrucción cout es posible introducir símbolos con significado especial. Todos ellos se valen del símbolo ****. Los más comunes son los siguientes:

\n Nueva línea
\t Tab

Por ejemplo, la instrucción

cout<<"Mi nombre es : \n";

Escribe la frase *Mi nombre es :* y luego se salta de línea. Así, si quieres insertar una línea en blanco se puede utilizar la instrucción:

cout<<"\n";

Entrada de Datos usando cin

La instrucción cin asigna a alguna variable un valor proporcionado por medio del teclado. La sintaxis de una sentencia que utilice *cin* es la siguiente:

cin>> *nombre_de_variable****;***

Observe que la sentencia termina con un punto y coma. Observe también que se utiliza el símbolo >> (dirección contraria a la utilizada por cout) que se conoce como operador de extracción u operador ***extraer***.

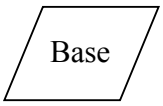
Cuando se esté ejecutando un programa, una instrucción *cin* hará que el programa se detenga solicitando al usuario del programa que proporcione un valor a través del teclado. El usuario tendrá entonces que teclear el valor y presionar la tecla ***Enter***. El programa no lee los valores hasta que la tecla Enter es presionada. Para proporcionar datos sólo es necesario utilizar la instrucción cin. Sin embargo, la instrucción cin no envía por sí sola ningún mensaje a pantalla. Es por ello que la opción cin por sí sola sería difícil de usar, pues el usuario del programa tendría que saber el orden en el cual tiene que proporcionar los datos del programa. Para solucionar este problema, la lectura de datos se hace generalmente a través de una combinación de cout y cin, de la forma siguiente:

cout<< "Dame el valor del area";
cin>> area;

Con estas instrucciones, el programa enviaría a pantalla el mensaje y se detendría para esperar a que el valor de la variable *area* le sea proporcionado por medio del teclado. Por otro lado, el usuario del programa, al leer el mensaje, sabría que el dato que tiene que proporcionar en ese momento es el area y sólo tendría que teclearlo y presionar la tecla Enter para que el programa continúe ejecutándose. En pocas palabras, por ejemplo, existiría una equivalencia entre símbolos del diagrama de flujo e instrucciones en C++ de la forma siguiente:

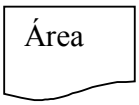
1)

double base, area;



cout<< "Dame la base del rectángulo \n";
cin>> base;

2)



cout<< "El area es " << area;

Se pueden solicitar al usuario del programa los valores de muchas variables utilizando un sólo cin. Esto se hace de la forma siguiente:

cin>> identificador_1 >> identificador_2 >> ... ;

Observe que en ese caso se necesitan tantos operadores de extracción como variables se desea solicitar. Nuevamente, la sentencia termina en punto y coma. En tal caso, habría que presionar la tecla Enter después de proporcionar cada valor.

EXPRESIONES ARITMÉTICAS: ORDEN DE PRECEDENCIA

Como se ha discutido anteriormente, en un programa C++ es posible manipular variables y números usando los operadores aritméticos de suma, resta, multiplicación y división (y funciones matemáticas como seno, coseno, potencia, etc.). Cuando las expresiones aritméticas de un programa son complejas e involucran varias instancias de estas operaciones, el procesador de la computadora tendrá que decidir en que orden se deberán ejecutar dichas operaciones. Esta decisión se realiza con base en lo que se conoce como **Reglas de Precedencia**.

Reglas de Precedencia

Las reglas de precedencia permiten a un procesador decidir el orden en que las diferentes operaciones aritméticas son ejecutadas. Estas reglas son similares a las usadas en álgebra. Para las operaciones más comunes el orden de precedencia es:

Función (raíz cuadrada, seno, coseno, etc.)	Alta precedencia ↓
* Multiplicación / División	
+ Suma - Resta	Baja precedencia

Es de hacer notar que las operaciones en un mismo renglón de la tabla tienen la misma precedencia. Cuando hay operaciones de la misma precedencia se ejecutan en orden izquierda a derecha. Como ejemplo, recuerde el programa que se realizó como ejercicio en el cual se solicitó definir:

promedio = numero_1 + numero_2 + numero_3 / 3;

Si se toma en cuenta el orden de precedencia, la sentencia anterior **no** da como resultado el promedio de tres números. La división tiene un orden de precedencia mayor que la suma. Por esa razón, antes de sumar los tres números, el procesador ejecutaría la división de numero_3 / 3 y a ese resultado le sumaría el numero_1 y el numero_2. Si, por ejemplo, numero_1 = 10, numero_2 = 5 y numero_3 = 9, el promedio sería 8. Sin embargo, el resultado de la sentencia sería:

promedio = 10 + 5 + 3 = 18

Es de hacer notar, sin embargo, que el uso de **paréntesis** en las expresiones aritméticas permite modificar el orden de precedencia de los operadores. Si la sentencia del programa se modifica de forma que:

promedio = (numero_1 + numero_2 + numero_3) / 3;

entonces el resultado sería el correcto. El procesador trataría de ejecutar la división, pero en este caso el numerador, debido a la presencia del paréntesis, es la suma de los tres números y no sólo el numero_3.

Como ejercicio, para las siguientes fórmulas matemáticas, encuentre la expresión correspondiente en C++:

$b^2 - 4ac$

$x(y+z)$

$$\frac{1}{x^2+x+3}$$

$$\frac{a+b}{c-d}$$

Mezcla de Tipos

Los operadores aritméticos pueden utilizarse para manipular tanto variables de tipo entero (*int*) como variables de tipo de punto flotante (*double*). Si un operador relaciona dos variables enteras, el resultado será un número entero. Asimismo, si un operador relaciona dos variables de punto flotante, el resultado es un número de punto flotante. Los operadores pueden utilizarse, sin embargo, también para relacionar dos variables de distinto tipo entre sí. En ese caso, si una de las dos variables es double, el resultado será double.

Como ejemplo, ¿cuál es el valor de precio_total luego de ejecutarse las siguientes sentencias?:

```
double precio_total;
precio_total = 7/2;
```

La relación entre dos números enteros es un entero, 7/2 = **3**. Como complemento a dicha operación, el operador % permite obtener el residuo de una división:

```
residuo = 7%2;
```

Luego de ejecutarse dicha sentencia, residuo es igual a 1.

Como contraparte:

```
double precio_total;
precio_total = 7.0/2;
```

da como resultado precio_total = 3.5. Note que 7.0 es de punto flotante y 2 es entero.

EJERCICIO

Suponga que el calor específico (Cp) de una sustancia puede calcularse a partir de la expresión:

$$C_p = \frac{a + bT}{c}$$

donde T es la temperatura en grados Kelvin y a, b y c son tres parámetros definidos para cada sustancia en particular.

Escriba un programa en C++ que ejecute el cálculo del Cp utilizando la expresión anterior.

EJERCICIO

Definición de Problema

Dados 3 números, se desea crear un programa que calcule su suma, el promedio y el producto entre ellos y que muestre los resultados en pantalla. Los números dados pueden tener parte fraccionaria.

Análisis del Problema

Datos: numero_1, numero_2, numero_3

Resultados: suma, promedio, producto

Formulaciones requeridas:

suma = numero_1 + numero_2 + numero_3

promedio = suma / 3

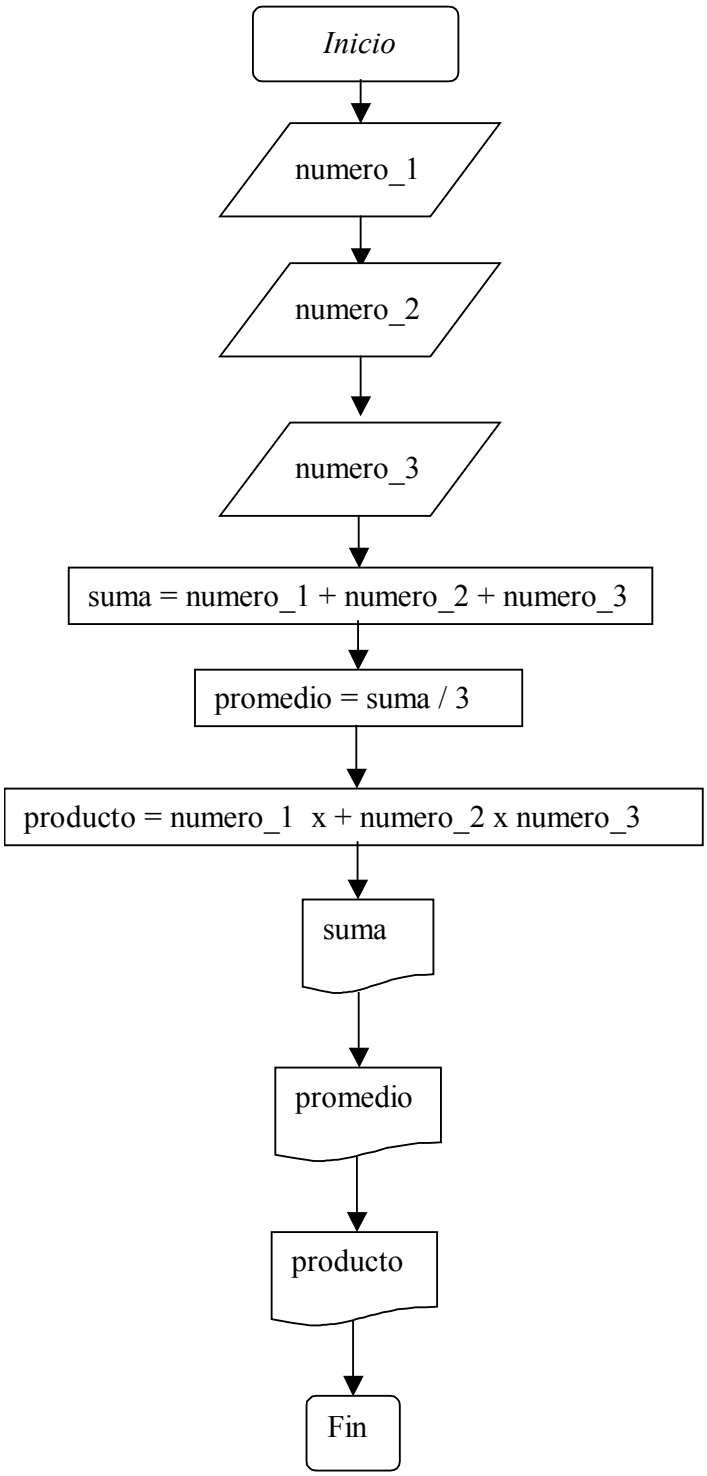
producto = numero_1 x numero_2 x numero_3

Tipo de datos:

Como los tres números pueden tener parte fraccionaria, deben de considerarse de tipo numérico de punto flotante. Como las otras cantidades (suma, promedio, producto) se calculan usando números de punto flotante, entonces también deben ser declaradas como variables numéricas de punto flotante.

Creación del Diagrama de Flujo

El diagrama de flujo necesario para resolver el problema se muestra en la página siguiente.


Inicio
numero_1
numero_2
numero_3
suma = numero_1 + numero_2 + numero_3
promedio = suma / 3
producto = numero_1 x + numero_2 x numero_3
suma
promedio
producto
Fin

Codificación

La codificación del diagrama de flujo corresponde al siguiente programa en C++:

```
/* Este es un programa sencillo que permite el
   calculo de la suma, el promedio y el producto
   de tres numeros.
   Creado por:
   22 de Septiembre de 2004
 */

#include <iostream.h>
#include <stdlib.h>
int main()
{
     /* Declaracion de Variables */

     double numero_1, numero_2, numero_3;
     double suma, promedio, producto;

     /* Entrada de Datos */
     cout<< "Dame el valor del primer numero \n";
     cin>> numero_1;

     cout<< "Dame el valor del segundo numero \n";
     cin>> numero_2;

     cout<< "Dame el valor del tercer numero \n";
     cin>> numero_3;

     /* Procesamiento de Datos*/
     suma = numero_1 + numero_2 + numero_3;
     promedio = suma / 3.0;
     producto =  numero_1 * numero_2 * numero_3;

     /* Salida de Resultados */
     cout<< "\n";
     cout<< "La suma de los numeros es " << suma <<" \n";
     cout<< "El promedio de los numeros es " << promedio <<" \n";
     cout<< "El producto de los numeros es " << producto <<" \n";
     cout<< "\n";

     system("PAUSE");
     return 0;
}
```

UNIDAD II

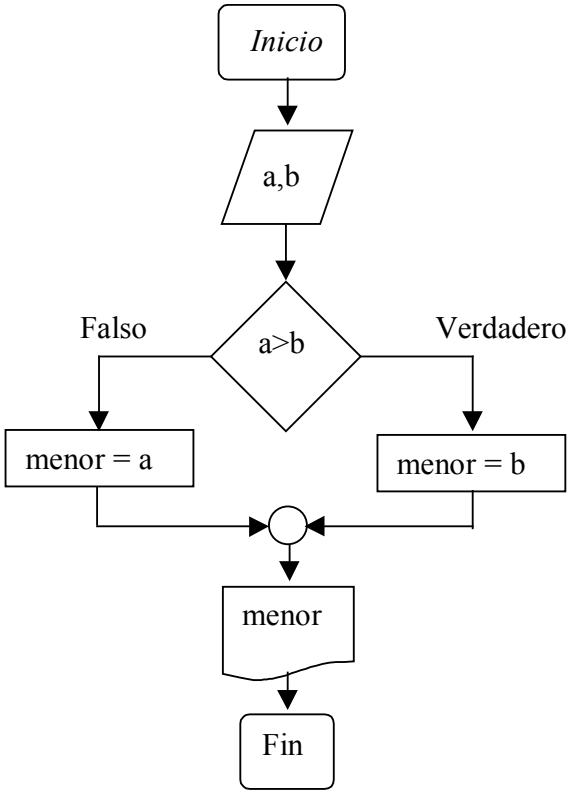
ESTRUCTURAS DEL LENGUAJE Y SUS ESTATUTOS

SENTENCIAS CONDICIONALES SIMPLES: if-else

Anteriormente se discutió que una de las estructuras utilizadas en la programación estructurada es la **Estructura Selectiva** o Condicional. Se explican aquí las sentencias que se utilizan en el lenguaje C++ para representar este tipo de estructuras.

Proposiciones Condicionales

Algunas veces es necesario que un programa seleccione entre alternativas dependiendo de los valores de algunas variables. Recordemos el ejercicio de tarea acerca del diagrama de flujo que sirvió para encontrar el menor de dos números:



Observe la interpretación que, en palabras, se podría dar a la estructura condicional de dicho ejemplo:

si *a>b* ***entonces***

menor = b

si no es así

menor = a

Existe una sentencia en C++ que permite este tipo de expresiones. Se trata de la sentencia **if-else**. (if = si, else = de otra forma, por el contrario, etc.) Para el ejemplo que se está analizando, la sentencia sería:

if (*a>b****)***

menor = b;

else

menor = a;

Esta es la forma más simple de una sentencia *if-else*. A la expresión entre paréntesis se le denomina **Expresión Booleana** y, al igual que una variable booleana, al evaluarla se tiene como resultado al valor de ***Falso*** o ***Verdadero***. Cuando se ejecuta un programa que contiene una sentencia *if-else*, solamente las sentencias de una de las alternativas se considera. Si la expresión booleana es verdadera, la sentencia escrita por debajo de *if (expresion_booleana)* se ejecuta. Si la expresión booleana falsa, solamente la sentencia por debajo de *else* es la que se ejecuta.

En el ejemplo, cada una de las alternativas contiene solamente una sentencia (menor=b y menor=a, respectivamente). Sin embargo, cada alternativa puede presentar varias sentencias.

En general, la sintaxis de una sentencia condicional *if-else* es la siguiente.

Una sola sentencia en cada alternativa:

```
if (expresion_booleana)
    sentencia_de_verdadero;
else
    sentencia_de_falso;
```

Observe que las líneas correspondientes a if y a else **no llevan punto y coma**.

Cuando se tienen más de una secuencia en cada alternativa:

```
if (expresion_booleana)
{
   sentencia_1_de_verdadero;
   sentencia_2_de_verdadero;
                 :
   ultima_sentencia_de_verdadero;
}
else
{
    sentencia_1_de_falso;
    sentencia_2_de_falso;
                 :
    ultima_sentencia_de_falso;
}
```

Las sentencias en cada uno de las alternativas pueden ser cualquier sentencia ejecutable de C++. Cuando se tiene una situación como ésta, en la cual hay una lista de sentencias entre llaves, se dice que se tiene una **sentencia compuesta**.

Expresiones Booleanas

Recordemos que la expresión boolena de una sentencia *if-else* deber ser encerrada entre paréntesis. La forma más simple de una expresión booleana consiste de dos números o variables que son comparadas entre sí a través de

algún operador. Los operadores utilizados para comparación están constituidos por uno o dos símbolos. Los operadores son los siguientes:

Operador en Matemáticas	Español	Operador en C++	Ejemplo
=	Igual a	==	*edad==18*
≠	Desigual a	!=	*var_1 != 1*
<	Menor que	<	*contador < 20*
≤	Menor ó igual que	<=	*suma <= 15*
>	Mayor que	>	*a > b*
≥	Mayor ó igual que	>=	*tiempo >= limite*

Para escribir los operadores de dos símbolos no debe de dejarse espacio entre ellos. Un error común es utilizador un solo símbolo = para comparar si dos variables o números son iguales. Se debe de asegurar de usar dos símbolos. Recuerde que un solo símbolo = se usa para asignaciones. Observe que el resultado de una comparación será el valor de falso o verdadero.

Es posible combinar varias comparaciones a través de los operadores "*and*" ("y" en español) y "*or*" ("o" en español). En C++ el operador "and" se representa como **&&**, mientras que el operador "or" se representa como **||**. Por ejemplo, la siguiente expresión evalúa si el valor de la variable x es mayor que 2 y menor que 7:

(2 < x) && (x < 7)

Debido al operador "and", la expresión sólo tiene el valor de verdadero **si ambas** comparaciones son verdaderas.

Por otro lado, la siguiente expresión evalúa si el valor de la variable y es menor a 0 o mayor que 12:

(y < 0) || *(y > 12)*

Debido al operador "or", la expresión anterior es verdadera **si alguna** (o las dos) de las comparaciones es verdadera. Hay que recordar que cuando se usa una expresión booleana en una sentencia *if-else*, toda la expresión debe de usarse entre paréntesis. Por ejemplo, la siguiente es la primera línea de una sentencia *if-else*:

if ((temperatura > 35) && (humedad > 85) **)**

Existe además un símbolo especial que sirve para obtener el *valor contrario* de una expresión booleana. Se dice también que dicho símbolo sirve para obtener la "*negación*" de una expresión booleana. El símbolo es **!**. Por ejemplo, la expresión:

!(x > y)

En este caso, si x es mayor que y, el valor de la expresión *(x>y)* sería verdadero. Sin embargo, debido a la negación, el valor de la expresión completa *!(x > y)* sería falso.

Formalmente, la sintaxis para el uso de los operadores && y || es la siguiente:

(comparacion_1) && (comparacion_2)... && (ultima_comparacion)

(comparacion_1) || (comparacion_2)... || (ultima_comparacion)

Ambos operadores se pueden combinar utilizando el uso de paréntesis como en:

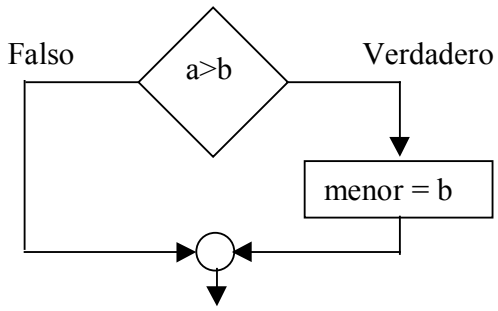
((comparacion_1) && (comparacion_2)) || *(comparacion_3)*

Ejemplos de uso de la sentencia *if-else*:

```
if (  (calificacion > 8.0) && (calificacion < 9.0)  )
   cout<< "La calificacion esta entre 8 y 9 \n";
else
    cout<< "La calificacion no esta entre 8 y 9 \n";
```

Notas

- Algunas veces se desea que una de las alternativas de una estructura *if-else* no ejecute ninguna instrucción. En ese caso, es posible utilizar la estructura condicional omitiendo la parte de la estructura que corresponde a *else*. En C++ ese tipo de estructuras se le conoce simplemente como sentencia **if**. Por ejemplo, la siguiente parte de un diagrama de flujo:



corresponde a la instrucción:

```
if  (a>b)
   menor = b
```

- Un error común es tratar de utilizar **operadores de comparación** en serie. Por ejemplo:
  if (x<z<y)
  Esto es incorrecto. La forma correcta sería:
  if ((x<z) && (z<y))

EJERCICIOS

1. Escriba una sentencia if-else que muestre en pantalla la palabra *Alto* si el valor de la variable *puntuacion* es mayor que 100, y que muestre la palabra *Bajo* si el valor de la variable puntuacion es menor o igual que 100. La variable puntuacion es de tipo *int*.

2. Escriba una sentencia *if-else* que muestre en pantalla la palabra *Aprobado* si el valor de la variable *examen* (variable de tipo *double*) es mayor o igual a 70 y el valor de la variable programas_entregados es mayor o igual a 8; en caso contrario, el programa muestra en pantalla la palabra *Reprobado*.

3. Suponga que se tienen dos variables de tipo *double* llamadas *ahorro* y *gasto*. Escriba una sentencia *if-else* que, si el valor de ahorro es mayor al valor de gasto, entonces muestre en pantalla la palabra *Solvente*, disminuya el valor de la variable ahorro mediante la resta del valor de gasto a su valor original, y asigne a la variable gasto el valor de cero. En caso contrario (si gasto es mayor que ahorro), simplemente debe mostrarse en pantalla la palabra *Quiebra*.

4. Considere dos variables de tipo *int* llamadas temperatura y presion. Escriba una sentencia *if-else* que muestre en pantalla la palabra *Alarma* si la variable presion es mayor a 200 o si la variable temperatura es mayor a 100. En caso contrario, se debe mostrar en pantalla la palabra *Normal*.

5. ¿Que muestra en pantalla la siguiente sentencia?

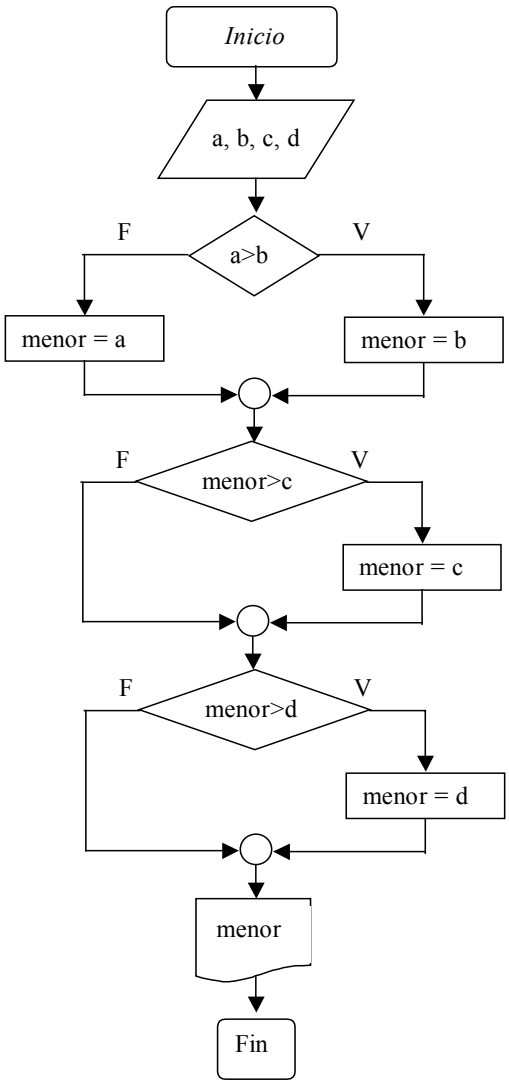
```
if (0)
  cout<< "0 es equivalente a verdadero  \n";
else
  cout<< "0 es equivalente a falso  \n";
```

6. Escriba un programa completo en C++ que encuentre el menor de tres números a, b y c.

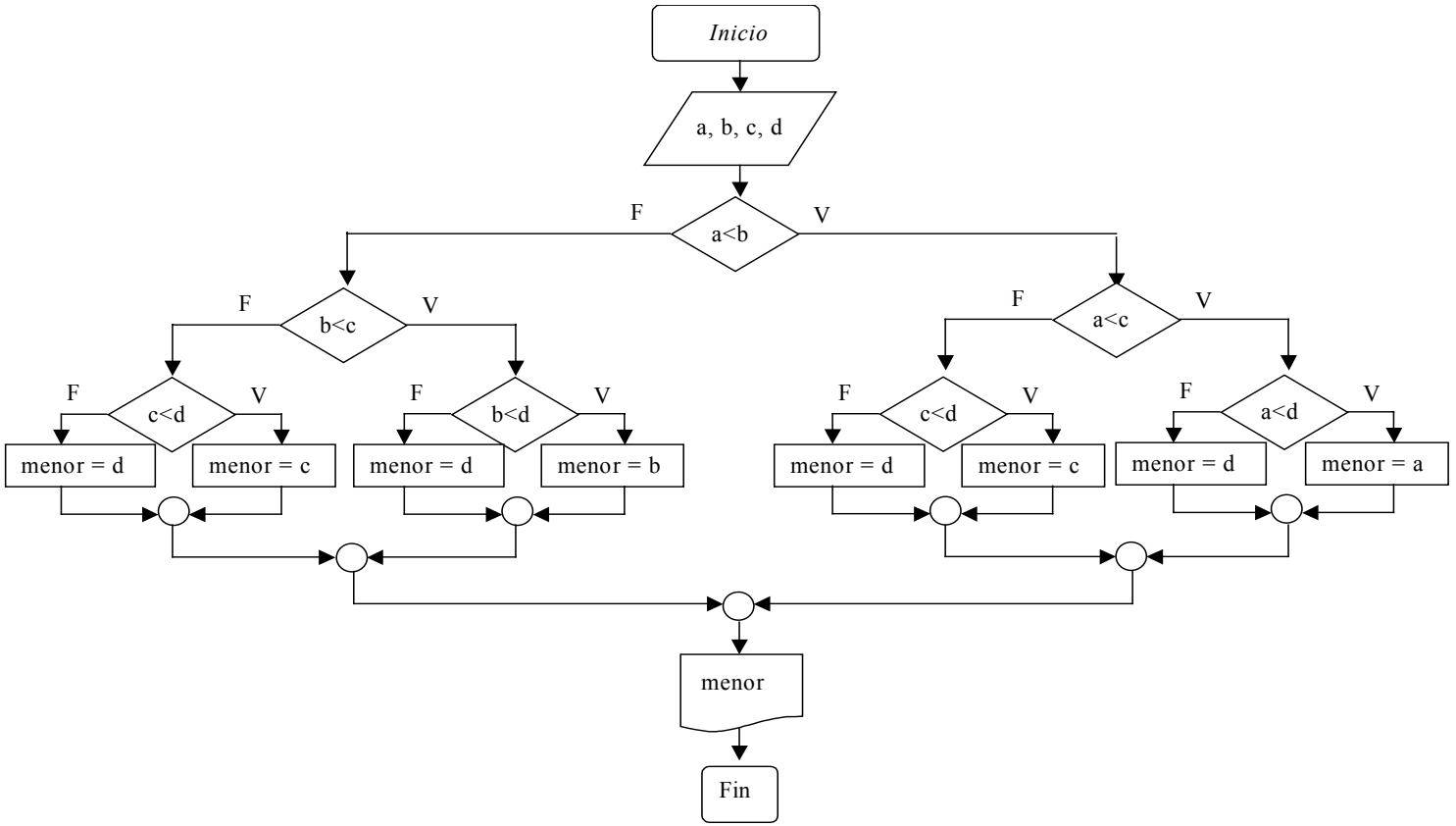
EJERCICIOS

Suponga que dados cuatros números (a, b, c y d), se desea determinar cual de ellos es el número menor y mostrar dicho número en pantalla. Los siguientes son dos diagramas de flujo que resuelven el problema. Elabore los programas en C++ que corresponden a cada uno de los diagramas. Suponga que los cuatro números pueden tomar valores numéricos de punto flotante.

Alternativa 1



Alternativa 2



EJEMPLO if-else

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

int main()
{
/* Este programa constituye un ejemplo de la aplicacion de
        las sentencias condicionales if-else. El programa encuentra
        y muestra en pantalla el menor de 4 numeros */

     /* Declaracion de Variables */
     double a, b, c, d, menor;

     /* Entrada de Datos*/
     cout<< "Dame los cuatro numeros a comparar.\n";
     cout<< "Presione enter despues de cada numero \n";
     cin>>a >>b >>c>>d;

     /* Procesamiento de Datos */
     if (a>b)
       menor = b;
     else
       menor = a;

     if (menor>c)
       menor = c;

     if (menor>d)
       menor = d;

     /* Salida de Resultados*/

     cout<<"\n";
     cout<<"El numero menor es "<<menor;
     cout<<"\n";

     system("PAUSE");
     return 0;
}
```

ESTRUCTURAS CÍCLICAS

Se discuten en este documento las sentencias que se utilizan en el lenguaje C++ para representar la tercera de las estructuras utilizadas en la programación estructurada: La **Estructura Repetitiva** o Iterativa.

Sentencias de Procesamiento Iterativo: while y do-while

En muchos programas será necesario ejecutar acciones en forma repetitiva. Una parte de un programa que repite una sentencia o un grupo de sentencias se denomina **ciclo**. El lenguaje C++ tiene varias formas de representar ciclos. Una de estas formas es a través de la sentencia ***while*** (o ciclo ***while***). Antes de crear el programa C++ para un ejemplo de relevancia, analicemos el siguiente ejemplo:

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

int main()
{
/* Este programa es un ejemplo de la utilización de
       las sentencias repetitivas while */

     int conteo;

     cout<< "Cuantas veces deseas que te diga Hola? \n";
     cin>>conteo;

     while (conteo > 0)
     {
         cout<<"Hola ";
         conteo = conteo – 1 ;
     }

    cout<< " \n";
    cout<< "Es todo \n";

    system("PAUSE");
     return 0;
}
```

La parte del programa que se encuentra en negritas y en mayor tamaño es un ejemplo de uso de la ***sentencia repetitiva while***. La traducción de while en español es ***mientras***. Por ello, la sentencia:

```
while (conteo > 0)
{
    cout<<"Hola ";
    conteo = conteo – 1 ;
}
```

puede entenderse como "*mientras conteo sea mayor a cero, ejecuta las sentencias entre llaves*".

Observe que luego de la palabra *while* se encuentra una comparación (expresión booleana) entre paréntesis. Por otra parte, hay una conjunto de sentencias que se encuentran encerradas entre llaves. A dicho grupo de sentencias se le conoce como el ***cuerpo de la sentencia*** while. Las sentencias entre llaves se repiten mientras la expresión booleana tenga el valor de verdadero. A cada repetición del ciclo se le conoce como **iteración**. Las sentencias entre llaves se ejecutan en el orden en que se escriben.

Observe tres casos de lo que se mostraría en pantalla con la ejecución de este programa:

Corrida 1:

```
Cuantas veces deseas que te diga Hola?
3
Hola Hola Hola
Es todo
Presiona cualquier tecla para continuar...
```

Corrida 2:

```
Cuantas veces deseas que te diga Hola?
1
Hola
Es todo
Presiona cualquier tecla para continuar...
```

Corrida 3:

```
Cuantas veces deseas que te diga Hola?
0

Es todo
Presiona cualquier tecla para continuar...
```

Analicemos el caso de la Corrida 1. Luego del mensaje:

Cuantas veces deseas que te diga Hola?

El usuario teclea el número *3 que,* a través de la instrucción cin se le asigna a la variable *conteo*. Inicia entonces el ciclo repetitivo. Cuando una sentencia while se ejecuta, lo primero que se realiza es que se evalúa el valor de verdad de la expresión entre paréntesis (***conteo>0***). Observe que al principio la variable conteo vale 3, por lo que *conteo>0* da como resultado el valor verdadero. Debido a que el valor de dicha expresión es verdadero, se ejecutarán las sentencias del cuerpo de la sentencia while (entre llaves):

```
cout<<"Hola ";
conteo = conteo – 1 ;
```

Por esta razón se imprimirá en pantalla la palabra Hola por primera vez. Luego de ello la el valor de la variable conteo se reduce en uno. Su valor cambia de 3 a 2:

conteo = 3 – 1 = 2

Con la llave } de la sentencia while se termina la primera iteración y comienza la siguiente. Nuevamente, lo primero que se hace es que se evalúa la expresión booleana. Dado que ahora la variable conteo vale 2, la expresión *conteo>0* sigue siendo verdadera y nuevamente se ejecutan las sentencias entre llaves. Por ello se mostrará nuevamente la palabra Hola y otra vez la variable conteo reduce su valor en uno:

conteo = 2 – 1 = 1

El procedimiento continúa en la misma forma hasta que, después de la tercera iteración, la variable conteo vale cero. Cuando esto ocurre, la expresión conteo>0 ya tiene el valor de falso y por tanto las sentencias entre llaves ya no se ejecutarán por cuarta vez.

En general, la sintaxis de una sentencia *while* es la siguiente.

Una sola sentencia en el ciclo:

```
while (expresion_booleana)
sentencia_del_ciclo;
```

Cuando se tienen más de una sentencia en el ciclo:

```
while (expresion_booleana)
{
   sentencia_1_del_ciclo;
   sentencia_2_del_ciclo;
                :
   ultima_sentencia_del_ciclo;
}
```

Observe que las líneas correspondientes a while y a las llaves **no llevan punto y coma**.

Existe una forma equivalente de expresar la sentencia while. Esto es a través de la sentencia **do-while**. La sintaxis de la sentencia do-while es como sigue:

Una sola sentencia en el ciclo:

```
do
   sentencia_del_ciclo;
while (expresion_booleana);
```

Cuando se tienen más de una sentencia en el ciclo:

```
do
{
   sentencia_1_del_ciclo;
   sentencia_2_del_ciclo;
                :
   ultima_sentencia_del_ciclo;
} while (expresion_booleana);
```

Observe que la sentencia do-while ***termina con punto y coma*** luego de la expresión booleana.

Es decir, es lo mismo escribir:

```
while (conteo > 0)
{
    cout<<"Hola ";
    conteo = conteo – 1 ;
}
```

que escribir:

```
do
{
    cout<<"Hola ";
    conteo = conteo – 1 ;
} while (conteo > 0);
```

Ejemplo: Cálculo del Factorial de un Número Entero

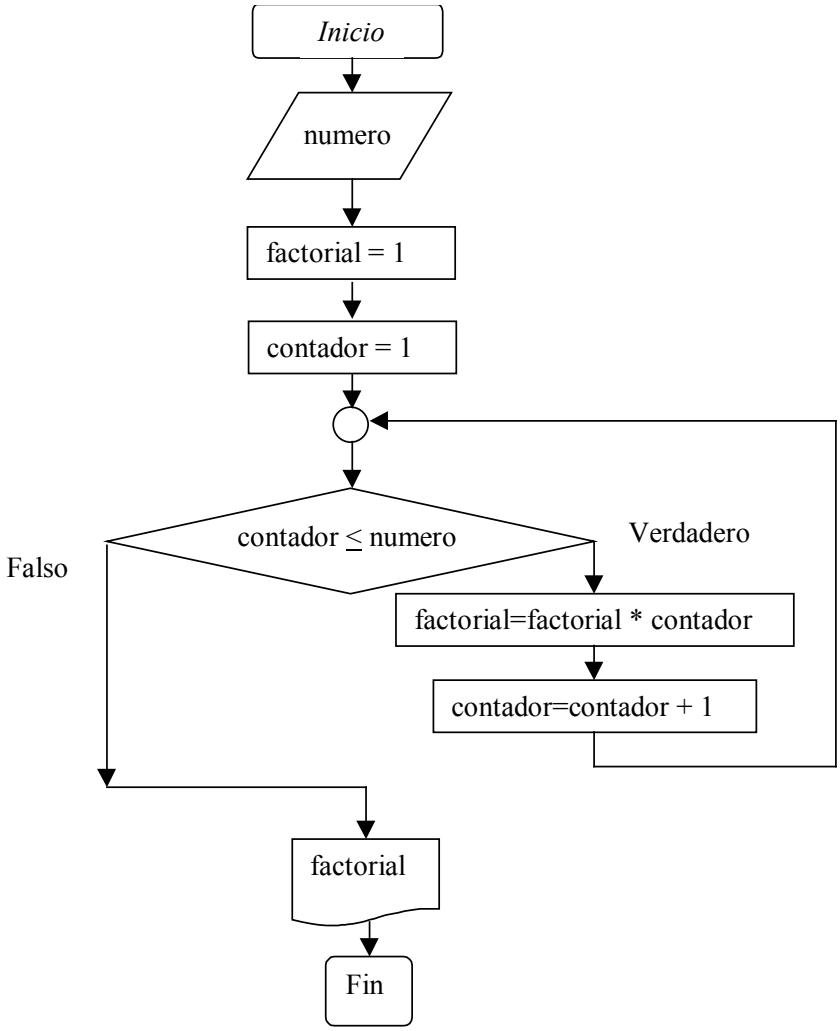
Anteriormente se analizó el diagrama de flujo para determinar el factorial de un número entero. Dicho diagrama de flujo se muestra en la página siguiente. Observe que un conjunto de instrucciones del diagrama de flujo se realizan en forma repetitiva mientras se satisface una condición. En este caso la condición es *contador*≤*numero*. Es decir, **mientras** *contador*≤*numero*, se ejecutarán las asignaciones:

```
factorial = factorial * contador;
contador = contador + 1;
```

El programa en C++ correspondiente a este diagrama de flujo es:

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

int main()
{
/* Este programa permite el calculo del factorial de un numero entero */

int numero, factorial contador;

    cout<< "Dame un numero entero \n";
    cin>>numero;

   factorial = 1;
```

```
    contador = 1;

    while (contador <= numero)
    {
        factorial = factorial * contador;
        contador = contador + 1 ;
    }
    cout<<"El factorial de "<< numero<< " es "<<factorial << "\n";

    system("PAUSE");
    return 0;
}
```



Nota Importante acerca de while y do-while

Es importante mencionar que con la sentencia *do-while*, las sentencias del ciclo **se ejecutan por lo menos una vez**, mientras que en la sentencia *while* las sentencias del ciclo pudieran no ser ejecutadas ni una sola vez, dependiendo de la expresión boleana. Es decir, cuando se ejecuta una sentencia do-while, la primera vez se ejecutan las sentencias del cuerpo de la sentencia y es hasta entonces que se evalúa la expresión boleana. El ciclo se repetirá sólo si la expresión boleana sigue teniendo el valor de verdadero. Por otro lado, en la sentencia while, aún la primera vez, lo primero que se evalúa es la expresión boleana y sólo si ésta es verdadera las sentencias del ciclo se ejecutarán. En otras palabras, las sentencias while y do-while son equivalentes sólo si el ciclo se ejecuta al menos una vez.

Operadores de Incremento y Decremento

Hasta ahora, los operadores aritméticos que se han estudiado (+, -, *, /) involucran dos operandos. Por ejemplo a+b, x*y, etc. Por ello, a dichos operadores se le conocen como operadores binarios. Se introducen aquí los operadores conocidos como operadores unarios. Los más comunes se representan como **++** (operador incremental) y **--** (operador decremental). Estos operadores se aplican sobre **una sola variable** y se aplican sobre variables **entera**s. El operador ++ incrementa el valor de una variable en uno. Mientras que el operador –- disminuye el valor de una variable en uno.
Por ejemplo:

```
n++;
m--;
```

Son sentencias ejecutables de C++. La primera sentencia hace que el valor de n aumente en uno. La segunda sentencia hace que el valor de m disminuya en uno. Es decir, las sentencias anteriores son equivalentes a:

```
n = n + 1;
```

m= m - 1;

Este tipo de operadores se utilizan muy comúnmente en ciclos. Por ejemplo, en el programa del cálculo del factorial, la segunda sentencia del ciclo:

contador = contador +1;

se pudo haber expresado como:

contador++;

Ciclos Infinitos

Las sentencias while o do-while no terminan su ejecución hasta que la expresión boleana que se evalúa (entre paréntesis después de la palabra *while*) es falsa. Por ello es que el ciclo contiene normalmente alguna asignación que permite cambiar el valor de verdad de la expresión boleana, de forma que, si al principio la expresión boleana es verdadera, llegará un momento en que su valor cambia a falso. Si la expresión boleana siempre es verdadera el ciclo continuará ejecutándose indefinidamente. En ese caso se dice que se tiene un **ciclo infinito**. Habrá que tener cuidado para evitar dicha situación.

Indentación

Existen algunas recomendaciones básicas para lograr un buen estilo de programación.

- Una de ellas es escribir comentarios.
- Otra es escribir en grupo a aquellos elementos que son considerados naturalmente como un grupo. Una forma de hacer esto es utilizar una nueva línea para separar aquellas sentencias que pueden considerarse como separadas.
- La tercera es que las sentencias encerradas entre las llaves de las sentencias compuestas *if-else*, *while* y *do-while* debería ser indentadas. Esto contribuye en mucho a la claridad del programa.

EJERCICIOS: while Y do-while

1. ¿Qué se muestra en pantalla si se ejecutan las siguientes sentencias (suponiendo que el resto del programa que no se presenta es correcto)?

```
int x;
x=10;
while (x>0)
{
  cout<< x <<"\n";
  x = x – 3;
}
```

2. ¿Qué se mostraría en pantalla si en el ejercicio anterior el signo > fuera reemplazado por <?
3. ¿Qué se mostraría en pantalla si se ejecutan las sentencias siguientes?

```
int x;
x=10;
do
{
  cout<< x <<"\n";
  x = x – 3;
} while (x>0);
```

4. ¿Qué se mostraría en pantalla si se ejecutan las sentencias siguientes?

```
int x;
x=-42;
do
{
  cout<< x <<"\n";
  x = x – 3;
} while (x>0);
```

5. ¿Qué se muestra en pantalla si se ejecutan las siguientes sentencias (suponiendo que el resto del programa que no se presenta es correcto)?

```
int x;
x=10;
while (x>0)
{
  cout<< x <<"\n";
  x = x + 3;
}
```

6. La siguiente sentencia *if-else* se compila y se ejecuta sin errores. Sin embargo, su escritura no refleja un buen estilo de programación. Re-escríbela de forma que se ajusta a la forma que se ha utilizado en los ejercicios.
   if (x<0) {x=7; cout<<"x es ahora positiva ";} else {x=-7; cout<<"x es ahora negativa";}

7. Suponga que se desea hacer una conversión de una distancia en metros a su equivalente en centímetros y pies. Escriba un programa que haga dichas conversiones, pero que además le permita al usuario del programa realizar el cálculo tantas veces como quiera.

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

int main()
{
    /* Este programa permite la conversión de metros a pies y a
       centímetros tantas veces como el usuario lo quiera*/

    /* Declaración de variables*/
    double distancia_metros, distancia_cm, distancia_ft;
    int repetir_o_no;

    /* Procesamiento */
    repetir_o_no = 1;

    while (repetir_o_no == 1)
    {
      cout<< "Dame una cantidad en metros \n";
      cin>> distancia_metros;

      distancia_cm = distancia_metros * 100;
      distancia_ft = distancia_metros * 3.048;

      cout<<"\n";
      cout<<"Su equivalente en centimetros es "<<distancia_cm<<" \n";
      cout<<"Su equivalente en pies es "<<distancia_ft<<" \n";

      cout<<"\n";
      cout<<"Deseas realizar otro calculo similar? \n";
      cout<<"Escribe el numero 1 si lo deseas, si no, \n";
      cout<<"escribe cualquier otro numero entero \n";
      cin>>repetir_o_no;
      cout<<"\n";
    }

    system("PAUSE");
    return 0;
}
```

EJERCICIO

Definición de Problema

Se desea hacer una conversión de una distancia en metros a su equivalente en centímetros y pies. Se solicita desarrollar programas que hagan dichas conversiones, pero que además permitan al usuario realizar el cálculo tantas veces como quiera.

Análisis del Problema

Datos: distancia_en_metros

Resultados: distancia_en_ft, distancia_en_cm

Formulaciones requeridas:

distancia_en_cm = 100 * distancia_en_metros

distancia_en_ft = 3.048 * distancia_en_metros

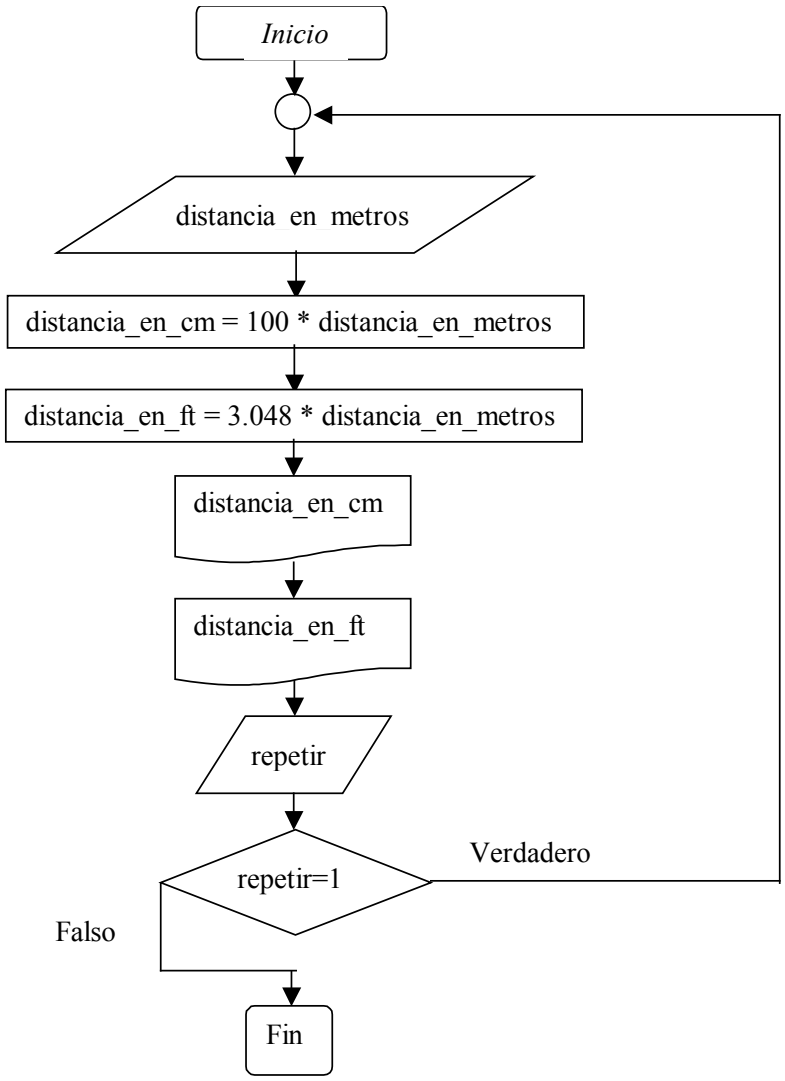
Observación: El programa requiere una estructura iterativa para que exista la posibilidad de repetir el cálculo.

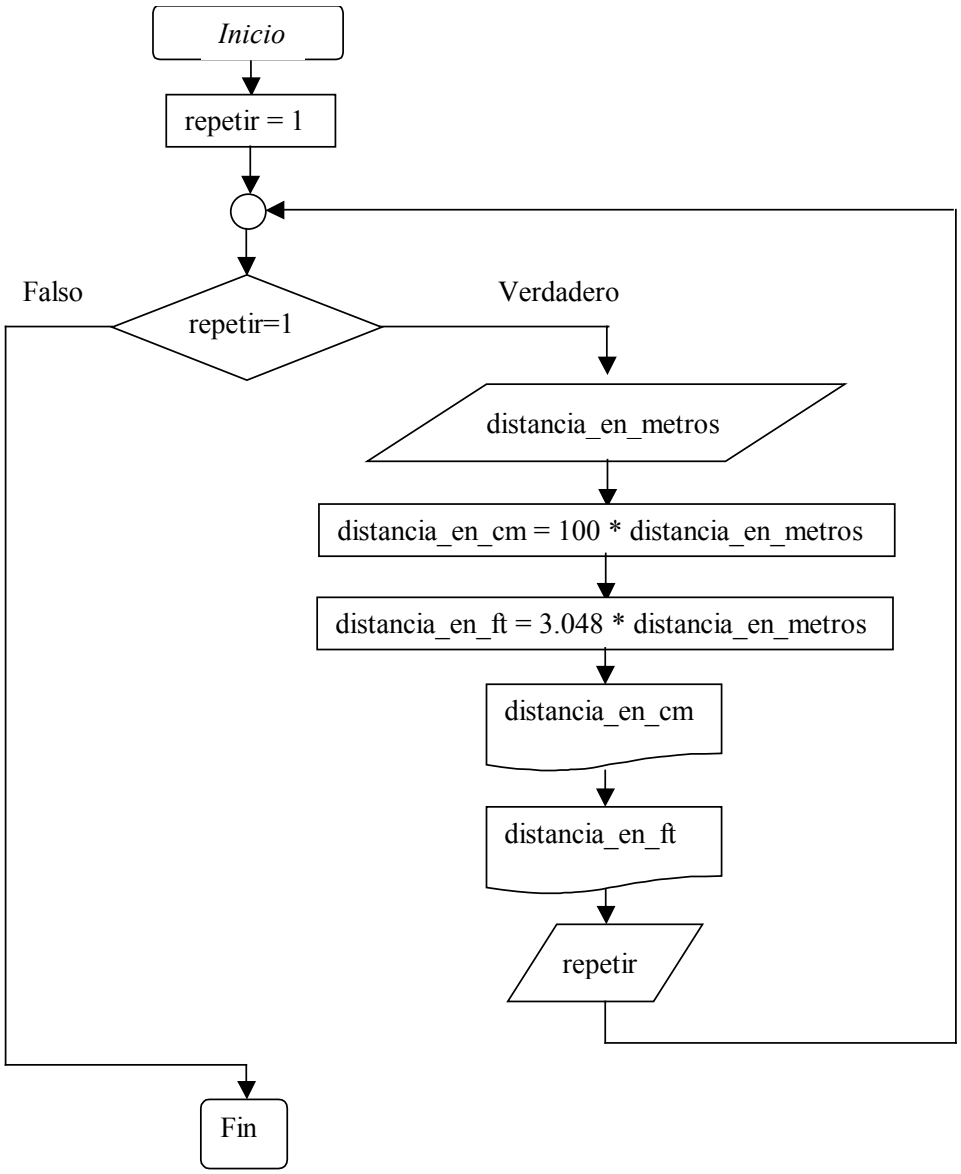
Tipo de datos:

Como los tres valores de distancia pueden tener parte fraccionaria, deben de considerarse de tipo numérico de punto flotante. Por otra parte, es necesaria otra variable que se pueda usar en la expresión booleana del ciclo iterativo con al finalidad de decidir si el ciclo se repite o no. Esta variable puede ser ya sea un entero o un caracter.

Creación del Diagrama de Flujo

El diagrama de flujo para resolver el problema se muestra a continuación. Este diagrama representa el uso de la sentencia *do-while*. En el diagrama de flujo que se muestra en la página siguiente se representa el uso de la sentencia *while*.


Inicio
distancia en metros
distancia_en_cm = 100 * distancia_en_metros
distancia_en_ft = 3.048 * distancia_en_metros
distancia_en_cm
distancia_en_ft
repetir
repetir=1
Verdadero
Falso
Fin


Inicio
repetir = 1
Falso
repetir=1
Verdadero
distancia_en_metros
distancia_en_cm = 100 * distancia_en_metros
distancia_en_ft = 3.048 * distancia_en_metros
distancia_en_cm
distancia_en_ft
repetir
Fin

Codificación

La codificación del primer diagrama de flujo corresponde al siguiente programa en C++:

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

int main()
{
    /* Este programa permite la conversión de metros a pies y a
       centimetros tantas veces como el usuario lo desee.
       Ejemplifica la aplicacion de la sentencia do-while */

    /* Declaración de variables*/
    double distancia_en_metros, distancia_en_cm, distancia_en_ft;
    int repetir;

    /* Procesamiento */
     do
    {
       cout<< "Dame una cantidad en metros \n";
       cin>> distancia_en_metros;

       distancia_en_cm = distancia_en_metros * 100;
       distancia_en_ft = distancia_en_metros * 3.048;

       cout<<"\n";
       cout<<"Su equivalente en centimetros es "
            <<distancia_en_cm<<" \n";
       cout<<"Su equivalente en pies es "
            <<distancia_en_ft<<" \n";

       cout<<"\n";
       cout<<"Deseas realizar otro calculo similar? \n";
       cout<<"Escribe el numero 1 si lo deseas, si no, \n";
       cout<<"escribe cualquier otro numero entero \n";
       cin>>repetir;
       cout<<"\n";
    } while (repetir == 1);

    system("PAUSE");
    return 0;
}
```

Práctica

a) Escriba el programa de la página anterior en Dev-C++ y verifique sus resultados.
b) Modifique el programa de forma que en lugar de ser de tipo entero, la variable repetir sea ahora de tipo caracter.
c) Modifique el programa del inciso b) de forma que, en lugar de usar la sentencia do-while, se utilice ahora la sentencia while como en el segundo diagrama de flujo.

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

int main()
{
    /* Este programa permite la conversión de metros a pies y a
       centimetros tantas veces como el usuario lo desee.
       Ejemplifica la aplicacion de la sentencia do-while */

    /* Declaración de variables*/
    double distancia_en_metros, distancia_en_cm, distancia_en_ft;
    char repetir;

    /* Procesamiento */
     do
    {
       cout<< "Dame una cantidad en metros \n";
       cin>> distancia_en_metros;

       distancia_en_cm = distancia_en_metros * 100;
       distancia_en_ft = distancia_en_metros * 3.048;

       cout<<"\n";
       cout<<"Su equivalente en centimetros es "
            <<distancia_en_cm<<" \n";
       cout<<"Su equivalente en pies es "
            <<distancia_en_ft<<" \n";

       cout<<"\n";
       cout<<"Deseas realizar otro calculo similar? \n";
       cout<<"Escribe la letra s si lo deseas, si no, \n";
       cout<<"escribe cualquier otra letra \n";
       cin>>repetir;
       cout<<"\n";
    } while ( (repetir == 's') || (repetir=='S') );

    system("PAUSE");
    return 0;
}
```

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

int main()
{
    /* Este programa permite la conversión de metros a pies y a
       centimetros tantas veces como el usuario lo desee.
       Ejemplifica la aplicacion de la sentencia do-while */

    /* Declaración de variables*/
    double distancia_en_metros, distancia_en_cm, distancia_en_ft;
    char repetir;

    /* Procesamiento */
    repetir = 's';

    while ( (repetir == 's') || (repetir=='S') )
    {
      cout<< "Dame una cantidad en metros \n";
      cin>> distancia_en_metros;

      distancia_en_cm = distancia_en_metros * 100;
      distancia_en_ft = distancia_en_metros * 3.048;

      cout<<"\n";
      cout<<"Su equivalente en centimetros es "
          <<distancia_en_cm<<" \n";
      cout<<"Su equivalente en pies es "
          <<distancia_en_ft<<" \n";

      cout<<"\n";
      cout<<"Deseas realizar otro calculo similar? \n";
      cout<<"Escribe la letra s si lo deseas, si no, \n";
      cout<<"escribe cualquier otra letra \n";
      cin>>repetir;
      cout<<"\n";
    }

    system("PAUSE");
    return 0;
}
```

ESTRUCTURAS REPETITIVAS: for

Anteriormente se discutió a la sentencias while y do-while que se utilizan en C++ para programar la ejecución de ciclos; existe sin embargo otra forma de representar sentencias repetitivas. Esta otra forma es a través de la sentencia ***for*** que se detalla en este documento.

Sentencia *for* : Ejemplo de Uso

Un programador podría representar cualquier cálculo iterativo a través de las sentencias *while* y *do-while*. Sin embargo, existe un tipo de ciclo tan común que se ha creado una sentencia especial para representarlo; este ciclo se utiliza generalmente en cálculos numéricos e involucra operaciones con números que se incrementan (o disminuyen) en la misma forma en cada iteración del ciclo. Para estos ciclos se creó la sentencia ***for***.

Veamos el siguiente ejemplo de uso de la sentencia while:

```
suma=0;
n = 1;
while (n<=10)
{
  suma = suma + n;
   n++;
}
```

Observe que esta sentencia se ejecuta de forma que la sentencia de asignación del ciclo se ejecuta 10 veces, y por lo tanto el ciclo da como resultado la suma de los número ***del*** **1** ***hasta*** el **10**. Observe también que en cada iteración la variable n aumenta su valor en ***1***.

Este mismo cálculo puede realizarse con el ciclo ***for*** de la forma siguiente:

```
suma=0;
for (n=1; n<=10; n++)
   suma = suma + n;
```

La sentencia

for (n=1; n<=10; n++)

debería interpretarse como:

"*Desde n igual a 1, mientras que n<=10, aumentando n de uno en uno, ejecuta la(s) sentencia(s) siguiente(s) ...*"
En este caso se dice que **n** es la ***variable de control*** del ciclo. Analicemos la sentencia anterior. Observe que una sentencia for consiste de la palabra reservada for seguida por un conjunto de *tres cosas* encerradas entre paréntesis y separadas por punto y coma.

for(inicializacion_de_variable**;** expresion_boleana**;** actualizacion_de_variable**)**

La primera de las cosas entre paréntesis inicializa la variable de control del ciclo, la segunda proporciona una expresión boleana para poder verificar cuando se termina el ciclo y la última nos dice como la variable de control del ciclo se va a actualizar en cada iteración. Al igual que en el caso de la sentencia if-else la sentencia for no va seguida por punto y coma.
Luego de la sentencia for viene la sentencia que debe ejecutarse en el ciclo.

Esto se puede generalizar a través de la siguiente **sintaxis**:

```
for(inicializacion_de_variable; expresion_boleana; actualizacion_de_variable)
        sentencia_a_ser_ejecutada_en_el_ciclo;
```

Lo anterior corresponde al caso en el que sólo se ejecuta una sentencia en cada iteración del ciclo.

Si se compara el ciclo while con el ciclo for, se vería que el ciclo for es equivalente a un ciclo while de la forma:

```
inicializacion_de_variable;
while(expresion_boolena)
{
  sentencia_a_ser_ejecutada_en_el_ciclo;
  actualizacion_de_variable;
}
```

Así, por ejemplo, si un ciclo while se define como:

```
numero=10;
while(numero>=0)
{
  cout<<numero;
  numero--;
}
```

El ciclo se puede representar en forma equivalente con la sentencia for:

```
for(numero=10; numero>=0; numero--)
      cout<<numero;
```

En general, cuando se ejecutan más de una sentencia en el ciclo, la sintaxis de una sentencia *for* es la siguiente.

```
for(inicializacion_de_variable; expresion_boleana; actualizacion_de_variable)
{
   sentencia_1_del_ciclo;
   sentencia_2_del_ciclo;
                :
   ultima_sentencia_del_ciclo;
}
```

La diferencia con la sintaxis escrita con anterioridad es el uso de llaves.

EJERCICIOS

1. Re-escribe los siguientes ciclos utilizando la sentencia for en lugar de utilizar la sentencia while:

a)
```
int i=1;
while(i <= 10)
{
  cout<<"X";
  i = i + 3;
}
```

b)
```
int i=1;
while(i <= 10)
{
  if( (i<5) && (i != 2) )
    cout<<"X";
  i ++;
}
```

c)
```
int m=100;
do
{
    cout<<"X";
   m = m + 100;
} while(m < 1000);
```

2. ¿Qué se muestra en pantalla si se ejecutan las siguientes sentencias?

```
int n = 1024, i;
int log = 0;
for( i = 1;   i < n; i = i * 2)
   log++;
cout<<n<<" "<<log<<"\n";
```

3. ¿Qué se muestra en pantalla si se ejecutan las siguientes sentencias?

```
int n;
for( n = 10;  n>0 ; n = n  - 2)
{
   cout<<" Hello ";
   cout<<n<<"\n";
}
```

4. ¿Qué se muestra en pantalla si se ejecutan las siguientes sentencias?

```
int count;
for( count = 1;  count<5 ; count++)
    cout<<(2*count)<<" ";
```

NOTAS ADICIONALES EN LA EJECUCIÓN DE CICLOS

Secuencia de Ejecución de la Sentencia *for*

Para complementar el estudio de la sentencia *for*, se mostrará el procedimiento de ejecución de las siguientes sentencias:

```
int p=1;
int indice;
for(indice=100; indice<=1000; indice = indice*p)
    p = p + indice/100;
```

El orden de ejecución de las sentencias sería el siguiente:

1) La variable *p* es declarada e inicializada con el valor de 1

   p=1

2) La variable indice es declarada

3) **Ejecución del ciclo**, la variable indice es la variable de control del ciclo:
   a) La variable indice se **inicializa** con el valor de 100

      indice = 100

   b) Se analiza la expresión booleana *¿indice<=1000*? Como esto es verdadero, **las sentencias del ciclo se ejecutan**.

      p = p + indice/100 = 1 + 100/100 = 2

   c) Luego de la ejecución de las sentencias, la variable de control del ciclo, *indice*, se **actualiza**:

      *indice = indice*p* = 100 * 2 = 200

   d) Se analiza nuevamente la expresión booleana *¿indice<=1000*? Como es **verdadera**, las sentencias del ciclo se ejecutan.

      p = p + indice/100 = 2 + 200/100 = 4

   e) Luego de la ejecución de las sentencias, la variable de control del ciclo se actualiza:

      *indice = indice*p* = 200 * 4 = 800

f) Se analiza la expresión booleana *¿indice<=1000*? y otra vez las sentencias del ciclo se ejecutan.
p = p + indice/100 = 4 + 800/100 = 12

g) La variable de control del ciclo se actualiza:
*indice = indice*p* = 800 * 12 = 9600

h) Se analiza nuevamente la expresión booleana *¿indice<=1000*? Esta vez, sin embargo, la expresión booleana es **falsa** y con ello **termina la ejecución del ciclo**.

Se puede observar que, en resumen, cuando se ejecuta una sentencia for se realizan los siguientes pasos:

1) Se inicializa la variable de control
2) Se analiza la expresión booleana y, si es verdadera, se ejecutan las sentencias del ciclo
3) Se actualiza la variable de control
4) Se repiten los pasos 2 y 3 hasta que la expresión booleana sea falsa.

Sentencia *break*

En algunas ocasiones es necesario terminar un ciclo antes de que éste termine de forma natural. Por ejemplo, suponga que un ciclo debe solicitar 10 números positivos y calcular la suma de cada uno de ellos. Si en alguno de los casos el usuario se equivoca y proporciona un número negativo, una de las opciones sería simplemente enviar un mensaje de error y terminar el ciclo, como en el siguiente ejemplo:

```
int suma, n, numero;
suma = 0;
for(n=1;n<=10;n++)
{
   cout<<"Dame un numero positivo  \n";
   cin>>numero;
   if (numero<=0)
   {
     cout<<"Error. El numero debe ser positivo \n";
     break;
    }
   suma = suma + numero;
}
```

Observe que, en forma natural, el ciclo pediría 10 valores. Sin embargo, si el usuario proporcionara un número negativo en la iteración 5, por ejemplo, el ciclo terminaría ahí.

Es importante mencionar que la sentencia *break* puede utilizarse para terminar cualquier sentencia iterativa (ciclo), no únicamente para terminar la sentencia for. En otras palabras, *break* también puede utilizarse dentro de una sentencia while o do-while. El uso de la sentencia *break*, sin embargo, no implica necesariamente que el programa termina, implica simplemente que la ejecución del ciclo terminaría y se continuaría ejecutando las sentencias del programa colocadas después del ciclo. Observe también que la sintaxis de la sentencia es simplemente la palabra clave break seguida de punto y coma.

break;

Sentencias *for* Anidadas

El cuerpo de un ciclo (lo que se encierra entre llaves) puede contener cualquier tipo de sentencia, de forma que es posible escribir un ciclo iterativo dentro de otro. Cuando esto ocurre, se dice que se tienen **sentencias anidadas**. El mismo término se utiliza cuando dentro de alguno de los casos de una sentencia condicional (if) se encuentra otra sentencia condicional. El siguiente es un ejemplo de ciclos anidados:

```
int n,m;
for(n=1; n<=3;n++)
   for(m=3;m>=1;m--)
      cout<< n << " x "<<m <<" = " <<n*m <<"\n";
```

Como **ejercicio**, determine que se muestra en pantalla con la ejecución de dichas sentencias.

Haga lo mismo para las siguientes sentencias:

a)

```
double muestra1, double muestra2, suma=0.0;
for(muestra1=2.0; muestra1>0;muestra1 = muestra1 – 0.5)
{
   for(muestra2=1.0;muestra2<8.0;muestra2 = muestra2 * 2.0)
  {
      suma = suma + muestra1 + muestra2;
      cout<< suma <<"\n";
   }
}
```

b)

```
int n=5;
while(n>0)
{
   if(n==2)
     break;
   cout<< n << "\n";
   n--;
}
```

SENTENCIAS CONDICIONALES MÚLTIPLES: switch

Como ya se analizó anteriormente, una sentencia if-else posee sólo dos alternativas. Dicha sentencia permite a un programa seleccionar entre dos acciones posibles (casos falso y verdadero). Existe muchas veces, sin embargo, la necesidad de incluir en un programa sentencias que permitan la selección de varias (más de dos) alternativas. Para ello se pueden utilizar sentencias if-else *anidadas* o múltiples sentencias if-else en secuencia, como ya se ha hecho con anterioridad. Una alternativa a esto es el uso de la sentencia ***switch***, sentencia de C++ que se diseñó especialmente para representar una selección condicional múltiple.

Sentencia *switch*

La forma más simple de estudiar la sentencia switch es comenzar con un ejemplo que muestre su estructura básica.

```
#include <iostream.h>
#include<stdlib.h>
int main()
{
  int dia;
  cout<<"Dame un numero entero entre 1 y 7 \n";
  cin>>dia;
  cout<<"\nEl dia correspondiente es ";
  switch(dia)
   {
       case 1:
           cout<<"Lunes";
           break;
       case 2:
           cout<<"Martes";
           break;
       case 3:
           cout<<"Miercoles";
          break;
       case 4:
           cout<<"Jueves";
           break;
       case 5:
           cout<<"Viernes";
           break;
       case 6:
           cout<<"Sabado";
           break;
       case 7:
           cout<<"Domingo";
           break;
```

```
        default:
            cout<<"Error en el numero";
    }
     cout<<"\n";
    return 0;
    system("PAUSE");
}
```

Este ejemplo se discutió con anterioridad. Lo que el programa realiza es pedir un número entero entre 1 y 7 y, dependiendo de ese valor, mostrará en pantalla el nombre de un día de la semana.

Lo primero que se escribe en dicha sentencia es el identificador ***switch***. Cuando se ejecuta una sentencia *switch*, uno de varias alternativas se ejecuta. Para determinar cual de los casos se debe ejecutar se utiliza lo que se conoce como la ***expresión de control*** de la sentencia, que se escribe **entre paréntesis**. En el ejemplo la expresión de control es la variable *dia*. La expresión de control escrita entre paréntesis deberá siempre ser un valor booleano (FALSE o TRUE), un valor entero o un caracter. Durante la ejecución de la sentencia *switch*, se analiza la expresión de control para obtener su valor.

Cada una de las alternativas de la selección múltiple se representa por el identificador **case** seguido de un **valor constante** (entero, caracter o booleano) y **dos puntos**. Lo que se hace a continuación es que se compara el valor de la expresión de control con los valores constantes de los casos de la sentencia. Cuando encuentra el caso cuya constante sea igual al valor de la expresión de control, se ejecutan las sentencias correspondientes a dicho caso (hasta encontrar una sentencia break). Si por alguna razón no se encuentra ninguna constante igual a la expresión de control, entonces se ejecutan las sentencias que corresponden al caso **default**, que generalmente se escribe al final de la sentencia switch (si éste caso no existiera, entonces no se ejecutaría ninguna de las alternativas).

Observe que los casos de una sentencia switch se escriben **entre llaves**. Note que después de las sentencias de cada caso se tiene una sentencia **break**. Si las sentencias break no se incluyeran, entonces al ejecutarse uno de los casos, la computadora continuaría ejecutando las sentencias de todos los casos

colocados por debajo del caso que si debe ejecutarse. La sentencia break automáticamente hace que termine la ejecución de la sentencia switch.
De acuerdo a la anterior, la sintaxis de la sentencia switch es la siguiente:

```
switch(expresion_de_control)
{
   case constante_1:
         sentencias_del_caso_1:
          break;
   case constante_2:
         sentencias_del_caso_2:
          break;
            ⋮
   case constante_n:
         sentencias_del_caso_n:
          break;
   default:
          sentencias_del_caso_default:
}
```

Este es otro ejemplo de aplicación de la sentencia switch:

```
#include <iostream.h>
#include<stdlib.h>
int main()
{
  char calificacion;
  cout<<"Dame la calificacion en escala de caracteres \n";
  cin>>calificacion;
  cout<<"\nLa calificacion numerica equivalente es ";
  switch(calificacion)
   {
       case 'A':
           cout<<"100";
           break;
       case 'B':
           cout<<"85";
           break;
```

```
        case 'C':
            cout<<"70";
           break;

        case 'D':
        case 'F':
            cout<<"Reprobatoria. Ve a estudiar";
            break;
       default:
            cout<<"Error en la calificacion proporcionada";
   }
    cout<<"\n";
   return 0;
   system("PAUSE");
}
```

Sentencia *switch* para escribir Menús

Una de las aplicaciones más útiles de la sentencia switch es el uso de un Menú. En esta aplicación, dependiendo de la selección del usuario, es posible escoger entre diversas funciones a ser ejecutadas, como se mostrará posteriormente a través de un ejercicio.

APLICACIÓN DE LA SENTENCIA SWITCH

EJERCICIO 1

Codifique el siguiente programa en C++. Se trata de un ejemplo de la aplicación de la sentencia *switch* en menús.

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

    /* Este es un programa que ejemplifica el uso de la sentencia
       swtich en la creación de menus. Permite el calculo de areas de
       varias formas geometricas*/

int main()
{
    /* Declaracion*/
    int opcion;
    double area_cua, area_tria, area_cir, area_cil;
    double PI = 3.1415926;
    double lado, radio, base, altura;

    /* Seleccion de opciones */
    cout<<"Selecciona el calculo del area que deseas ejecutar \n";
    cout<<"\n";
    cout<<"1  Area de un cuadrado \n";
    cout<<"2  Area de un circulo \n";
    cout<<"3  Area de un triangulo \n";
    cout<<"4  Area exterior de un cilindro \n";
    cout<<"\n";
    cout<< "Escribe el numero de opcion y presiona Enter \n";
    cin>>opcion;
    cout<<"\n";
    switch(opcion)
    {
      case 1:
              cout<<"Dame el lado del cuadrado \n";
              cin>>lado;
              cout<<"\n";
              area_cua = lado * lado;
```

```
            cout<<"El area del cuadrado es "<<area_cua<<"\n";
            break;
        case 2:
            cout<<"Dame el radio del circulo \n";
            cin>>radio;
            cout<<"\n";
            area_cir = PI * radio * radio;
            cout<<"El area del circulo es "<<area_cir<<"\n";
            break;
        case 3:
            cout<<"Dame la base y la altura del triangulo \n";
            cin>>base>>altura;
            cout<<"\n";
            area_tria = (base * altura) / 2.0;
            cout<<"El area del triangulo es "<<area_tria<<"\n";
            break;
        case 4:
            cout<<"Dame el radio y la altura del cilindro \n";
            cin>>radio>>altura;
            cout<<"\n";
            area_cil = 2.0* PI*radio*radio + PI * 2.0 * radio * altura;
            cout<<"El area del cilindro es "<<area_cil<<"\n";
            break;
        default:
            cout<<"Error en la opcion seleccionada\n";
    }
    cout<<"\n";
    system("PAUSE");
    return 0;
}
```

EJERCICIO 2

¿Como proporcionaría al usuario (la persona que ejecuta el programa) la opción de repetir el cálculo?

EJERCICIOS: switch y for

1. Escriba un programa en C++ que tome como dato el numero de letras de un nombre propio y que, con dicho dato, solicite cada una de las letras del nombre y cuente el numero de vocales y el numero de consonantes de dicho nombre.

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

/* Este programa toma como datos el numero de letras
   y luego cada una de las letras de un nombre propio. Como resultado
   proporciona el numero de vocales y el numero de consonantes
   del nombre */

int main()
{
    /* Declaracion */
    int no_letras, no_vocales, no_consonantes;
    char letra;  /* caracter para pedir cada letra */
    int n; /* Variable de control */

    /* Entrada de Datos */
    cout<<"Cuantas letras tiene tu nombre propio? \n";
    cin>>no_letras;
    cout<<"\n";

    no_vocales = 0;
    no_consonantes = 0;

    /* Solicitud de CADA UNA DE LAS LETRAS e identificacion
       de si se trata de una vocal o una consonante */
    for(n=1; n<=no_letras; n++)
    {
       if (n==1)   /* Pidiendo letra */
         cout<<"Dame la primer letra \n";
       else
         cout<<"Dame la siguiente letra \n";
       cin>>letra;

       /* Identificacion de letra */
       switch(letra)
       {
         case 'A':
         case 'E':
         case 'I':
         case 'O':
         case 'U':
```

```
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                  no_vocales++;
                  break;
            default:
                  no_consonantes++;
          }
      }
      cout<<"\n";
      cout<<"Tu nombre tiene "<<no_vocales<<" vocales y "<<no_consonantes
      <<" consonantes \n";
      cout<<"\n";
      system("PAUSE");
      return 0;
}
```

2. Escriba un programa en C++ que calcule el costo de tres llamadas telefónicas de larga distancia que se realizan en un día cualquiera. Suponga que el costo de la llamada depende de la hora en que se realiza, de la siguiente forma:

   a) De las 8 hasta las 18 3 pesos el minuto
   b) Después de las 18 hasta las 22 2 pesos el minuto
   c) Después de las 22 hasta antes de las 8 1 peso el minuto

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

int main()
{
    int hora, minutos,n;
    double hora_fraccion,minutos_llamada, costo;

    costo =0.0;
    for(n=1; n<=3;n++)
    {
      cout<<"A que hora se hizo la llamada "<<n<<" \n";
      cout<<"Dame la hora y los minutos. Presiona enter despues"
          <<" de cada dato\n";
      cin>>hora>>minutos;
      cout<<"\n";
      cout<<"Cuantos minutos duro la llamada "<<n<<" \n";
      cin>>minutos_llamada;
      cout<<"\n";
      hora_fraccion = hora + minutos/60.0;
      if((hora_fraccion>=8.0) &&(hora_fraccion<=18.0))
        costo = costo + minutos_llamada * 3;
      else if((hora_fraccion>18.0) &&(hora_fraccion<=22.0))
        costo = costo + minutos_llamada * 2;
      else if(((hora_fraccion>=22.0) &&(hora_fraccion<=24.0))||
        ((hora_fraccion>=0.0) &&(hora_fraccion<8.0)))
        costo = costo + minutos_llamada * 1;
    }
    cout<<"El costo de las 3 llamadas fue "<<costo<<" pesos\n";
    cout<<"\n";
    system("PAUSE");
    return 0;
}
```

UNIDAD III

TEMA I

ARREGLOS

ARREGLOS

Un arreglo se utiliza para procesar colecciones de datos de un mismo tipo, como una lista de temperaturas, de calificaciones, de nombres, etc. En este documento se establecen las bases del manejo de arreglos.

Introducción

¿Son útiles los arreglos en programación? Comencemos el estudio de arreglos en C++ tratando de responder a esta pregunta. Para ello, recordemos el ejercicio en el que se identificaban cuantas vocales y cuantas consonantes tiene un nombre propio dado. En dicho ejercicio fue necesario escribir un ciclo (sentencia ***for***) en el que se pedía cada letra del nombre y luego se analizaba dicha letra para saber si era o no una vocal (sentencia ***switch***):

```
for(n=1; n<=no_letras; n++)
    {
       cout<<"Dame la letra "<<n<<" del nombre \n";
       cin>>letra;
       /* Identificacion de letra */
       switch(letra)
       {
         case 'A':
         case 'E':
         case 'I':
         case 'O':
         case 'U':
         case 'a':
         case 'e':
         case 'i':
         case 'o':
         case 'u':
                no_vocales++;
                break;
         default:
                no_consonantes++;
       }
    }
```

Uno puede observar que, en dicho ciclo, cada una de las letras del nombre se asigna a la variable ***letra*** en cada iteración. Cada nuevo valor de letra sobre-escribe al valor anterior. Por ello, cuando termina de ejecutarse el ciclo, la

variable tiene como valor únicamente a la última de las letras del nombre. ¿Que pasaría, sin embargo, si alguien quisiera conservar a todas las letras del nombre hasta el final del programa?. Para poner un ejemplo, suponga que un nombre propio tiene 7 letras. Piense bien en dicha situación y se dará cuenta que (***sin el uso de arreglos***) el querer mantener a cada una de las siete letras complica mucho el análisis. Por principio de cuentas, sería necesario tener siete variables de tipo carácter (letra_1, letra_2, letra_3,..., letra_7), una para cada letra. Además, hubiera también sido necesario escribir la sentencia switch 7 veces, una vez para cada una de dichas letras. Piense en el tamaño del programa que sería necesario.

Aparte de dicho ejercicio, en programación es muy común encontrarse con problemas en los cuales es necesario tener muchas variables de un mismo tipo. Específicamente en ingeniería química, un ejemplo que se verá durante la carrera es una columna de destilación, en la cual una de las variables importantes es la temperatura. Si la columna tuviera 50 secciones (llamados platos) necesitaríamos 50 variables para guardar la temperatura de cada uno de ellos. Imagine tener que declarar las variables:

double temperatura_1, temperatura_2, ... , temperatura_50;

Tener que hacer algo como eso sería absurdo. Afortunadamente, el uso de arreglos (en cualquier lenguaje de programación) hace muy simples este tipo de tareas.

DECLARACIÓN DE ARREGLOS

Un arreglo es prácticamente una lista de variables. Cada una de las variables de la lista tiene un nombre que está constituido por dos partes. Una de las partes es el nombre del arreglo, nombre que van a compartir cada una de las variables de la lista. La otra parte es diferente para cada variable y es lo que permite identificar a una variable de la lista de otra.

Como ejemplo de declaración de un arreglo, analicemos la siguiente sentencia:

double temperatura[5];

Ese es un ejemplo típico de la declaración de un arreglo. En este ejemplo, se está declarando que existe un arreglo de 5 variables de tipo double. Al tipo double se le conoce como el ***tipo*** del arreglo. El ***nombre del arreglo*** es temperatura. El número entre corchetes indica cuantos elementos tiene el arreglo (en este caso 5) y se le conoce como el ***tamaño del arreglo***. La sentencia anterior sería lo mismo que declarar cada una de las variables del arreglo por separado:

```
double temperatura[0], temperatura[1], temperatura[2], temperatura[3],
        temperatura[4];
```

Las variables como temperatura[0] y temperatura[1] que se declaran al declarar un arreglo se denominan ***variables indexadas*** o ***elementos del arreglo***. Al número entre corchetes se le denomina índice. ***Es muy importante que observe que los índices comienzan a numerarse desde el cero, no desde el uno***. De forma que el número de variables se identifican por un índice que va desde el **cero** hasta el número del ***tamaño del arreglo menos uno***. Es preciso que no exista confusión. En la declaración:

```
double temperatura[5];
```

el número 5 indica que el arreglo tiene 5 variables indexadas. Por otra parte, temperatura[0] o temperatura [1] son elementos del arreglo identificados por un índice que irá desde cero hasta cuatro.

Cualquier variable indexada puede ser utilizada en cualquier parte del programa como si se tratara de cualquier variable. Es decir, es válido escribir sentencias como las siguientes:

```
Temperatura[0] = Temperatura[0] + 273;  /* asignacion*/
cin>> Temperatura[2];  /*entrada de datos*/
cout<<"El valor de la Temperatura es "<<Temperatura[3]<<"\n"; /* salida*/
```

Otro punto importante es que el número escrito entre corchetes en la **declaración** de un arreglo tiene que ser un número constante. Sin embargo, el índice escrito entre corchetes no necesariamente tiene que ser un entero

constante cuando se utilizan variables indexadas en las sentencias de un programa, sino que puede ser una expresión. Esto es válido siempre y cuando la expresión de como resultado un número que se encuentre entre cero y el tamaño del arreglo menos uno.

Por ejemplo, en el caso que estamos analizando de la temperatura la expresión siguiente:

Temperatura[i] = Temperatura[i] + 273;

sería válida siempre y cuando la variable *i* tenga un valor entre 0 y 4.

El uso de arreglos es de especial importancia en ciclos iterativos. En particular, cuando se utiliza la sentencia ***for*** es relativamente fácil manipular arreglos utilizando la variable de control del ciclo. Las siguientes sentencias servirían para guardar las 7 letras de un nombre en el ejemplo con el cual iniciamos nuestra explicación del uso de arreglos:

```
char letra[7];
for(n=1; n<=7; n++)
{
        cout<<"Dame la letra "<<n<<" del nombre \n";
        cin>>letra[n-1];
]
```

Sumarizando, la sintaxis en la declaración de arreglos es:

Nombre_del_tipo Nombre_del_arreglo**[**tamanio_del_arreglo**];**

Ejemplos:

```
int arreglo_grande[100];
double a[3];
double b[5], x, y, z;
```

Como se observa en el último ejemplo, es posible declarar arreglos conjuntamente con otras variables del mismo tipo.

Errores Comunes

El error más común en el manejo de arreglos es el tratar de usar un elemento del arreglo que no existe. Este error sólo es detectado por algunos de los compiladores de C++ y se muestra con un mensaje similar a:

array index out of range

Por ejemplo, suponga que se declara el arreglo:

double a[3];

con ello se está declarando un arreglo de 3 elementos: a[0], a[1] y a[2]. Si en el programa se tuviera algo como:

```
double a[3], x=3.5;
int n=2;
n++;
a[n] = 2*3* x;
```

En este caso se presentaría un error dado n=3 y dado que la variable a[3] no existe.

INICIALIZACIÓN DE ARREGLOS

Al igual que otras variables, los arreglos se pueden inicializar al momento de declararse. Para hacer eso, es necesario enlistar los valores de cada uno de los elementos del arreglo **entre llaves** y **separados por comas**. Ejemplo:

```
int b[3] = {2, 12, 1};
```

En casos como éste, cuando se escriben cada uno de los valores de los elementos del arreglo, algunos compiladores permiten omitir el tamaño del arreglo en la declaración. Por ello, la declaración:

```
int b[] = {2, 12, 1};
```

sería equivalente a la anterior.

EJERCICIOS Y OPERACIONES CON ARREGLOS UNIDIMENSIONALES

Qué se muestra en pantalla con la ejecución de las siguientes sentencias?

```
int i, temp[10];
for(i=0; i<10; i++)
    temp[i] = 2*i;
for(i=0; i<10; i++)
    cout<<temp[i] <<" \n";
```

Cuál es el error de las siguientes sentencias?

```
int ejemplo[10], indice;
for(indice=1; indice<=10; indice++)
     ejemplo[indice] = 3 * indice;
```

En la siguiente declaración de un arreglo:

double calificación[5];

Cuál es el nombre del arreglo?

Cuál es el tipo del arreglo?

Cuál es el tamaño del arreglo (Cuántos elementos tiene)?

Cuál es el rango de valores que el indice i puede tener si se usa en el programa

calificación[i]

Qué se muestra en pantalla cuando se ejecutan las siguientes sentencias?

```
double a[3] = {1.1, 2.2, 3.3};
cout<< a[0] <<" "<<a[1] << "  "<< a[2] << "\n";
a[1] = a[2];
cout<< a[0] <<" "<<a[1] << "  "<< a[2] << "\n";
```

Qué se muestra en pantalla cuando se ejecutan las siguientes sentencias?

```
char simbolo[3] = {'a', 'b', 'c'};
int indice;
for(indice = 0; indice <3; indice++)
      cout<<simbolo[indice];
```

Cuál es el error en las dos declaraciones de arreglos siguientes?

```
const int tamanio = 4;
int x[4]={8,7,6,4,3};
int y[tamanio -4];
```

APLICACIONES: SUMAS Y PRODUCTOS USANDO ARREGLOS Y SENTENCIAS REPETITIVAS

Algunas de las actividades más comunes que se realizan utilizando ciclos con la sentencia for son sumatorias y productos repetitivos. Cuando se realizan estas operaciones generalmente se tiene conocimiento de cuantos elementos tiene la sumatoria o el producto repetitivo.

Como ejemplo consideremos que se tiene la expresión:

$$F_T = \sum_{n=1}^{3} f_n$$

¿ Como evaluar dicha sumatoria utilizando un ciclo for ?

Las siguientes sentencias nos proporcionarían una forma de representar dicha sumatoria en C++ :

```
double FT, f[3];
int n;
for(n=0; n<3; n++)
   cin>>f[n];
/* Lo siguiente es la sumatoria */
```

```
FT = 0.0;
for(n=0; n<3; n++)
   FT = FT + f[n];
```

Los siguientes dos aspectos deberán cuidarse siempre que se desea realizar una sumatoria o un producto repetitivo:

1) Se declarará una variable a la cual se le asignará el valor de la sumatoria o el producto. Dicha variable debe **inicializarse**. Cuando se trata de una sumatoria la variable generalmente se inicializa con el valor de cero. Cuando se trata de un producto generalmente se inicializa con el valor de uno.
2) La sumatoria o el producto se logra a partir de una asignación dentro de un ciclo. Note que, dentro del ciclo, la variable a contener la sumatoria o el producto aparece en ambos lados de la asignación.

Observe que las mismas reglas aplican para el siguiente ejemplo de un producto repetitivo:

$$V_M = \prod_{n=1}^{3} V_n$$

Este cálculo se puede realizar a través de las siguientes sentencias en C++ :

```
double VM, V[3];
int n;
for(n=0; n<3; n++)
   cin>>V[n];
/* Lo siguiente es la sumatoria */
VM = 1.0;
for(n=0; n<3; n++)
   VM = VM * V[n];
```

EJERCICIOS

Escriba las sentencias en C++ (no es necesario que escriba todo el programa, como en los ejemplos) que representen las siguientes sumatorias y productos repetitivos.

1) $A = 1 + \sum_{n=1}^{5} (x - 3)$

2) Escriba las sentencias que sumen todos los números pares entre 100 y 200.

3) $P = \prod_{k=1}^{10} \frac{x_k}{k}$

4) Escriba las sentencias que obtengan el producto de todos los números entre 37 y 55.

ARREGLOS MULTIDIMENSIONALES

En ocasiones es útil tener arreglos de más de un índice. Esto se puede hacer en C++ y en la mayoría de los lenguajes de programación. La siguiente sentencia declara un arreglo multidimensional de variables de punto flotante que llevan el nombre genérico de Temperatura:

double Temperatura[3][20];

Los índices de este arreglo son:

Temperatura[0][0]	Temperatura[1][0]	Temperatura[2][0]
Temperatura[0][1]	Temperatura[1][1]	Temperatura[2][1]
⋮	⋮	⋮
Temperatura[0][19]	Temperatura[1][19]	Temperatura[2][19]

Observe que con la declaración anterior se declaran simultáneamente 60 variables indexadas. Note también que los valores que indican el tamaño del arreglo multidimensional deben encerrarse entre corchetes (para cada conjunto de valores).

De hecho, los arreglos multidimensionales pueden contener cualquier número de índices, pero rara vez son necesarios más de dos para la mayoría de las aplicaciones en ingeniería química. La sintaxis formal para declarar un arreglo multidimensional es:

nombre_tipo nombre_arreglo [tamanio_1] [tamanio_2] ... [ultimo_tamanio];

EJERCICIOS: OPERACIONES CON ARREGLOS MULTIDIMENSIONALES

¿Qué se muestra en pantalla cuando se ejecutan las sentencias de cada uno de los casos siguientes:

1)

```
int n,m;
int producto[3][3];
for(n=1; n<=3;n++)
   for(m=1;m<=3;m++)
      producto[n-1][m-1] = n*m;

for(n=2; n>=0;n--)
   for(m=2;m>=0;m--)
      cout<<producto[n][m];
```

2)

```
int n,m;
double promedio[3]={0.0,0.0,0.0};
for(n=1; n<=3;n++)
{

   for(m=1;m<=3;m++)
   {
      promedio[n-1] = promedio[n-1] +n*m;
   }
  promedio[n-1] = promedio[n-1]/3.0;
}
for(m=2;m>=0;m--)
      cout<<promedio[m];
```

INICIALIZACIÓN DE ARREGLOS MULTIDIMENSIONALES

La inicialización de arreglos multidimensionales es muy similar a la que se describió para arreglos de un solo índice. Otra vez es necesario enlistar los valores de cada uno de los elementos del arreglo **entre llaves** y **separados por comas**. La diferencia sin embargo, es que es necesario tener más de un grupo de valores encerrados entre llaves. Por ejemplo:

```
int x[2][3]={{1,2,3},{4,5,6}};
```

Observe que tenemos un conjunto de llaves que encierra a otros grupos de números también encerrados entre llaves y separados por comas. ¿Cuántos grupos de números se tienen encerrados por las llaves de los extremos?
2, porque la dimensión del primer índice es **2**. ¿Cuántos números se tienen en cada uno de los grupos? 3, por el tamaño correspondiente al segundo índice es **3**.

El ejemplo equivaldría a tener el siguiente arreglo de números. El primer índice representaría al renglón y el segundo índice a la columna:

1	2	3
x[0][0]	x[0][1]	x[0][2]
4	5	6
x[1][0]	x[1][1]	x[1][2]

APLICACIONES EN MATRICES Y SISTEMAS COMPLEJOS

Suponga que se tiene el siguiente arreglo de números:

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

Elabore un programa en C++ que calcule cuantos “ceros” aparecen en cada renglón del arreglo.

```
#include <iostream.h>
#include <stdlib.h>
/* Ejemplo de manipulacion de indices en un arreglo */
int main()
{    /* Declaracion e inicializacion del arreglo */
     int x[5][5]= { {0, 2,    5, 7, 6},
                    {0, 0, 0, 3, 8},
                    {2, 9, 6, 3, 4},
                    {1, 5, 6, 1, 4},
                    {0, 9, 2, 5, 0} };
     int n,m, numero;

     /* Conteo y Salida de Resultados*/
     for(n=0; n<5; n++)
     {
       cout<<"Renglon "<<n+1<<":    ";
       numero = 0;
       for(m=0; m<5;m++)
       {
        if(x[n][m] == 0)
           numero++;
       }
       cout<<numero<<" ceros \n";
     }
     cout<<"\n";
     system("PAUSE");
     return 0;
}
```

EJERCICIO

Uso de arreglos: El método de Gauss-Jordan. Resolver el sistema de Ecuaciones siguiente:

$$\begin{aligned} 6x_1 - 2x_2 + 2x_3 + 4x_4 &= 12 \\ 12x_1 - 8x_2 + 6x_3 + 10x_4 &= 34 \\ 3x_1 - 13x_2 + 9x_3 + 3x_4 &= 27 \\ -6x_1 + 4x_2 + x_3 - 18x_4 &= -38 \end{aligned}$$

Representación Matricial

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 12 \\ 34 \\ 27 \\ -38 \end{bmatrix}$$

Método de Gauss-Jordan (Obtención de una Matriz Diagonal)

$$\left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 12 \\ 12 & -8 & 6 & 10 & 34 \\ 3 & -13 & 9 & 3 & 27 \\ -6 & 4 & 1 & -18 & -38 \end{array}\right]$$

Paso 1:

Al segundo renglón restarle el primer renglón multiplicado por 12/6

Al tercer renglón restarle el primer renglón multiplicado por 3/6

Al cuarto renglón restarle el primer renglón multiplicado por -6/6

$$\left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 12 \\ 0 & -4 & 2 & 2 & 10 \\ 0 & -12 & 8 & 1 & 21 \\ 0 & 2 & 3 & -14 & -26 \end{array}\right]$$

Paso 2:

Al primer renglón restarle el segundo renglón multiplicado por -2/-4

Al tercer renglón restarle el segundo renglón multiplicado por -12/-4

Al cuarto renglón restarle el segundo renglón multiplicado por 2/-4

$$\left[\begin{array}{cccc|c} 6 & 0 & 1 & 3 & 7 \\ 0 & -4 & 2 & 2 & 10 \\ 0 & 0 & 2 & -5 & -9 \\ 0 & 0 & 4 & -13 & -21 \end{array}\right]$$

Paso 3:

Al primer renglón restarle el tercer renglón multiplicado por 1/2

Al segundo renglón restarle el tercer renglón multiplicado por 2/2

Al cuarto renglón restarle el tercer renglón multiplicado por 4/2

$$\left[\begin{array}{cccc|c} 6 & 0 & 0 & 11/2 & 23/2 \\ 0 & -4 & 0 & 7 & 19 \\ 0 & 0 & 2 & -5 & -9 \\ 0 & 0 & 0 & -3 & -3 \end{array}\right]$$

Paso 4:

Al primer renglón restarle el cuarto renglón multiplicado por (11/2)/-3

Al segundo renglón restarle el cuarto renglón multiplicado por 7/-3

Al tercer renglón restarle el cuarto renglón multiplicado por -5/-3

$$\left[\begin{array}{cccc|c} 6 & 0 & 0 & 0 & 6 \\ 0 & -4 & 0 & 0 & 12 \\ 0 & 0 & 2 & 0 & -4 \\ 0 & 0 & 0 & -3 & -3 \end{array}\right]$$

$x_1 = 6/6 = 1 \quad x_2 = 12/-4 = -3 \quad x_3 = -4/2 = -2 \quad x_4 = -3/-3 = 1$

```
#include <iostream.h>
#include <stdlib.h>
/* Metodo de Gauss Jordan para la Solucion de Sistemas
   de Ecuaciones Lineales*/
int main()
{    /* Declaracion e inicializacion de la Matriz */
     double A[4][4]= {     {6, -2, 2, 4},
                           {12, -8, 6, 10},
                            {3, -13, 9, 3},
                           {-6, 4, 1, -18}    };
     /* Termino del lado derecho y variables del problema */
     double x[4], b[4]={12,34,27,-38}, factor;
     int n,m,k;

     /* Diagonlizacion de la Matriz */
     for(n=1; n<=4; n++)
     {
       for(k=1;k<=4;k++)
       {
         if (n != k)
         {
            factor = (A[k-1][n-1] / A[n-1][n-1]);
            for(m=1; m<=4;m++)
                 A[k-1][m-1] = A[k-1][m-1] - (  factor * A[n-1][m-1] );
            b[k-1] = b[k-1] - (factor * b[n-1]);
         }
       }
     }
     /* Salida de la Matriz Diagonal a pantalla */
     for(n=1; n<=4; n++)
     {
       for(k=1;k<=4;k++)
       {
          cout<<A[n-1][k-1]<<"\t";
       }
       cout<<"\t"<<b[n-1];
       cout<<"\n";
     }
     cout<<"\n";
     cout<<"\n";
     /* Calculo de las incognitas */
     for(k=1;k<=4;k++)
     {
        x[k-1] = b[k-1]/A[k-1][k-1];
        cout<<"x"<<k<<" = "<<x[k-1]<<"\n";   /* Salida de Resultados */
     }
     cout<<"\n";
     cout<<"\n";
     system("PAUSE");
     return 0;
}
```

EJERCICIO

Uso de arreglos: El "cuadrado mágico".

Un cuadrado mágico es una matriz cuadrada con un número impar de renglones y columnas. En dicha matriz, los números de cada renglón, de cada columna y de cada una de las diagonales, suman el mismo valor. Por ejemplo:

6	1	8
7	5	3
2	9	4

Note que los números de todos los renglones, columnas y diagonales suman 15. Una técnica a través de la cual se puede generar un cuadrado mágico es la siguiente: Se comienza asignando un valor de 1 al elemento central de la primera fila. A continuación se escriben los valores sucesivos (2,3, etc.) desplazándose desde la posición anterior una fila hacia arriba y una columna hacia la izquierda. Estos cambios se realizan tratando a la matriz como si estuviera envuelta sobre sí misma, de forma que moverse una posición hacia arriba desde la fila superior lleva a la inferior, y moverse una posición hacia la izquierda desde la primera columna conduce a la última. Si la nueva posición ya está ocupada, en lugar de desplazarse hacia arriba y a la izquierda, se moverá sólo una posición hacia abajo. Escriba un programa que muestre un cuadro mágico de dimensión n, donde n puede estar entre 1 y 9.

```
#include <iostream.h>
#include <stdlib.h>
int main()
{
    /* Este programa muestra en pantalla un cuadrado mágico
       de dimension impar n */
    /* Declaración del máximo arreglo necesario */
    int Cuadrado[9][9], n, i, j, k, inicio, fin, prer, prec;

    /* Datos: dimension del cuadrado */
    cout<<"Dame la dimension del cuadrado magico que deseas \n";
    cout<<"Debe ser un numero impar entre 1 y 9 \n";
    cin>>n;
    cout<<"\n";
    /* Inicializacion del arreglo */
    for(i=0;i<n;i++)
       for(j=0;j<n;j++)
          Cuadrado[i][j] = 0;

    /* Llenado del cuadrado */
    inicio = n/2;
    fin = n * n;
    Cuadrado[0][inicio] = 1;
    i=0;
    j=inicio;
    for(k=2;k<=fin;k++)
    {
       prer = i;          /* Conservar posición inicial */
       prec = j;
       i--;               /* Un renglón hacia arriba*/
       if(i<0)
         i=n-1;           /* Si era el primero, ir al ultimo*/
       j--;               /* Una columna a la izquierda*/
       if(j<0)
         j=n-1;           /* Si era la primera, ir a la ultima */

       if(Cuadrado[i][j] != 0)    /* Posicion ocupada */
       {
         i= prer;             /* Recuperar posicion inicial */
         j = prec;
         i++;                 /* Un renglón hacia abajo*/
         if(i>n-1)            /* Si es el ultimo, ir al primero */
           i = 0;
         Cuadrado[i][j] = k;
       }
       else
         Cuadrado[i][j] = k;  /* Posicion no ocupada */
    }
```

```
    /* Salida a pantalla */
    cout<<"El cuadrado es: \n";
    cout<<"\n";
    for(i=0;i<n;i++)
    {
      for(j=0;j<n;j++)
      {
       cout<<Cuadrado[i][j]<<"\t";
      }
      cout<<"\n";
    }
    cout<<"\n";
    system("PAUSE");
    return 0;
}
```

UNIDAD III

TEMA II

FUNCIONES

FUNCIONES

Cuando es necesario escribir programas complicados para resolver problemas complejos, una práctica común entre los programadores es descomponer el algoritmo (el diagrama de flujo) en varias partes. Cada de una de éstas partes puede codificarse en forma independiente en la forma de subprogramas. Así, habrá un cierto número de subprogramas que se encargan de realizar sólo parte de las tareas requeridas para resolver el problema; estos subprogramas estarán relacionados de forma que su ejecución conjunta permitirá la solución del programa global.

Funciones como Subprogramas en C++

C++ permite la definición de este tipo de subprogramas. En algunos lenguajes de programación, las subpartes son llamadas procedimientos o subrutinas. En C++ las subpartes de un programa se denominan **funciones**.

Una de las ventajas de dividir los programas en subprogramas es que diferentes programadores pueden realizar diferentes tareas. Este tipo de trabajo de equipo es indispensable para la elaboración de programas complicados en un tiempo razonable.

FUNCIONES PREDEFINIDAS

El lenguaje C++, como la mayoría de los lenguajes de programación, permite el uso de *"bibliotecas"* con **funciones predefinidas** que se pueden utilizar en cualquier programa. Se discutirá primero como se utilizan estas funciones predefinidas y, posteriormente, se mostrará como un programador puede construir sus propias funciones.

Uso de Funciones Predefinidas

Se utilizará la función **sqrt** (**sq**uare **r**oo**t** = *raíz cuadrada*) para ejemplificar el uso de funciones predefinidas. La función *sqrt* toma el valor de un número, por ejemplo 9.0, y calcula el valor de su raíz cuadrada, en este caso 3.0. El valor que la función toma como punto de partida (9.0 en el ejemplo) se le conoce como su **argumento**. Al valor que calcula se le conoce como **valor de regreso** (o retorno).

Algunas funciones pueden tener más de un argumento, pero todas las funciones tienen un solo valor de retorno. Si se trata de comparar a una función con los programas que se han analizado hasta ahora, los argumentos son análogos a los datos, mientras que los valores de retorno son análogos a los resultados.

Un ejemplo del uso de una función es el siguiente:

```
raiz = sqrt(9.0);
```

A la expresión *sqrt(9.0)* se le conoce como llamado a la función (o invocación a la función). El argumento de una función puede, como en este caso, ser un valor constante, pero también puede ser una variable o una expresión más complicada. La única restricción en este sentido es que la constante, la variable o la expresión deben de proporcionar un valor que sea del tipo requerido por la función.

Las funciones pueden utilizarse como parte de cualquier expresión legal en C++. Por ejemplo, las siguientes son expresiones válidas en C++:

```
double venta, beneficio, area;
venta = 100.50;
area = 27.5;
beneficio = sqrt(venta);
cout<< "El lado del cuadrado es " << sqrt(area) <<"\n";
```

LLAMADO A FUNCIONES

Un llamado a una función consiste en el nombre de una función seguida por la lista de sus argumentos encerrados entre paréntesis. Si hay más un de argumento, los argumentos se separan mediante comas. Un llamado a una función puede ser usado como cualquier otra expresión en C++ siempre y cuando se conserve la consistencia entre los tipos de las variables del programa. La sintaxis es la siguiente.

Si hay un solo argumento:

```
nombre_de_funcion(argumento)
```

si hay más de un argumento:

nombre_de_funcion(argumento_1, argumento_2, ... , ultimo_argumento)

Bibliotecas de Funciones

Se recordará que, cuando se analizó la instrucción *cout*, se vió que era necesario incluir en el programa a la biblioteca *iostream.h*, dado que la definición de cout se encontraba en dicha biblioteca. De la misma forma, para utilizar algunas funciones matemáticas será necesario incluir en nuestros programas otras bibliotecas de C++. Esta bibliotecas son, por ejemplo, **math.h** y **stdlib.h**. Esto significa que, en programas en los que se utilicen funciones predefinidas, será necesario utilizar la directiva **include** para incluir en el programa la definición de dichas funciones. En el caso de la biblioteca math.h, el programa deberá contener la instrucción:

#include <math.h>

A los archivos que tienen extensión **.h** se les conoce como archivos de encabezados. A través de la directiva **include**, los archivos de encabezados proporcionan al compilador la información básica contenida en la biblioteca correspondiente.

Algunas Funciones Predefinidas

Algunas funciones predefinidas se describen en la Tabla siguiente:

Nombre	Descripción	Tipo de Argumentos	Tipo de Valor de Regreso	Ejemplo	Valor	Biblioteca
sqrt	Raíz Cuadrada	double	double	sqrt(4.0)	2.0	math.h
pow	Potencia	double	double	pow(2.0,3.0)	8.0	math.h
abs	Valor absoluto de un int	int	int	abs(-7) abs(7)	7	stdlib.h
fabs	Valor absoluto de un double	double	double	fabs(-7.5) fabs(7.5)	7.5	math.h
ceil	Redondeo hacia el número inmediato superior	double	double	ceil(3.2) ceil(3.9)	4.0	math.h

floor	Redondeo hacia el número inmediato inferior	double	double	floor(3.2) floor(3.9)	3.0	math.h
sin	Seno	double	double	sin(0.0)	0.0	math.h
cos	Coseno	double	double	cos(0.0)	1.0	math.h
tan	Tangente	double	double	tan(0.0)	0.0	math.h

La más complicada de las funciones de la tabla es la función pow que sirve para obtener la potencia de un número. Por ejemplo, las siguientes sentencias son un ejemplo de aplicación de la función pow:

```
double resultado, x=3.0, y=2.0;
resultado = pow(x,y);
cout<< resultado;
```

Las sentencias anteriores mostrarían en pantalla al número 9.0.

Ejemplos

Las siguientes expresiones algebraicas y en C++ son equivalentes:

$\sqrt{x+y}$	sqrt(x+y)
x^{y+7}	pow(x,y+7)
$\lvert x-y \rvert$	abs(x-y)
sen(angulo)	sin(angulo)

El siguiente programa calcula las raíces la ecuación cuadrática $ax^2 + bx + c = 0$

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

int main()
{
     /* Este programa permite el calculo de las raices
        de una ecuacion cuadratica */

     /* Declaración de variables*/
     double a, b, c, x_1, x_2;

    /* Entrada de datos */
     cout<< "Dame los coeficientes a,b y c de la ecuacion cuadratica \n";
     cin>>a >> b >> c;

    /* Procesamiento de datos */

     x_1 = ( -b + sqrt( pow(b,2.0) – 4.0 * a * c) ) / (2.0 * a);
     x_2 = ( -b - sqrt( pow(b,2.0) – 4.0 * a * c) ) / (2.0 * a);

   /* Salida de Resultados */
     cout<<"\n";
     cout<<"La primera raiz es "<< x_1 << "\n";
     cout<<"La segunda raiz es "<< x_2 << "\n";
     cout<<"\n";

     system("PAUSE");
     return 0;
}
```

El siguiente programa también calcula las raíces de la ecuación cuadrática, pero considera el caso general en el que pueden haber raíces imaginarias:

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    /* Este programa permite el calculo de las raices
       de una ecuacion cuadratica */

    /* Declaración de variables*/
     double a, b, c, x_1, x_2;
     double x_1r, x_1i, x_2r, x_2i;

    /* Entrada de datos */
     cout<< "Dame los coeficientes a, b y c de la ecuacion cuadratica \n";
     cin>>a >> b >> c;

    /* Procesamiento de datos y Salida de Resultados */
     if ( (pow(b,2.0) - 4.0 * a * c) > 0)
     {
        x_1 = ( -b + sqrt( pow(b,2.0) - 4.0 * a * c) ) / (2.0 * a);
        x_2 = ( -b - sqrt( pow(b,2.0) - 4.0 * a * c) ) / (2.0 * a);

        cout<<"\n";
        cout<<"La primera raiz es "<< x_1 << "\n";
        cout<<"La segunda raiz es "<< x_2 << "\n";
        cout<<"\n";
     }
     else
     {
        x_1r = -b / (2.0 * a);
        x_1i = sqrt( fabs(pow(b,2.0) - 4.0 * a * c) )  / (2.0 * a);
        x_2r = -b / (2.0 * a);
        x_2i = -sqrt( fabs(pow(b,2.0) - 4.0 * a * c) )  / (2.0 * a);

        cout<<"\n";
        cout<<"La primera raiz es "<< x_1r <<" + " <<x_1i << " i \n";
        cout<<"La segunda raiz es "<< x_2r <<"  " <<x_2i << " i \n";
        cout<<"\n";
     }

     system("PAUSE");
     return 0;
}
```

DEFINICIÓN DE CONSTANTES

Con la finalidad de proporcionar un significado a los valores constantes que se utilizan en un programa, una práctica común en programación es asignar nombres a dichos valores y usar los nombres en el programa en lugar de utilizar los valores constantes.

C++ permite que cualquier cantidad, de cualquier tipo, pueda ser declarada (definida) como una constante. Una vez que se ha inicializado una cantidad que se considera constante, C++ no permitirá que su valor sea modificado durante la ejecución del programa.

Aunque no es estrictamente necesario, se acostumbra que los nombres de valores constantes sean escritos con **mayúsculas**. Para establecer que una cantidad va a poseer un valor constante, se utiliza el **modificador const**.

Por ejemplo, para declarar la variable INDICE como entera se utilizaría:

int INDICE;

Si además se desea que indice sea constante e igual a 3, se haría:

***const** int INDICE = 3;*

Declarando Constantes con el Modificador const

Cuando se inicializa una variable en una declaración, puedes definir también el hecho de que la variable no pueda cambiar su valor. Para ello se utiliza el modificador const en la declaración.

Sintaxis:

const nombre_del_tipo nombre_de_la_variable= valor_constante;

Ejemplos:

const double PI = 3.1415926;

const double R = 0.0821;

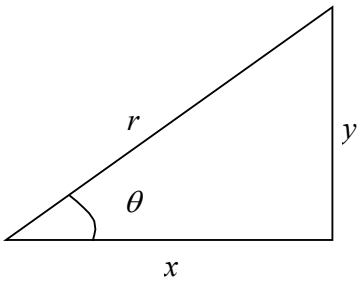
EJERCICIOS

1. Determine el valor de las siguientes expresiones en C++
   a) sqrt(16.0)
   b) pow(2.0,3.0)
   c) ceil(5.2)
   d) floor(5.2)
   e) sqrt(pow(3.0,2.0))
   f) 7/abs(-2)

2. Represente en C++ las siguientes operaciones aritméticas
   a) $\sqrt{x^2 + y^{1.5}}$
   b) $\tan\left(\frac{y}{x}\right)$
   c) $\sqrt{|x^3|}$
   d) $\sqrt[3]{x^2 + y^{1.5}}$
   e) sen(a) sen(b) – cos(a) cos(b)

3. Escriba un programa que, dado el diámetro de una esfera, calcule su volumen

4. Para la siguiente figura, haga un programa en el que dados θ y r, calcule los lados del triángulo rectángulo (x,y).



EJERCICIOS

Programa para el Cálculo del Volumen de una Esfera

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
int main()
{
    /* Este programa calcula el volumen de una esfera, dado
       el diametro de la misma */

    /* Declaración de variables*/
       double radio, diametro, volumen ;
       const double PI = 3.1415926;

    /* Procesamiento */

       cout<< "Dame el diametro de la esfera \n";
       cin>> diametro;

       radio = diametro / 2;
       volumen = (4*PI)/3 * pow(radio,3.0);

       cout<<"\n";
       cout<<"El  volumen de la esfera es "
            <<volumen<<" \n" << " \n";

       system("PAUSE");
       return 0;
}
```

Programa para el Cálculo de los Catetos de un Triángulo Rectángulo

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
int main()
{
    /* Este programa calcula el valor de los catetos de
       un triángulo rectángulo dados la hipotenusa y
       el ángulo entre ellos */

    /* Declaración de variables*/
      double cateto_opuesto,  cateto_adyacente,  hipotenusa, angulo;
      const double PI = 3.1415926;

    /* Procesamiento */

       cout<< "Dame la hipotenusa del triangulo \n";
       cin>> hipotenusa;

       cout<< "Dame el angulo que forman los catetos (en grados)\n";
       cin>> angulo;


       angulo = (angulo * PI) / 180;
       cateto_opuesto = hipotenusa * sin(angulo);
       cateto_adyacente = hipotenusa * cos(angulo);


       cout<<"\n";
       cout<<"El  cateto opuesto es "
            <<cateto_opuesto<<" \n";
       cout<<"El  cateto adyacente es "
            <<cateto_adyacente<<" \n";

       system("PAUSE");
       return 0;
}
```

FUNCIONES DEFINIDAS POR EL USUARIO

Las funciones que se han utilizado hasta ahora son funciones que el lenguaje de programación ha **predefinido** en sus bibliotecas o librerías. Sin embargo, también es posible que el programador defina y utilice sus **propias funciones**.

Definición de Funciones

Las funciones definidas por el programador se escriben "fuera" de la función *main*. Si se recuerda, uno de los objetivos del uso de las funciones es la descomposición de problemas complejos y el trabajo en grupo. El siguiente es un ejemplo de una definición de una función. Se utiliza antes de presentar la sintaxis formal de modo que nos podamos familiarizar con la terminología:

```
double square(double numero)
{
    double cuadrado;

    cuadrado = numero * numero;
    return cuadrado;
}
```

Esta definición de la función *square* consiste de:

1) El encabezado de la función

```
double square(double numero)
```

Observe que el encabezado no termina con punto y coma. Las partes del encabezado de una función son los siguientes.

a) Una lista de los argumentos de la función entre paréntesis:

(double numero**)**

Si se considera a la función como un programa pequeño, la lista de argumentos serían equivalentes a los datos que en un programa se introduciría a través de la instrucción cin. Es importante que observe que en la lista de argumentos se

indican tanto la lista de los argumentos necesarios como el tipo de cada uno de ellos.

b) El nombre de la función

square

que puede se cualquier identificador válido en C++.

c) El tipo de valor que regresa la función como resultado.

double

2) El cuerpo de la función

```
{
    double cuadrado;
    cuadrado = numero * numero;
    return cuadrado;
}
```

El cuerpo de la función se encierra entre llaves y en él se escriben las sentencias que se necesitan ejecutar para lograr el objetivo de la función. Contiene al menos una sentencia **return**. La sentencia return va seguida del nombre de una variable o de un valor constante.

return cuadrado;

El valor de dicha variable (o el valor de dicha constante) constituye el valor de regreso de la función. El tipo del valor que regresa la función debe ser consistente con el tipo del valor de regreso especificado en el encabezado de la función. Observe que, en este ejemplo, el valor de regreso de la función (tipo de la variable llamada cuadrado) es double, al igual que el tipo de valor de regreso especificado en el encabezado.

Llamado de Funciones

El llamado de las funciones definidas por el usuario se realiza de la misma forma que el llamado a funciones predefinidas. Por ejemplo, si un programador ha definido la función square de este ejemplo, los siguientes serían llamados válidos a la función:

```
double x,y,z;
x=2.0;
y = square(10.0);
z = square(x+y);
```

Observe que los argumentos que se pasan a la función (*10.0* en el primer caso y *x+y* en el segundo caso) son del tipo double y coinciden con el tipo definido para el argumento de la función. Asimismo, a la variable *y* se le asigna el valor de regreso de la función *square(10.0*). Esto es correcto porque el valor de regreso de la función es de tipo double y la variable *y* también es de tipo double. Note que no es necesario que los argumentos con que la función es llamada se nombren igual que los identificadores usados en el encabezado de la función.

Uso de Funciones Definidas por el Programador

Si se recuerda, cuando se utilizan funciones predefinidas es necesario incluir a la biblioteca o librería que contiene su definición. Por ejemplo, si se usa la función pow, es necesario incluir math.h

Algo similar es necesario para funciones definidas por el programador. Ese algo es que, después de las directivas *include* pero antes de la función main, es necesario escribir el **prototipo de la función**. El prototipo de la función no es más que el encabezado de la función seguido de punto y coma. El siguiente ejemplo muestra como se podría utilizar en un programa una función definida por el programador.

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

double square(double numero);     /* Prototipo de la funcion*/

int main()
{
     /* Este programa es solo un ejemplo de la definicion y uso
        de una funcion definida por el programador*/

     double x,y,z;

     cin>>x>>y;

     z = square(x);      /* Llamado a la funcion*/
     z = z * y;

     cout<<"\n";
     cout<<z;
     cout<<"\n";

     system("PAUSE");
     return 0;
}

/* Esta funcion calcula el cuadrado de un numero*/
double square(double numero)
{
     double cuadrado;

     cuadrado = numero * numero;
     return cuadrado;
}
```

EJERCICIOS

1. Escriba la definición y el prototipo de una función que recibe tres argumentos de tipo entero y que regresa el promedio de sus tres argumentos.

2. Escriba la definición y el prototipo de una función que toma un argumento de tipo de numérico de punto flotante. Como resultado, la función regresa el caracter 'N', si el argumento es cero o negativo, o el caracter 'P' si el argumento es positivo.

3. ¿Qué muestra en pantalla el siguiente programa?

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

char misterio(int argumento_1, int argumento_2);
int main()
{
    cout<< "m" << misterio(10,9) << "no \n";
    cout<<  " \n";
    system("PAUSE");
    return 0;
}

char misterio(int argumento_1, int argumento_2)
{
    if (argumento_1 >= argumento_2)
       return 'a';
    else
       return 'o';
}
```

EJERCICIOS

1. Escriba un programa que utilice la función que ejecuta el cálculo del factorial de un número entero. En dicho programa se deberá calcular el numero de combinaciones posibles que se tienen para seleccionar *m* números a partir de un total de *n* números.

2. Debido a la inflación, los costos de los bienes cambian con el tiempo. Así, por ejemplo, si se desea comprar un auto dentro de dos años, no se debería de considerar únicamente el precio actual del automóvil, si no que debería también tomarse en cuenta el aumento de precio que tendrá debido a la inflación. Escriba un programa que reciba como datos el costo de un producto en la actualidad, el número de años a partir de la fecha actual en que se va a comprar el producto y la tasa de inflación anual. El resultado del programa debe ser el costo del producto al tiempo en que se va a realizar la compra. Se debe realizar el programa de forma que el cálculo del precio del producto se realice a partir de la definición de una función.

3. El calor específico (Cp) y la presión de vapor (Pv) son dos cantidades usadas comúnmente en cálculos de ingeniería química. Ambas propiedades suelen tener valores diferentes para cada sustancia, y ambas dependen de la temperatura. Dos expresiones utilizadas con frecuencia para determinar estas cantidades son:

$$\frac{C_p}{R} = a + bT + cT^2 + \frac{d}{T^2}$$

$$\ln(P_v) = A - \frac{B}{T + C}$$

donde *T* es la temperatura en grados Kelvin, Pv es la presión de vapor en kPa, Cp es el calor específico en (cal / (mol K)) y R vale 1.987.

a, *b*, *c*, *d*, *A*, *B* y *C* son constantes que dependen de la sustancia en particular. Elabore un programa que, dados los valores de dichas constantes y el valor de la temperatura, calcule los valores de la presión de vapor y del calor específico. El programa deberá obtener el resultado a través del llamado a dos funciones dados los parámetros necesarios. Una función realiza el cálculo de la presión de vapor y la otra realiza el cálculo de la capacidad calorífica.

```
/* Este programa calcula el numero de combinaciones de m numeros
   que se pueden obtener a partir de un total de n numeros (donde
   n>m) */
#include <iostream.h>
#include <stdlib.h>

int factorial(int numero);   /* Prototipo de la funcion que calcula
                                el factorial de un numero */
int main()
{
     /* Declaracion de Variables  */
     int combinaciones;
     int n, m;

     /* Entrada de datos  */
     cout<<"Dame el numero total de numeros (n) y el tamano de la muestra(m)\n"
     <<"Presiona Enter despues de cada numero \n";
     cin>>n >>m;
     cout<<"\n";
     /* Procesamiento */
     combinaciones = factorial(n) / (factorial(m) * factorial(n-m));
      /*  Salida de Resultados  */
     cout<<"El numero de combinaciones de " <<m   <<" numeros "
     <<"a partir de " <<n << "\n" << "numeros es "<<combinaciones <<"\n";
     cout<<"\n";

     system("PAUSE");
     return 0;
}

/* Funcion que calcula el factorial de un numero entero*/
int factorial(int numero)
{
  /*  Declaracion */
  int contador;
  int producto;

  /* Inicializacion*/
  contador = 1;
  producto = 1;

  /*  Calculo iterativo del factorial de numero */
  while(contador <= numero)
  {
     producto = producto * contador;
     contador++;
  }
  return producto;     /* Valor de regreso */
}
```

```
#include <iostream.h>
#include <stdlib.h>
/* Prototipo de la Funcion */
double nuevo_precio(double price, double rate, int years);

/* Funcion Principal */
int main()
{
 /* Este programa esta diseñado para calcular el precio de un bien
    cualquiera en un tiempo determinado tomando en cuenta la tasa
    de inflacion anual */

 /* Declaracion de variables */
    double precio_actual,tasa, precio_a_futuro;
    int periodo;

 /* Entrada de datos */
    cout<<"Dame el precio actual del producto en pesos. \n";
    cin>>precio_actual;
    cout<<"\n";
    cout<<"Dentro de cuantos anios realizara la compra.\n";
    cin>>periodo;
    cout<<"\n";
    cout<<"Dame la tasa de interes en procentaje \n";
    cin>>tasa;
    cout<<"\n";

 /* Procesamiento de datos */
    precio_a_futuro = nuevo_precio(precio_actual, tasa, periodo);

 /* Salida de datos */
    cout<<"El precio del producto luego de "<< periodo
        << " anios es " << precio_a_futuro <<"\n" <<"\n";;

    system("PAUSE");
    return 0;
}

/*  Funcion que realiza el calculo del precio de un producto
   en el futuro considerando la inflacion */
double nuevo_precio(double price, double rate, int years)  /* encabezado */
{
    int counter;

    rate = rate / 100.0;
    counter = 1;
    while (counter <= years)
    {
     price = price * ( rate + 1.0);
     counter = counter + 1;
    }
    return price;    /*Valor de regreso, resultado*/
}
```

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

/* Prototipos de Funciones */
/* Calculo de calor especifico */
double heat_capacity(double c1, double c2,
                     double c3, double c4, double T);

/* Calculo de presion de vapor */
double vapor_pressure(double C1, double C2,
                      double C3, double T);

int main()
{
  /*Este programa sirve para calcular el calor especifico y la presion de
    vapor de una sustancia conociendo sus constantes y su temperatura
    en grados kelvin*/

  /*Declaracion de variables*/
    double Cp, Pv, Temperatura;
    double a,b,c,d,A,B,C;

  /*Entrada de datos*/
    cout<<"Dame el valor de las cuatro constantes para el calculo del Cp\n";
    cout<<"Presiona enter despues de cada valor \n";
    cin>> a >> b >> c >> d;
    cout<<"\n";
    cout<<"Dame el valor de las tres constantes para el calculo de Pv.\n";
    cout<<"Presiona enter despues de cada valor \n";
    cin>> A >> B >> C;
    cout<<"\n";
    cout<<"Dame el valor de la Temperatura en grados Kelvin\n";
    cin>> Temperatura;
    cout<<"\n";

  /* Procesamiento de Datos */
    Cp = heat_capacity(a, b, c, d, Temperatura);
    Pv = vapor_pressure(A, B, C, Temperatura);

  /* Salida de Resultados */
    cout<<"\n";
    cout<<"El calor especifico es de la sustancia es "<<Cp<<" cal/(mol K) \n";
    cout<<"\n";
    cout<<"La presion de vapor es de la sustancia es "<<Pv<<" KPa \n";
    cout<<"\n";

     system("PAUSE");
     return 0;
}
```

```
/* Funcion para el calculo de calor especifico */
double heat_capacity(double c1, double c2,
                     double c3, double c4, double T)
{
    double calor_especifico;
    const double R=1.987;

    calor_especifico=( c1 + c2* T + c3 *pow(T,2.0) +(c4 / pow(T,2.0)) )*R;

    return calor_especifico;   /* valor de regreso */
}

/* Funcion para el calculo de la presion de vapor */
double vapor_pressure(double C1, double C2,
                      double C3, double T)
{
    double presion_de_vapor;

    presion_de_vapor = exp( C1 - C2 /(T + C3));

    return presion_de_vapor;  /*  valor de regreso */
}
```

VARIABLES LOCALES Y VARIABLES GLOBALES

Como se discutió en clase, cuando uno llama una función (como la función *sqrt*), no es necesario saber los tipos y los nombres de las variables que se declaran en la definición de dicha función. Simplemente se requiere de conocer el tipo de los argumentos que necesita y el tipo de valor que regresa.

Esto se debe a que en C++ las variables que se definen en las funciones son independientes de las variables que se definen en la función main (programa principal) o en cualquier otra función. Si se declara una variable en una función y se declara otra variable con el mismo nombre en la función principal del programa (main), estas dos variables, aunque tengan el mismo nombre, se consideran como dos variables diferentes. Observe el siguiente ejemplo:

```
#include <iostream.h>
#include <stdlib.h>
double square(double number);     /* Prototipo de la funcion*/
int main()
{
     /* Este programa servira para ejemplificar el uso de variables locales*/

     double cuadrado;
     double numero;
     cout<<"Dame el valor de un numero \n";
     cin>>numero;

     cuadrado = square(numero);

     cout<<"\n";
     cout<<" El cuadrado del numero es "<<cuadrado;
     cout<<"\n";

     system("PAUSE");
     return 0;
}
/* Funcion para el calculo del cuadrado de un numero*/
double square(double number)
{
     double cuadrado;

     cuadrado = number * number;
     return cuadrado;
}
```

la variable cuadrado se declara tanto en la función main como en la función square y la declaración es idéntica:

double cuadrado;

Sin embargo, la variable cuadrado de la función main y la variable cuadrado de la función square se consideran **como dos variables diferentes**.

Variables Locales

Las variables que se declaran dentro del cuerpo de una función se dice que son **locales** a dicha función o que tienen un alcance **local**. En el ejemplo, hay una variable llamada cuadrado que es local a la función main y otro variable también llamada cuadrado pero que es local a la función square. Dichas variables son diferentes. Los valores que toma una variable local dentro de una función pueden ser completamente distintos a los que tome una variable del mismo nombre pero que sea local a otra función.
Se dice que una variable local es usada (definida, modificada, etc.) únicamente dentro de la función en la que es definida

Constantes y Variables Globales

Las constantes y las variables también pueden tener alcances **globales**. Es decir, es posible que una misma variable o constante puedan ser utilizadas por **diferentes funciones** y que (en el caso de variables) puedan ser modificadas por cada una de ellas. Cuando esto ocurre, se dice que se tiene constantes globales o variables globales.

Para que el compilador considere a una variable como global, es necesario definirlas al comienzo del programa, antes de la función main y del resto de las funciones del programa. Es práctica común colocar todas las constantes globales y las variables globales en grupo, justo por debajo de las directivas include y antes de los prototipos de las funciones.

El uso de constantes globales es muy común, sin embargo, el uso de variables globales generalmente no es necesario y muy frecuentemente provoca dificultades en entender del algoritmo del programa, por lo cual las variables globales casi no se utilizan. En el siguiente ejemplo, observe que la constante PI es una constante global, mientras que las variables area y volumen son variables locales definidas en la función main y en otras funciones. No obstante que hay variables con el mismo nombre, éstas son variables locales y por tanto se trata de variables diferentes.

```
//Programa para el calculo del area de un circulo y el volumen de una esfera
//usando dos funciones. Se usa el mismo valor de radio para los dos calculos.
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

const double PI = 3.14159;  /*  Tiene alcance global. Se coloca por debajo de
                                los include y antes de los prototipos */

/* Prototipos */
double area_circulo(double radio);
double volumen_esfera(double radio);

int main( )
{
   double radio_de_ambos, area, volumen;  /*Alcance local*/

   cout<<"Dame el radio del circulo y de la esfera \n";
   cin>>radio_de_ambos;

   area = area_circulo(radio_de_ambos);
   volumen = volumen_esfera(radio_de_ambos);

   cout<<"El area del circulo es "<<area<<"\n";
   cout<<"El volumen de la esfera es "<<volumen<<"\n";
   cout<<"\n";

   system("PAUSE");
   return 0;
}

/*Observe que las dos funciones siguientes usan la constante global PI*/
```

```
/* Funcion para el calculo del area de un circulo */
double area_circulo(double radio)
{
    double area;  /* Alcance local, diferente de la variable definida en main */
    area = PI * pow(radio, 2.0);        /* Uso de constante global */
    return area;
}


/* Funcion para el calculo del volumen de una esfera */
double volumen_esfera(double radio)
{
    double volumen; /*Alcance local, diferente de la variable definida en main*/
    volumen = (4.0/3.0) * PI * pow(radio, 3.0);
    return volumen;       /* Uso de constante global */
}
```

FUNCIONES SIN VALOR DE REGRESO: TIPO VOID

En las funciones que se han discutido hasta ahora siempre se ha tenido un valor de regreso, que es el resultado de ejecutar las sentencias de la función. Sin embargo, es posible definir funciones que ejecutan cualquier tipo de sentencias, pero que no proporcionan un valor de regreso.

Para hacer saber al compilador que una función no produce algún valor de regreso se utiliza el identificador **void**. Así por ejemplo, la función llamada:

void mensajes_de_salida (double area_circulo, double volumen_esfera)

es una función que requiere de dos argumentos de tipo double pero que, dado que se define como de tipo void, no proporcionará ningún valor de regreso.

Para llamar a una función sin valor de regreso no es necesario hacer una asignación (como se hace para las funciones que si proporcionan valor de regreso), sino que basta utilizar el nombre de la función, definir los argumentos y terminar con un punto y coma. Por ejemplo, el siguiente es un llamado a la función mensajes de salida:

```
mensajes_de_salida(area, volumen);
```

Para que esto quede claro, usemos el mismo ejemplo del cálculo del área de un círculo y volumen de una esfera. En este caso, los resultados se mostrarán utilizando la función mensajes_de_salida que es una función sin valor de regreso:

```
//Programa para el calculo del area de un circulo y el volumen de una esfera
//usando dos funciones. Se usa el mismo valor de radio para los dos calculos.
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

const double PI = 3.14159;  /*  Tiene alcance global. Se coloca por debajo de
                                los include y antes de los prototipos */

/* Prototipos */
double area_circulo(double radio);
double volumen_esfera(double radio);
void mensajes_de_salida(double area_circulo, double volumen_esfera);

int main( )
{
   double radio_de_ambos, area, volumen;  /*Alcance local*/

   cout<<"Dame el radio del circulo y de la esfera \n";
   cin>>radio_de_ambos;

   area = area_circulo(radio_de_ambos);
   volumen = volumen_esfera(radio_de_ambos);

   mensajes_de_salida(area, volumen); /*Uso de funcion sin valor de regreso*/

   system("PAUSE");
   return 0;
}

/*Observe que las dos funciones siguientes usan la constante global PI*/


/* Funcion para el calculo del area de un circulo */
double area_circulo(double radio)
{
    double area;  /* Alcance local, diferente de la variable definida en main */
```

```
    area = PI * pow(radio, 2.0);        /* Uso de constante global */
    return area;
}


/* Funcion para el calculo del volumen de una esfera */
double volumen_esfera(double radio)
{
    double volumen; /*Alcance local, diferente de la variable definida en main*/
    volumen = (4.0/3.0) * PI * pow(radio, 3.0);
    return volumen;       /* Uso de constante global */
}

/* Funcion para mostrar los resultados del calculo*/
void mensajes_de_salida(double area_circulo, double volumen_esfera)
{
    cout<<"\n";
    cout<<"El area del circulo es "<<area_circulo<<"\n";
    cout<<"El volumen de la esfera es "<<volumen_esfera<<"\n";
    cout<<"\n";

}
```

Observe que la función mensajes_de_salida no regresa ningún valor como resultado (no hay sentencia return), simplemente ejecuta sentencias para mostrar los resultados en pantalla. Sin embargo, es posible usar la palabra reservada return dentro de funciones que no regresan ningún valor como en el ejemplo siguiente:

```
/* Funcion para mostrar los resultados del calculo*/
void mensajes_de_salida(double area_circulo, double volumen_esfera)
{
    cout<<"\n";
    cout<<"El area del circulo es "<<area_circulo<<"\n";
    cout<<"El volumen de la esfera es "<<volumen_esfera<<"\n";
    cout<<"\n";
   return;
}
```

En este caso, el identificador return no está asociado a ningún valor ni a ninguna variable, y sólo se utiliza para indicar que la función ha terminado de ejecutar sus sentencias.

MÚLTIPLES DEFINICIONES DE UNA FUNCIÓN

C++ permite que se le asigne el mismo nombre a diferentes funciones. Existen, sin embargo, algunas restricciones que serán mencionadas un poco más adelante. Por ejemplo, suponga que en un mismo programa se tienen las siguientes definiciones de una función:

```
//Función que calcula el promedio de dos numeros
double promedio (double numero1, double numero2)
{
  double valor_prom;
  valor_prom = (numero1 + numero2)/2.0;
  return valor_prom;
}

//Función que calcula el promedio de tres numeros
double promedio (double numero1, double numero2, double numero3)
{
  double valor_prom;
  valor_prom = (numero1 + numero2 + numero3)/3.0;
  return valor_prom;
}
```

Lo anterior es correcto (aunque no recomendable) y no sería marcado como error por parte del compilador. El compilador distinguiría una función de la otra con base al número de argumentos que se utilicen cuando se llame a la función. Observe también que algunas variables en dichas funciones tienen el mismo nombre, pero no importa, dado que se trata de variables locales definidas dentro de una función.

El siguiente es un programa que llama a las dos funciones anteriores:

```
//Programa para el calculo del promedio de 2 y 3 numeros.
#include <iostream.h>
#include <stdlib.h>


/* Prototipos */
//Funcion que calcula el promedio de dos numeros
double promedio (double numero1, double numero2);
//Funcion que calcula el promedio de tres numeros
double promedio (double numero1, double numero2, double numero3);

int main( )
{
   double numero_1, numero_2, numero_3, prom_2, prom_3;

   cout<<"Dame el valor de 3 numeros caulesquiera \n";
   cin>>numero_1 >>numero_2 >>numero_3;

   prom_2= promedio(numero_1, numero_2);
   prom_3 = promedio(numero_1, numero_2, numero_3);

    cout<<"\n";
    cout<<"El promedio de los 2 primeros numeros es "<<prom_2<<"\n";
    cout<<"El promedio de los 3 numeros es "<<prom_3<<"\n";
    cout<<"\n";

   system("PAUSE");
   return 0;
}
```

Existe sin embargo la siguiente restricción para que sea posible definir funciones diferentes con el mismo nombre:

Cuando dos o más funciones tienen el mismo nombre, las definiciones de tales funciones deben de ser diferentes en lo que respecta a sus argumentos. Es decir, el numero o el tipo de sus argumentos debe de ser diferente.

Así, lo siguiente sería incorrecto:

```
//Función que calcula el promedio aritmetico de dos numeros
double promedio (double numero1, double numero2)
{
  double valor_prom;
  valor_prom = (numero1 + numero2)/2.0;
  return valor_prom;
}

//Función que calcula el promedio geometrico de dos numeros
double promedio (double numero1, double numero2)
{
  double valor_prom;
  valor_prom = sqrt(numero1 * numero2);
  return valor_prom;
}
```

Si se usan en forma independiente, ambas funciones son correctas. Sin embargo, si se usan juntas, existe un error, dado que se tienen dos funciones con el mismo nombre, con el mismo numero de argumentos y con el mismo tipo de argumentos. Así, cuando se llame a las funciones, el compilador no sabría distinguir a cual de las dos se está llamando. No obstante, aún cuando las variables usadas en las funciones se llamen igual, no existe error en ese sentido. Por último, observe que un programa que llame a la segunda función requeriría incluir a la librería math.h, mientras que en el caso de la primer función esto no es necesario.

EJERCICIOS

1. Luego de que se ejecuta el siguiente programa, ¿cuál es el valor de la variable area de la función main? ¿cuál es el valor de la variable area de la función justo cuando se ha terminado de ejecutar dicha función?

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

double area_circulo(double radio);
int main( )
{
   double radio, area, lado, area_circ;

   radio = 4;
   lado = 3;

   area_circ = area_circulo(radio);
   area= lado * lado;

   system("PAUSE");
   return 0;
}

double area_circulo(double radio)
{
    double area;
    const double PI = 3.14159;
    area = PI * pow(radio, 2.0);
    return area;
}
```

2. ¿Qué mostraría en pantalla el siguiente programa?

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

const double PI = 3.14159;
double area;

void area_circulo(double radio);
```

```
int main( )
{
   double radio, altura;

   radio = 4;
   altura = 3;
   area = 0;

   area_circulo(radio);
   area= area * 2.0 + PI * altura * 2.0 * radio ;
   cout<< area<<"\n";
   system("PAUSE");
   return 0;
}

void area_circulo(double radio)
{
    area = PI * pow(radio, 2.0);
}
```

3. Escriba una función sin valor de regreso que muestre el valor de la variable area calculada en el programa anterior.

4. Escriba una función sin valor de regreso que reciba como argumento una variable con valor entero y dos enteros constantes llamados OPMIN y OPMAX (donde OPMAX>OPMIN). La función mostrará en pantalla la palabra VERDADERO si el entero se encuentra entre los valores de las dos constantes y la palabra FALSO si el entero no se encuentra entre dichos valores.

UNIDAD IV

TEMA I

ARCHIVOS (FICHEROS)

ARCHIVOS (FICHEROS)

Siempre que se usa una computadora, se utilizan archivos para almacenar la información. Por ejemplo, cuando se crea un programa en C++, se crea un archivo con extensión *cpp* que contiene el código del programa.
También se pueden utilizar archivos para almacenar los resultados que se obtienen en un programa y para leer los datos que requiere un programa. Este documento explica el uso de archivos I/O en C++.

Archivo I/O

El término I/O (Input/Output) se refiere a la "***entrada***" de datos y la "***salida***" de resultados de un programa.
Hasta ahora, se han manejado la sentencia ***cout*** para "salida" de resultados y la sentencia ***cin*** para "entrada" de datos a un programa. cout y cin envían los mensajes o leen la información a/desde la **pantalla** de la computadora. El mecanismo por el cual la información se lee y se envía a la computadora a través de la pantalla se dice que es salida/entrada ***estándar***.
Es posible, sin embargo, en lugar de enviar mensajes a pantalla enviarlos a un archivo. De la misma forma, en lugar de leer datos a través de la pantalla, es también posible leerlos desde un archivo.
Ambas formas son válidas y útiles. La diferencia es que cuando se utiliza la salida a pantalla, la información se obtiene sólo en forma **temporal**, pues se pierde cuando termina de ejecutarse el programa y la ventana desaparece. Por otra parte, si los resultados se envían a un archivo, este archivo queda guardado en el disco aún después de que el programa termina de ejecutarse. Por ello se dice que los archivos son una forma de almacenar los datos en forma **permanente**.
Los archivos que se guardan y se leen se dice que son de tipo texto, pues pueden contener únicamente caracteres en código ASCII (como letras, números, algunos símbolos, etc). Así, es posible editar archivos de este tipo con

cualquier editor de texto (como Notepad, Wordpad, Word, etc.) siempre y cuando se guarden como archivos de texto.
Cuando un programa envía información a un archivo, se dice que el archivo es de *escritura*. Cuando un programa lee información desde un archivo, se dice que el archivo es de *lectura*.

TIPOS DE ARCHIVOS I/O

Además de la clasificación en lo que respecta a si el archivo es de escritura o de lectura, los archivos I/O se clasifican en dos categorías: Archivos secuenciales o archivos de acceso aleatorio.

Archivos Secuenciales

En estos archivos, la información sólo puede leerse y escribirse empezando desde el principio del archivo. Los archivos secuenciales tienen algunas características que hay que tener en cuenta:
1. La escritura de nuevos datos siempre se hace al final del archivo.
2. Para leer un dato concreto del archivo hay que avanzar siempre hasta donde se encuentre dicho dato. Si el dato requerido se encuentra antes del dato en que está se está posicionado el archivo en un momento dado, será necesario regresar al comienzo del archivo y avanzar hasta el dato necesario.

Archivos de Acceso Aleatorio

Los archivos de acceso aleatorio son más versátiles, permiten acceder a cualquier parte del archivo en cualquier momento, como si fueran arreglos en memoria. Las operaciones de lectura y/o escritura pueden hacerse en cualquier punto del archivo.

En realidad C++ no distingue si los archivos que usamos son secuenciales o de acceso aleatorio, es el tratamiento que hagamos de ellos lo que los clasifica como de uno u otro tipo. En el curso se considerará sólo el caso en el que el archivo se lee en orden desde el principio hasta el final (archivos secuenciales).

DECLARACIÓN DE ARCHIVOS I/O Y APLICACIONES

Si se recuerda, las instrucciones cin y cout se definen en la librería *iostream.h*, y es por ello que siempre incluimos esa librería en nuestro programa. En forma similar, cuando se desea usa archivos I/O, es necesario incluir la librería *fstream.h*, es decir, se debe incluir en el programa:

```
#include <fstream.h>
```

Asimismo, cin y cout pueden verse como dos comandos que ya han sido definidos y que "conectan" la información del programa con la pantalla del computador. Si se quiere que la información se "conecte" con un archivo es necesario declarar dos nuevos comandos que sirvan para ese propósito y que se puedan usar en lugar de cin y cout. Una declaración típica que se realiza es la siguiente:

```
ifstream  in_stream;
ofstream   out_stream;
```

Esto puede interpretarse como una declaración de dos "comandos" (in_stream y out_stream) que pueden utilizarse en lugar de cin y cout. ifstream ("input-file-stream") y ofstream(output-file-stream) se encuentran definidos en la librería fstream.h.

Si se realiza la declaración anterior, entonces sería posible utilizar in_stream y out_stream exactamente como se han usado cin y cout. La diferencia es que la salida de resultados y la entrada de datos se harían a y desde un archivo, respectivamente. Por ejemplo:

```
int numero;
in_stream>>numero; /*  Lee el valor de numero desde un archivo*/
out_stream<<"El numero es ">>numero>>"\n";/*Escribe datos a un archivo*/
```

Ahora, en el ejemplo anterior se habla de que se escribe y se lee a y desde un archivo. ¿Desde/de cuál archivo? La respuesta a esa pregunta se debe de incluir en el programa a través de instrucciones como la siguiente:

```
in_stream.open("entrada.dat");
out_stream.open("salida.dat");
```

Instrucciones como las anteriores se deben de incluir en el programa antes de usar in_stream y out_stream. Lo que las dos instrucciones anteriores no están indicando es que el programa utilizará un archivo denominado entrada.dat para leer datos y que va a escribir los resultados en un archivo llamado salida.dat. Se dice que las instrucciones anteriores están "abriendo" archivos para que éstos puedan ser usados para leer o escribir datos.

Por último, una vez que el programa se ha ejecutado, es necesario "cerrar" los archivos que se "abrieron". Para ello se utiliza:

```
in_stream.close();
out_stream.close();
```

Resumen de Sentencias e Instrucciones para Usar Archivos I/O

1) Incluir la librería correspondiente:
   #include <fstream.h>

2) Seleccione los nombres para dos comandos que utilice en lugar de cin y cout para leer y escribir de/desde un archivo. Si por ejemplo selecciona a_ent (para entrada en lugar de cin) y a_sal (para salida en lugar de cout), deberá declararlos como:

   ```
   ifstream a_ent;
   ofstream a_sal;
   ```

3) Conecte los comandos al archivo en que quiera guardar los resultados o al archivo del cual quiera leer los datos. Suponga que se quieren guardar los datos en un archivo llamada salida.dat y se quieren leer los datos desde el archivo entrada.dat. Se deberá usar entonces:

```
a_ent.open("entrada.dat");
a_sal.open("salida.dat");
```

4) A partir de entonces, cada vez que use a_sal es como usar cout, pero la salida es al archivo, y cada vez que use a_ent es como usar cin, pero la lectura de datos es desde el archivo. Ejemplo:

```
a_ent>>variable;
a_sal<<variable2;
```

5) Cierre los archivos luego de utilizarlos:

```
a_ent.close();
a_sal.close();
```

EJEMPLO

```
#include <fstream.h>
#include<iostream.h>
#include<stdlib.h>

int main()
{
  /* Este programa pide 3 numero a un usuario y luego los escribe en
     el archivo salida.dat */
  int x[3];
  ofstream csalida;

  csalida.open("salida.dat");

  /* Observe que cin y cout se pueden seguir usando como antes */
  cout<<"Dame el valor de 3 numeros enteros \n"
      <<"Presiona Enter despues de cada uno \n";
  cin>>x[0]>>x[1]>>x[2];

  /* Escribir los valores en el archivo */
  csalida<<x[0]<<"\t"<<x[1]<<"\t"<<x[2]<<"\n";

  /* Cierra el archivo */
  csalida.close();

  system("PAUSE");
  return 0;
}
```

PRÁCTICA DE APLICACIÓN DE ARCHIVOS

En este documento se explica la forma en que se pueden utilizar archivos I/O para guardar los resultados o leer los datos de un programa. Esta práctica pretende mostrar la diferencia que existe entre usar archivos y usar la pantalla (también conocida como salida estándar).

1) Codifique lo siguiente.

```
#include <fstream.h>
#include<iostream.h>
#include<stdlib.h>

int main()
{
  /* Este programa evalúa la integral de   x dx   entre dos límites
     y va  guardando los resultados parciales en un archivo */
  int n;
  double x, a, b, integral, dx;
  ofstream   csalida;  /* Instruccion similar a cout */

  csalida.open("resultados.dat"); /* Este es el archivo de resultados */

  /* Observe que cin y cout se pueden seguir usando como antes */
  cout<<"Dame el valor de los limites de la integral de \n"
      <<"xdx    Proporcione el limite inferior, presione \n"
      <<"Enter y proporcione entonces el limite superior \n";
  cin>>a>>b;  /* Corra el programa con 0 y 1 como valores de a y b */

  /* Se usan 100 elementos diferenciales */
  dx = (b-a)/100.0;

  /* Inicializa las variables del calculo de la integral */
  integral =0;
  x = a;

  /* Calcule el producto diferencial */
  for(n=1; n<=100; n++)
  {
     x = x + dx;
     integral = integral + x * dx;
     csalida<<x<<"\t"<<integral<<"\n";
  }
```

```
  /* Cierra el archivo */
  csalida.close();

  cout<<"\n";
  system("PAUSE");
  return 0;
}
```

2) Use el explorador de windows para buscar el archivo resultados.dat

Este archivo deberá estar en el directorio en el que se estuvo guardando el programa anterior. Cómo se generó dicho archivo. Que contiene?

3) Escriba el siguiente programa

```
#include <fstream.h>
#include<iostream.h>
#include<stdlib.h>

int main()
{
  /* Porque este programa muestra en pantalla los numeros
     0.01 y 0.0001? */

  double x, integral;
  ifstream   c_ent;  /* Instruccion similar a cin */

  c_ent.open("resultados.dat"); /* Este es el archivo de datos */

  c_ent>>x>>integral;

  cout<<x<<"\t"<<integral<<"\n";

  c_ent.close();

  cout<<"\n";
  system("PAUSE");
  return 0;
}
```

EJERCICIOS

1. Escriba un programa en C++ que genere un archivo llamado *intermedio.dat* Este archivo deberá contener la lista de los números pares entre el 2 y el 100 (inclusive). Escriba luego un programa que lea los números del archivo *intermedio.dat* y los sume, de forma que envíe a pantalla el resultado de dicha suma. Observe que tendría que ejecutar el primer programa para generar el archivo antes de ejecutar el segundo programa (dado que el segundo programa hace uso del archivo).

2. Escriba un programa en C++ que muestre una tabla con los valores de la raíz cuadrada de los números del 1 al 10. Asimismo, la tabla de valores se deberá guardarse también en un archivo llamado *raiz.dat*. El archivo debería quedar mostrar una tabla como la siguiente:

```
-------------------------------------
Numero        Raiz Cuadrada
-------------------------------------
   1              1
   2              1.41421
   ⋮                 ⋮
   10             3.16228
```

ARCHIVOS DE ACCESO ALEATORIO

Las instrucciones necesarias para la escritura y la lectura de archivos de acceso aleatorio van más allá de los objetivos de este curso. Sin embargo, para el lector interesado en su aprendizaje, se recomienda utilizar la referencia 6 y estudiar las instrucciones: seekg, seekp, tellg, tellp, read y write. Asimismo, será necesario que se estudie la forma general de declarar archivos (en la que se indica si el archivo es de lectura o escritura y se indica cual es la posición inicial para la lectura o la escritura).

SOLUCIÓN DEL EJERCICIO 2

```
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

int main()
{
  /* Este programa calcula la raiz cuadrada de los numeros del 1
     al 10, los muestra en pantalla y los escribe en
     el archivo raiz.dat */

  double numero, raiz;
  ofstream csalida;

  csalida.open("raiz.dat");

  /* Sentencias para hacer la Tabla en pantalla*/
  cout<<"--------------------------- \n";
  cout<<"Numero \t"<<"Raiz Cuadrada \n";
  cout<<"--------------------------- \n";

  /* La misma Tabla en archivo */
  csalida<<"-------------------------- \n";
  csalida<<"Numero \t"<<"Raiz Cuadrada \n";
  csalida<<"-------------------------- \n";

  /* Ciclo para realizar el calculo y mostrar resultados */
  for(numero=1.0; numero<=10.0; numero=numero+1.0)
  {
     raiz = sqrt(numero);
     cout<<numero<<"\t"<<raiz<<"\n"; /* Salida a pantalla */
     csalida<<numero<<"\t"<<raiz<<"\n"; /* Salida a archivo */
  }

  cout<<"\n";

  /* Cierra el archivo */
  csalida.close();

  system("PAUSE");
  return 0;
}
```

UNIDAD IV

TEMA II

CADENAS DE CARACTERES

CADENAS DE CARACTERES

Hasta ahora se han manejado variables de tipo caracter cuyo valor es *un sólo caracter*. En algunos casos, sin embargo, es necesario usar variables cuyos valores sean un conjunto (*cadena*) de caracteres, como en bases de datos con nombres de personas, instituciones, etc. Una cadena de caracteres podría verse como un conjunto de caracteres (numero, letras, símbolos de código ascii) ordenados.

Aunque no se había visto de esa manera, todos los mensajes que enviamos a pantalla entre comillas usando la instrucción cout son en realidad cadenas de caracteres. Por ejemplo, la sentencia:

cout<<"Hola a todos ";

permite que se muestra en pantalla el conjunto de caracteres *H, o, l, a, espacio, a, espacio, t, o, d, o, s, espacio*.

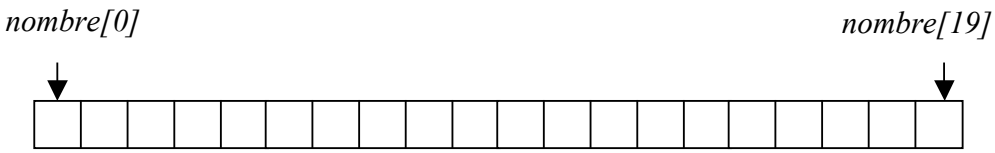
Obviamente, el procesador de la computadora no sabe nada acerca del idioma español, por lo que el procesador simplemente muestra los caracteres anteriores en el orden que se le indica.

En C++, como en casi cualquier otro lenguaje de programación, es posible utilizar variables para almacenar y manipular cadenas de caracteres. La forma de hacer esto es a través de arreglos. Por ejemplo, la siguiente es la declaración de una variable (llamada *nombre*) que permite guardar una cadena de 19 caracteres:

char nombre[20];

Observe que se dijo que dicho arreglo, aunque tiene 20 elementos, puede contener una cadena de sólo 19 caracteres. Esto es debido a que las cadenas de caracteres se manejan muy parecido, pero no exactamente igual a un arreglo simple de caracteres. La diferencia es que, en una cadena de caracteres, al final de la cadena, el procesador automáticamente coloca un caracter especial conocido como el **caracter nulo** y representado por el símbolo *'\0'*. De esta forma, el procesador puede usar sólo 19 elementos del arreglo para los caracteres y usará el último elemento para el caracter nulo.

Algo que no se ha mencionado, todos los elementos de cualquier arreglo de un programa se almacenan en memoria en forma contigua. Por ejemplo, el arreglo nombre[20] que se declaró arriba se almacenaría en la memoria de la computadora de la siguiente forma:



Esta característica de los arreglos es muy importante cuando se manejan cadenas de caracteres. Así, cuando se usa un arreglo para almacenar una cadena de caracteres, dichos caracteres se almacenan en orden en cada uno de los elementos del arreglos. Por ejemplo, si la cadena es "Hola", esta cadena se almacenaría en un arreglo de 10 elementos de la siguiente forma:

H	o	l	a	\0	?	?	?	?	?

Observe que, luego de los elementos del arreglo que guardan los caracteres de la cadena que se tiene, viene un elemento que almacena al caracter nulo y el resto de los elementos (5) no son utilizados y no almacenan ningún valor. Para que se observe la diferencia, un arreglo simple de caracteres (no una cadena), sería almacenada como:

H	o	l	a	?	?	?	?	?	?

sin utilizar el caracter nulo al final de los caracteres del arreglo.

DECLARACIÓN DE CADENAS DE CARACTERES

Declaración de Arreglos para Almacenar Cadenas de Caracteres

Para declarar un arreglo que sirva para guardar una cadena de caracteres, se utiliza la siguiente sintaxis:

char *nombre_del_arreglo****[****numero_máximo_de_caracteres + 1****];***

Vea que es la misma sintaxis que se usa para declarar cualquier arreglo de caracteres pero, para determinar el tamaño del arreglo, simplemente hay que tomar en cuenta el tamaño máximo posible de la cadena y sumarle 1 (debido a que se necesita el carater nulo al final).

INICIALIZACIÓN DE CADENAS DE CARACTERES

La inicialización de un arreglo que contenga una cadena de caracteres se realiza generalmente al momento de declararlo, como en el caso siguiente:

char *nombre_del_arreglo****[****numero_máximo_de_caracteres + 1****]="****cadena****";***

o bien

char *nombre_del_arreglo****[****numero_máximo_de_caracteres+1****]={"****cadena****"};***

Por ejemplo:

char nombre[10]={"Juan"};
char nombre[10]="Pedro";

Nota importante: Si uno usa la siguiente inicialización (como se hizo anteriormente para arreglos simples):

char nombre[10]={'J', 'u', 'a', 'n'};

El resultado **no es una cadena de caracteres**, pues este tipo de inicialización no añade el caracter nulo al final, por lo que el resultado de dicha inicialización es una *arreglo simple de caracteres*.

FUNCIONES PARA MANIPULAR CADENAS DE CARACTERES

Asignación de Cadenas de Caracteres

Este y el siguiente son temas que debieran de verse como una excepción a los que se ha discutido anteriormente durante el curso. Para asignar un valor a una cadena de caracteres luego de declararlo, no se utiliza el símbolo de igual. Es decir, la asignación:

```
char nombre[20];
nombre = "Juan";
```

es **incorrecta**. Para hacer una asignación es necesario utilizar una función predefinida en el lenguaje. En el caso de C++, esta función es ***strcpy***. La función strcpy es una función sin valor de regreso que recibe dos argumentos. Un argumento es la variable cuyo valor se desea asignar y la otra es la cadena de caracteres que se desea asignar. Por ejemplo. Las siguientes dos asignaciones son correctas y equivalentes:

```
char nombre[20];
strcpy(nombre, "Juan"); /* Llamado a la función strcpy  */
```

o bien

```
char nombre1[20], nombre2[20]="Juan";
strcpy(nombre1, nombre2); /* Llamado a la función strcpy  */
```

Nota importante: Para manipular cadenas de caracteres se requiere de funciones como la función strcpy. Las funciones para manipular cadenas de

caracteres se encuentran definidas en la librería string.h. Por lo tanto, va a ser necesario que incluyamos una nueva librería en nuestros programas con cadenas de caracteres:

#include <string.h>

Comparación de Cadenas de Caracteres

Frecuentemente es necesario comparar cadenas de caracteres entre sí. Otra vez, en cadenas de caracteres se tiene un excepción respecto de lo que se ha visto. Para comparar dos cadenas de caracteres **no se pueden usar los operadores** **==**, **>=** ó **<=**. Para comparar dos cadenas se utiliza una función especial (también definida en string.h) llamada **strcmp**. strcmp recibe dos argumentos, que son las dos cadenas a comparar, y regresa un valor de tipo entero. Por ejemplo, para las mismas declaraciones de nombre1 y nombre2 dadas anteriormente:

x = strcmp(nombre1, nombre2);

es una sentencia correcta si x es de tipo entero. El valor que regresa la función strcmp es 0 si las dos cadenas son iguales. Regresa 1 si la primera cadena es mayor a la segunda y regresa –1 si la segunda cadena es mayor a la primera. Aquí, ser mayor no significa tener más caracteres, sino la comparación se hace considerando el número de código ascii de los caracteres. La comparación se hace uno a uno hasta que se encuentre un caracter diferente entre las dos cadenas.

Otras dos funciones para Cadenas de Caracteres

Existen otras dos funciones (aunque hay mucho más) que son de uso muy común para manipular cadenas de caracteres. Estas son las funciones **strlen** y **strcat**.

La función **strlen** recibe como argumento una cadena y da como valor de regreso un entero que corresponde al número de caracteres de la cadena (sin contar al caracter nulo). Por ejemplo, en el caso siguiente:

```
char nombre[10]=”Juan”;
int x;
x = strlen(nombre);
```

La variable x tendría un valor de 4 luego que se ejecutan las sentencias.

La función **strcat** recibe como argumentos dos cadenas y da como resultado la unión de ambas cadenas en el orden indicado. La segunda cadena se anexa a la primera cadena. Por ejemplo, las siguientes sentencias:

```
char nombre[20]=”Juan ”, apellido[10]=”Razo”;
strcat(nombre,apellido);
```

cambia el valor de la cadena *nombre* de “Juan “ a “Juan Razo”.

Para usar strlen y srtcat también se necesita string.h.

Uso de Cadenas con Funciones y Salidas de Resultados y Entradas de Datos

Debe destacarse que, cuando se trata de cadenas de caracteres, en las funciones strlen, strcat, strcmp y strcpy, se usa únicamente el nombre del arreglo que contiene a las cadenas, no se utiliza su dimensión. Esta es otra excepción a lo que se vió antes. Es decir, se usó, por ejemplo:

```
x = strlen(nombre);
```

y no

```
x = strlen(nombre[20]);
```

Lo mismo debe hacerse en casos como:

cin>>nombre;

y

cout<<nombre;

Arreglos de Variables para Almacenar Cadenas de Caracteres

Hemos visto que, para guardar una cadena que tenga como máximo 19 caracteres se utilizó:

char nombre[20];

¿Que pasaría, sin embargo, si uno estuviera haciendo una base de datos de 10 nombres?. Una opción sería por supuesto usar:

char nombre1[20], nombre2[20], ..., nombre10[20];

Sin embargo, una opción más sencilla y eficiente es usar arreglos multidimensionales. Así, por ejemplo, si

char nombre[20];

se usó para un solo nombre de 19 caracteres (máximo), la siguiente sentencia se puede usar para definir una variable que pueda contener 10 nombres de 19 caracteres:

char nombre[10][20];

Así, cada uno de nombre[0], nombre[1], ..., nombre[9] podría almacenar una cadena de 19 caracteres. Si se usan este tipo de arreglos multidimensionales, tanto las funciones de manipulación de cadenas como cin y cout, se debería usar con el nombre del arreglo seguido de la primera de las dimensiones del arreglo múltiple. Por ejemplo:

cin>>nombre[0];
cout<<nombre[3];
strcmp(nombre[4], nombre[5]);
strcat(nombre[2], nombre[1]);

APLICACIONES

1. Escriba un programa en C++ que reciba como dato una cadena de caracteres en letras minúsculas y la convierta de forma que se obtenga una cadena equivalente en letras mayúsculas.

2. Se dice que una cadena de caracteres es un **anagrama** de otra si ambas consisten sólo de letras y, tratando como iguales las letras mayúsculas y minúsculas, ambas contienen las mismas letras (*no necesariamente en el mismo orden*). Escriba un programa en C++ que reciba como dato a dos cadenas de caracteres y las modifique de forma que se obtengan los anagramas correspondientes. Por ejemplo:

Datos:	"gato"	"raton"
Anagramas:	"gatorn"	"ratong"

SOLUCIONES

```
#include<iostream.h>
#include <string.h>
#include<stdlib.h>

int main()
{
  /* Este programa convierte una cadena de letras minusculas
     a su correspondiente equivalente en letras mayusculas */

  char palabra_min[21], palabra_may[21];
  int len,n;

  /* Dato */
  cout<<"Dame una palabra en minusculas. Maximo 20 caracteres \n";
  cin>>palabra_min;

  /*calculo del numero de caracteres*/
  len = strlen(palabra_min);

  /*Conversion a mayusculas*/
  for(n=1; n<=len; n++)
     palabra_may[n-1] = palabra_min[n-1] - 32;
  /* Convertir arreglo en cadena */
  palabra_may[len] = '\0';

  /* Salida del resultado */
  cout<<"\n";
  cout<<"La palabra equivalente en mayusculas es "<<palabra_may<<"\n";
  cout<<"\n";

  system("PAUSE");
  return 0;
}
```

```
#include<iostream.h>
#include <string.h>
#include<stdlib.h>

int main()
{
  /* Este programa obtiene los anagramas de dos cadenas
     de caracteres */

  char cadena1[21], cadena2[21];
  char anagrama1[41], anagrama2[41];
  char aux[2];
  int len1,len2, n, m, add;

  /* Datos */
  cout<<"Dame una cadena de caracteres (letras, maximo 20) \n";
  cin>>cadena1;
  cout<<"Dame otra cadena de caracteres (letras, maximo 20) \n";
  cin>>cadena2;

  /*calculo del numero de caracteres de las cadenas*/
  len1 = strlen(cadena1);
  len2 = strlen(cadena2);

  /* Asignando contenido inicial */
  strcpy(anagrama1,cadena1);
  strcpy(anagrama2,cadena2);

  /*Obteniendo los anagramas*/
  /* Primero */
  for(n=1; n<=len2;n++)
  {
   add=1;
   for(m=1; m<=len1;m++)
   {
    if((cadena2[n-1]== cadena1[m-1]) || (cadena2[n-1]== (cadena1[m-1]-
32)))
    {
      add =0;
      break;
    }
   }
   if(add==1)
   {
    aux[0] =  cadena2[n-1];
    aux[1] = '\0';
    strcat(anagrama1,aux);
   }
```

```
  }

  /* Segundo */
  for(n=1; n<=len1;n++)
  {
   add=1;
   for(m=1; m<=len2;m++)
   {
    if((cadena1[n-1]== cadena2[m-1]) || (cadena1[n-1]== (cadena2[m-1]-
32)))
    {
      add =0;
      break;
    }
   }
   if(add==1)
   {
    aux[0] =  cadena1[n-1];
    aux[1] = '\0';
    strcat(anagrama2,aux);
   }
  }

  /* Salida del resultado */
  cout<<"\n";
  cout<<"Las palabras son \t"<<cadena1<<"\t\t"<<cadena2<<"\n";
  cout<<"Los anagramas son \t"<<anagrama1<<"\t\t"<<anagrama2<<"\n";
  cout<<"\n";

  system("PAUSE");
  return 0;
}
```

PRÁCTICA DE APLICACIÓN DE CADENAS DE CARACTERES

Los siguientes dos programas ejemplifican el uso y manipulación de cadenas de caracteres.

1) Codifique y ejecute lo siguiente. Omita comentarios en su código si requiere ahorro de tiempo

```
#include <fstream.h>  /* libreria para archivos IO */
#include<iostream.h>
#include <string.h> /* libreria de funciones para cadenas de caracteres*/
#include<stdlib.h>

int main()
{
  /* Este programa lee  las partes de un nombre usando cadenas
     de caracteres y luego las integra y analiza usando funciones de
     manipulación de cadenas. El nombre completo lo guarda en
     un archivo y lo muestra en pantalla */

  char materia[20]={"Matematicas"}; /* Inicializando una cadena*/
  char nombre_p[20], apellido_pat[20], apellido_mat[20]; /* Otras cadenas */
  char nombre[60]; /* Una ultima cadena */
  double calificacion;

  ofstream   csalida;  /* Instruccion similar a cout */

  csalida.open("lista.dat"); /* Este es el archivo de salida, lista.dat */

  /* Ahora se piden cadenas de caracteres, no solo un caracter */
  cout<<"Dame tu nombre propio. Si tienes mas de uno, dame el primero \n";
  cin>>nombre_p;
  cout<<"Dame apellido paterno. Solo dame una palabra si tiene muchas\n";
  cin>>apellido_pat;
  cout<<"Dame apellido paterno. Solo dame una palabra si tiene muchas\n";
  cin>>apellido_mat;
  cout<<"Dame tu calificacion final de "<< materia <<"\n";
  cin>>calificacion;
    /*Uso de funciones: formacion del nombre completo*/
  strcpy(nombre, nombre_p); /* copia la cadena nombre_p a nombre */
  strcat(nombre, " ");  /* Añade un espacio a nombre */
  strcat(nombre, apellido_pat);  /* Añade apellido_pat a nombre */
  strcat(nombre, " ");  /* Añade un espacio a nombre*/
  strcat(nombre, apellido_mat);  /* Añade apellido_mat a nombre*/
```

```
  /*  Salida de la informacion a pantalla y a archivo */
  cout<<"\n";
  cout<<materia<<"\n";    /*A pantalla*/
  cout<<nombre<<"\t"<<calificacion<<"\n";
  csalida<<materia<<"\n";   /* A archivo */
  csalida<<nombre<<"\t"<<calificacion<<"\n";

  /* Cierra el archivo lista.dat*/
  csalida.close();

  cout<<"\n";
  system("PAUSE");
  return 0;
}
```

2) Use el explorador de windows para buscar el archivo lista.dat

Este archivo deberá contener la misma información que salió a pantalla en el programa anterior.

3) Escriba y ejecute el siguiente programa. Use nombres propios de una sola palabra

```
#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

int main()
{
  char nombre[2][20]; /* Observe: Arreglo de cadenas de caracteres */

  cout<<"Dame un nombre propio \n";
  cin>>nombre[0];
  cout<<"Dame otro nombre propio \n";
  cin>>nombre[1];

  /*Se muestra ahora en pantalla cada nombre observe que se usa
    un solo indice */
  cout<<"\n";
  cout<<"El primer nombre es "<<nombre[0]<<"\n";
  cout<<"El segundo nombre es "<<nombre[1]<<"\n";
  cout<<"\n";
```

```
  /*Uso de funciones: comparacion de los dos nombres anteriores*/

  if( ! strcmp(nombre[0],nombre[1]) )
     cout<<”Los dos nombres anteriores son iguales \n”;
  else
     cout<<”Los dos nombres anteriores son diferentes \n”;

   cout<<"\n";
   system("PAUSE");
   return 0;
}
```

UNIDAD IV

TEMA II

TIPOS DE DATOS ABSTRACTOS

TIPOS DE DATOS ABSTRACTOS (TDA)

Un TDA es un tipo de dato definido por el programador que se puede manipular de un modo similar a los tipos de datos definidos por el lenguaje. En pocas palabras, si los tipos de datos existentes en el lenguaje no son suficientes o no son eficientes para ciertas aplicaciones, la mayoría de los lenguajes de programación permiten al usuario definir sus propios tipos de datos. Esta definición consiste en establecer los elementos de que consta el tipo así como las operaciones que se pueden realizar con instancias de este tipo.

DECLARACIÓN DE TDA'S

Para construir un tipo abstracto se debe:

1) Establecer la definición del tipo
2) Definir también las operaciones (funciones y procedimientos) que pueden operar con dicho tipo
3) Ocultar la presentación de los elementos del tipo de modo que sólo se puede trabajar con ellos usando los procedimientos definidos en 2)
4) Poder crear instancias múltiples del tipo

Un TDA es el elemento básico de la abstracción de datos. Debe verse como una caja negra, pues la representación y la implementación deben permanecer "ocultas", de forma que para trabajar con los elementos de un TDA el único mecanismo permitido es el de usar las operaciones definidas para dicho TDA.
La mayoría de las aplicaciones de interés de los TDA implican el uso del concepto de apuntadores (referencia), que no son parte de los alcances de este curso. Así, las operaciones y definiciones que se verán a continuación son las más elementales posibles, sólo para ejemplificar la definición y el uso de un TDA.

La declaración de TDA's requiere al menos el uso de dos palabras reservadas en C++. La primera de ellas es la palabra reservada ***struct***. struct define una

estructura de datos; se dice que struct permite crear un tipo de datos que está compuesto de uno o varios elementos denominados campos. Los campos, a su vez, pueden ser variables de tipos definidos por el lenguajes u otros TDA's. Considere el siguiente ejemplo:

```
struct Tipo_Persona
{
  int   edad;
  double  altura;
  double peso;
  char  nombre[25];
};
```

En tal ejemplo, se esta creando una estructura de datos que definirá un nuevo tipo. Este nuevo tipo se denomina Tipo_Persona. Observe que los elementos del nuevo tipo son cuatro, un número entero, un caracter y dos números dobles; todos ellos representando características particulares del Tipo_Persona. En general, se esperaría desarrollar "operaciones" sobre este nuevo tipo que permitieran modificar cada uno de estos 4 campos o elementos.

Una vez definida esta estructura, para ejemplificar y mostrar el grado de abstracción que puede lograrse, se describe ahora la segunda de las palabras reservadas útiles en la declaración de TDA's. Esta segunda palabra reservada es ***typedef***. typedef se utiliza para definir el "alias" o sinónimo de un tipo de datos. Observe el siguiente ejemplo:

```
typedef struct Tipo_Persona Persona;
```

Lo que esta sentencia lograría es que se está definiendo un nuevo tipo de datos llamado Persona, que contiene exactamente la misma estructura y elementos que la estructura Tipo_Persona definida arriba (por ello se dice que, al haber usado typedef, ***struct Tipo_Persona*** y ***Persona*** son sinónimos). En otras palabras, con las definiciones anteriores existiría un nuevo tipo en el lenguaje

denominado Persona, que podrá utilizarse como cualquier otro tipo. Por ejemplo, si se tiene:

Persona Gabriel, Antonio;

Se estarán creando dos nuevas variables, Gabriel y Antonio, del tipo Persona. De esta forma, cada una de estas variables (Gabriel y Antonio) contiene todos los elementos definidos en la estructura Tipo_Persona. Obsérvese aquí, que la estructura que contienen estas dos nuevas variables queda "*oculta*". Por ello se habla de que un TDA es el elemento básico de la abstracción de datos.

La sintaxis de las dos palabras claves incluidas aquí es la siguiente:

```
struct identificador_1
{
  nombre_del_Tipo elemento_1;
  nombre_del_Tipo elemento_2;
  …
  nombre_del_Tipo elemento_n;
};
```

```
typedef struct identificador_1 sinonimo;
```

OPERACIONES SOBRE UN TDA

Las operaciones que se aplican a un TDA (numeral 2) generalmente caen dentro de las siguientes tipos básicos:

1) Construcción: Crean una nueva instancia del tipo
2) Transformación: Cambian el valor de uno o más elementos del tipo

3) Observación: Permiten determinar el valor de uno o más elementos de un tipo sin modificarlos
4) Iteradores: Permiten procesar todo los elementos de un TDA en forma secuencial

Aunque aquí no se analizarán de la creación de cada uno de estos tipos de operaciones, concluimos esta sección mostrando la forma en que cada uno de los elementos de una estructura de datos puede ser accedido. Observe los siguientes ejemplos:

```
Gabriel.edad = 25;
Antonio.altura = 1.76;
```

Observe que luego del nombre de las estructuras, el acceso a los elementos de dichas estructuras se realiza usando ***un punto y el nombre del elemento*** al que se quiere acceder. Logrado este acceso, los elementos de estas estructuras se pueden manipular como cualquier otra variable de su mismo tipo.

APLICACIONES

Las dos aplicaciones tradicionales de los TDA es la definición de una lista o de una pila, ambos conceptos muy útiles en el área de las estructuras de datos:

1) Una pila es una colección dinámica de datos de un mismo tipo, en la que los elementos se insertan y se extraen por un mismo extremo.
2) Una lista es una también una colección dinámica de datos de un mismo tipo, pero en este caso el acceso a cada uno de los elementos es por posición; se considera aquí que cada elemento de la lista tiene un único predecesor (excepto el primer elemento) y un único sucesor (excepto el último elemento).

Nuevamente, dada la necesidad de usar apuntadores, no entraremos en detalles con estas aplicaciones.