Informática II

Manual de asignatura

Sistema de Universidades Tecnológicas

Electricidad y Electrónica Industrial

Programa 2004

Créditos

Elaboró: Ing. Ricardo Simón Rangel

Revisó: Ing. Emiliano Mendoza Álvarez

Colaboradores:

Autorizó:

Contenido

Objetivo general

Desarrollar programas utilizando un lenguaje de programación

Tipo de aprendizaje por desarrollar

Conocimiento y habilidades

	Horas		
	Práctica	**Teoría**	**Total**
I Sistemas Numéricos	**2**	**4**	**6**
II Algoritmos	**8**	**2**	**10**
III Metodología de la programación	**2**	**3**	**5**
IV Lenguajes de programación	**31**	**8**	**39**
V Canales de comunicación	**10**	**5**	**15**

I

SISTEMAS NUMERICOS

Objetivo particular de la unidad

Conocer diferentes sistemas numéricos

Tipos de aprendizaje por desarrollar

Conocimiento

I.1 SISTEMAS NUMERICOS

Saber en la teoría (4 hrs.)

1.1 INTRODUCCIÓN:

Las computadoras digitales han hecho posible muchos avances científicos, industriales y comerciales que de otra manera nunca se hubieran alcanzado. Las computadoras se usan en cálculos científicos, en el procesamiento de información comercial y de negocios, control de tránsito aéreo, vía espacial, campo educativo y muchas otras áreas. La propiedad más sorprendente de una computadora digital es su generalidad. Puede seguir una secuencia de instrucciones, denominada programa, que opera según la información dada. El usuario puede especificar y cambiar los programas y/o la información de acuerdo con la necesidad especifica. A causa de esta flexibilidad, las computadoras digitales de propósito general pueden realizar una amplia variedad de tareas de procesamiento de la información.

La computadora digital de propósito general es el mejor ejemplo conocido de un sistema digital. Es característico de un sistema digital la manipulación de elementos discretos de información. Los elementos discretos de información se representan en un sistema digital mediante cantidades físicas denominadas señales. Las señales eléctricas como voltajes y corrientes son las más comunes. Las señales en todos los sistemas digitales en la actualidad tienen solo dos valores discretos y se dice que son binarios. El diseñador de un sistema digital esta limitado al uso de señales binarias debido a la confiabilidad más baja de los circuitos electrónicos múltiples. En otras palabras puede diseñarse un circuito con diez estados, que

usa un valor discreto de voltaje para cada estado, pero tendría una confiabilidad muy baja de operación. En contraste, un circuito de transistor que, ya sea este encendido o apagado, tiene dos valores de señal posible puede construirse para que sea en extremo confiable. Debido a esta restricción física de los componentes, y debido a que la lógica humana tiende a ser binaria, los sistemas digitales que están limitados a tomar valores discretos están, restringidos aún más a tener valores binarios.

1.2 NUMEROS BINARIOS

Un número decimal como 7392 representa una cantidad igual a 7 millares, más 3 centenas, más 9 decenas, más 2 unidades. Los millares, centenas, etc., son potencias de 10 implicadas por la posición de los coeficientes. Para ser más exactos, 7392 debe escribirse como:

$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$

Sin embargo, la convención es escribir solo los coeficientes y de sus posiciones se deduce las potencias necesarias de 10. En general un número con un punto decimal se representa por una serie de coeficientes como sigue:

$$a_5 . a_4 a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3}$$

Los coeficientes a son uno de los diez dígitos (0, 1, 2,..., 9), y el valor del subíndice j da el valor del lugar y, por tanto, la potencia de 10 por lo cual debe multiplicarse el coeficiente.

$$10^5 a_5 + 10^4 a_4 + 10^3 a_3 + 10^2 a_2 + 10^1 a_1 + 10^0 a_0 + 10^{-1} a_{-1} + 10^{-2} a_{-2} + 10^{-3} a_{-3}$$

El sistema de números decimales se dice que es la base, o raíz, 10 debido a que usa diez dígitos y los coeficientes se multiplican por potencias de 10. El sistema binario es un sistema diferente de números. Los coeficientes de los números del sistema binario tienen dos valores posibles: 0 y 1. Cada coeficiente a se multiplica por 2. Por ejemplo, el equivalente decimal del numero binario 11010.11 es igual a 26.75, como se muestra por la multiplicación de los coeficientes por potencias de 2. :

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 26.75$$

En general, un número expresado en un sistema base r tiene coeficientes multiplicados por la potencia de r:

$$a_n . r^n + a_{n-1} . r^{n-1} + \ldots + a_2 . r^2 + a_1 . r + a_0 + a_{-1} . r^{-1} + a_{-2} . r^{-2}$$

$$\ldots\ldots + a_{-m} . r^{-m}$$

Los coeficientes a varían en valor desde 0 a r-1. Para distinguir entre números de bases diferentes, se encierra entre paréntesis los coeficientes y se escribe un subíndice igual

ala base usada (excepto algunas veces para números decimales, donde el contenido indica obviamente que es decimal). Un ejemplo de un numero con base 5 es:

$$(4021.2)_5 = 4x5^3 + 0x5^2 + 2x5^1 + 1x5^0 + 2x5^{-1} = (511.4)_{10}$$

Obsérvese que los valores de los coeficientes con base 5 pueden ser solo 0, 1, 2, 3, y 4.

Es costumbre tomar los r dígitos necesarios para los coeficientes del sistema decimal cuando la base del número es menor que 10. Se usan las letras del alfabeto para completar los diez dígitos decimales cuando la base del número es mayor que 10. Por ejemplo, en el sistema hexadecimal (base 16), los primeros diez dígitos 10, 11, 12, 13, 14 y 15, respectivamente. Un ejemplo de un número hexadecimal es:

$$(B65F)_{16} = 11x16^3 + 6x16^2 + 5x16^1 + 15x16^0 = (46687)_{10}$$

Los primeros 16 números en los sistemas decimal, binario, octal y hexadecimal se listan en la tabla 1-1.

1.3 CONVERSIÓN DE LA BASE DE NUMEROS

Un número binario puede convertirse en decimal formando la suma de las potencias de dos de los coeficientes cuyo valor es 1. Por ejemplo:

$$= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = (10.375)_{10}$$

Tabla 1-1 Números con bases diferentes

Decimal (base 10)	Binario (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

El número binario tiene cuatro números 1 y el equivalente decimal se encuentra por adición de cuatro potencias de 2. En forma similar, un número que se expresa en la base r puede convertirse en su equivalente decimal multiplicando cada coeficiente por la potencia correspondiente de r y sumando. El siguiente es un ejemplo de la conversión de octal en decimal:

$(630.4)_8 = 6 \times 8^2 + 3 \times 8^1 + 0 \times 8^0 = (408.5)_{10}$

La conversión de decimal en binario o cualquier otro sistema con base r es más conveniente si el número se separa en una parte entera y en una parte fraccional, y la conversión de cada parte se hace por separado. La conversión de un entero de decimal en binario, se explica de manera más adecuada con un ejemplo.

Saber hacer en la práctica (2 hrs.)

1.4 Realizar conversiones entre bases (binario, octal y hexadecimal)

EJEMPLO 1.1: Convierta el decimal 41 en binario: Primero el 41 se divide entre 2 para dar un cociente entero de 20 y un residuo de 1 / 2. El cociente se divide otra vez entre 2 para dar un nuevo cociente y un residuo. Este proceso sé continua hasta que el entero cociente llega a ser cero. Los coeficientes del número binario se obtienen por los residuos como sigue:

Entero	Residuo
41	
20	1
10	0
5	0
2	1
1	0
0	1

Respuesta = $(101001)_2$

Ejemplo 1-2: Convierta el número decimal 153 en octal. La base requerida es 8. Primero 153 se divide entre 8 para dar un cociente entero de 19 y un residuo de 1. Entonces 19 se divide entre 8 para dar un cociente entero de 2 y un residuo de 3. Por último, 2 se divide entre 8 para dar un cociente de 0 y un residuo de 2. Este proceso puede manipularse en forma conveniente como sigue:

Entero	Residuo
1	1
3	3
2	2

Respuesta = $(231)_8$

La conversión de un a fracción decimal en binario se lleva a cabo por un método similar al que se utiliza para los enteros. Sin embargo, se usa la multiplicación en lugar de la división, y los enteros se acumulan en lugar de residuos. De nuevo el método se explica de manera más adecuada con un ejemplo.

Ejemplo 1-3: Convierta él numero decimal (0.6875) en número binario. Primero 0.6875 se multiplica por 2 para dar un entero y una fracción. La nueva fracción se multiplica por 2 para dar un entero y una nueva fracción. Este proceso sé continua hasta que la fracción llega a ser cero o hasta que el número de dígitos tiene suficiente exactitud. Los coeficientes del número binario se obtienen mediante los enteros como sigue:

	Entero		Fracción	Coeficiente
0.6875 x 2 =	1	+	0.375	1
0.3750 x 2 =	0	+	0.7500	0
0.7500 x 2 =	1	+	0.5000	1
0.5000 x 2 =	1	+	0.0000	1

Respuesta = $(0.1011)_2$

Para convertir una fracción decimal en un número expresado en base r, se usa un procedimiento similar. La multiplicación se hace por r en lugar de 2, y los coeficientes se encuentran por los enteros que pueden variar el valor desde 0 a (r – 1) en lugar de 0 y 1.

Ejemplo 1-4: Convierta el número decimal (0.513) en número octal, base 8.

0.513x 8 = 4.104
0.104 x 8 = 0.832
0.832 x 8 = 6.656
0.656 x 8 = 5.248
0.248 x 8 = 1.984
0.984 x 8 = 7.872

La respuesta, a siete cifras significativas, se obtiene de la parte entera de los productos:

$(0.513)_{10} = (0.406517...)_{8}$

La conversión de números decimales tanto en las partes enteras como fraccionarias se hace por la conversión de la parte entera y fraccionaria por separado y, entonces combinando las dos respuestas. Por eso el uso de los resultados de los ejemplos 1-1 y 1-3; se obtiene:

$(41.6875)_{10} = (101001.1011)_{2}$

A partir de los ejemplos 1-2 y 1-4, se obtiene:

$(153.513)_{10} = (231.406517)_{8}$

1.5 NUMEROS OCTALES Y HEXADECIMALES

La conversión de binario, octal y hexadecimal - y a la inversa – juega una parte importante en las computadoras digitales. Ya que 2 = 8 y 2 = 16, cada digito octal corresponde a tres dígitos binarios y cada digito hexadecimal corresponde a cuatro dígitos binarios. La conversión de binario en octal se lleva a cabo fácilmente por la partición del número binario en grupos de tres dígitos cada uno, principiando desde el punto binario y procediendo a la izquierda y a la derecha. Entonces el digito octal correspondiente se asigna a cada grupo. El siguiente ejemplo ilustra el procedimiento:

$(\ 1\ 0\ \ 110\ \ 001\ \ 101\ \ 011\ \ .\ \ 111\ \ 100\ \ 000\ \ 110\ ) = (26153.7406)_{8}$
$\quad 2\quad 6\quad\ 1\quad\ 5\quad 3\qquad\ 7\quad\ 4\quad\ 0\quad\ 6$

La conversión de binario en hexadecimal es similar, excepto que el número binario se divide en grupos de cuatro dígitos.

$(\ 10\ \ \ 1100\ \ \ 0110\ \ \ 1011\ \ .\ \ 1111\ \ \ 0010\ ) = (2C6B.F2)_{16}$
$\quad 2\quad\ C\quad\ \ 6\quad\ \ B\qquad\ F\quad\ \ 2$

El digito correspondiente hexadecimal (u octal) para cada grupo de dígitos binarios se recuerda con facilidad después de estudiar los valores listados en la Tabla 1-1.

La conversión de octal o hexadecimal en binario se realiza por un procedimiento inverso al anterior. Cada digito octal se convierte en su equivalente binario de tres dígitos. De manera semejante, cada digito hexadecimal se convierte en su equivalente binario de cuatro dígitos. Esto se ilustra en los siguientes ejemplos:

(673.124) = (110 111 011 . 001 010 100) $_2$
6 7 3 1 2 4

(306.D) = (0011 0000 0110 . 1101) $_2$
3 0 6 D

Los números binarios son difíciles de trabajar ya que requieren tres o cuatro veces más dígitos que su equivalente decimal. Por ejemplo, el número binario 111111111111 es equivalente al decimal 4095. No obstante las computadoras digitales utilizan números binarios y algunas veces es necesario que el operador humano o usuario se comunique en forma directa con la maquina mediante números binarios. Un esquema que retiene el sistema binario en la computadora, pero que reduce el número de dígitos que el humano debe considerar, emplea la relación entre el sistema de números binarios y el sistema octal o hexadecimal.

1.6 COMPLEMENTOS

Los complementos se usan en las computadoras digitales para simplificar la operación de sustracción y para manipulaciones lógicas. Hay dos tipos de complementos para cada sistema base r. (1) el complemento de r y (2) el complemento de (r-1). Cuando el valor de la base se sustituye, los dos tipos reciben los nombres de 2 y 1 para complementos de números binarios o, de 10 y 9 para complementos de números decimales

1.6.1 El complemento de "r"

Dado un número positivo N en base r con una parte entera de n dígitos, el complemento de r de N se define como $r^n - N$ para $N \neq 0$ y 0 para N = 0. el siguiente ejemplo numérico ayudará a aclarar la definición:

El complemento de 10 $(52\ 520)_{10}$ es $10^5 - 52\ 520 = 47480$

El número de dígitos en el número es n = 5

El complemento de 10 de $(0.3267)_{10}$ es $1 - 0.3267 = 0.6733$

No hay parte entera, de modo que $10^n = 10^0 = 1$

El complemento de 10 de $(25.639)_{10}$ es $10^2 - 25.639 = 74.361$

El complemento de 2 de (101100) es $(2^6)_{10} - (101100) = (1000000 - 101100) = 010100$

El complemento de 2 de (0.0110) es $(1 - 0.0110) = 0.1010$

Por la definición y los ejemplos, es claro que el complemento de 10 de un número decimal puede formarse dejando todos los ceros significativos sin cambios, se resta el primer dígito de cero menos significativo de 10 y, entonces se restan todos los otros dígitos más significativos de 9. El complemento de 2 puede formarse dejando todos los ceros menos significativos y el primer dígito no-cero sin cambio y, entonces, se reemplaza los 1 por 0 y los 0 por el 1 en todos los otros dígitos más significativos. Un tercer método más simple para obtener el complemento de r se da después de la definición del complemento (r – 1).

El complemento de r de un número existente para cualquier base r (r mayor que pero no igual a 1) y puede obtenerse por la definición que se presentó antes. Los ejemplos listados aquí usan números con r = 10 (decimal) y r = 2 (binario), debido a que estas son las dos bases de mayor interés para nosotros. El nombre del complemento se relaciona con la base del número que se usa. Por ejemplo, el complemento de (r – 1) de un número en la base 11 se denomina complemento de 10, ya que r -1 = 10 para r = 11.

1.6.2 El complemento de (r – 1)

Dado un número positivo N en base r con una parte entera de n dígitos y una parte fraccionaria de m dígitos, el complemento de (r – 1) de N se define como $r^n - r^{-m} - N$.
A continuación se presentan algunos ejemplos numéricos.

El complemento de 9 de $(52\ 520)_{10}$ es $(10^5 - 1 - 52\ 520) = 99\ 999 - 52\ 520 = 47\ 479$

No hay parte fraccionaria de modo que $10^{-m} = 10^0 = 1$

El complemento de 9 de $(0.3267)_{10}$ es $(1 - 10^{-4} - 0.3267) = 0.9999 - 0.3267 = 0.6732$

No hay parte entera, de modo que $10^n = 10^0 = 1$

El complemento de 9 de $(25.639)_{10}$ es $(10^2 - 10^{-3} - 25.739) = 99.999 - 25.639 = 74.360$

El complemento de 1 de (101100)2 es (2 – 1) – (101100) = (111111 – 101100) = 010011

El complemento de 1 de (0.0110) es $(1 - 2^{-4})_{10}$ – (0.0110) – (0.111 -0.0110) = 0.1001

Por los ejemplos, puede observarse que el complemento de 9 de un número decimal se forma simplemente al restar cada dígito de 9. el complemento de 1 de un número binario es aun más simple de formar: los 1 se cambian en 0 y los 0 en 1. ya que el complemento de (r – 1) se obtiene de manera muy fácil, algunas veces es conveniente usarlo cuando se desea el complemento de r. por las definiciones y mediante una comparación de los resultados que se obtuvieron en los ejemplos, se concluye que el complemento de r puede obtenerse del complemento (r – 1) después de la adición al dígito menos significativo. Por ejemplo, el complemento de 2 de 10110100 se obtiene del complemento de 1 de 01001011 por la adición de 1 para obtener 01001100.

Vale la pena mencionar que el complemento del complemento restablece el número a su valor original. El complemento r de N es $(r^n - N)$ y el complemento de $(r^n - N)$ es: $r^n - (r^n - N) = N$; y en forma similar para el complemento 1.

1.6.3 Sustracción con complementos de r

El método directo de sustracción que se enseña en las escuelas elementales usa el concepto de préstamo. En este método, se presta un 1 de una posición significativa más alta cuando el dígito minuendo es menor que el dígito sustraendo correspondiente. Esto parece ser más fácil cuando las personas realizan la resta con lápiz y papel. Cuando la resta se implanta mediante componentes digitales, se encuentra que este método es menos eficiente que el método que utiliza complementos y suma como se establece más adelante.

La sustracción de números positivos (M – N), ambos en base r, puede hacerse como sigue:

agregue el minuendo M al complemento de r des sustraendo N.

inspecciónese el resultado que se obtuvo en el paso 1 para un “acarreo final”:
si ocurre un “acarreo final” descártese
si no ocurre un “acarreo final”, tómese el complemento r del número que se obtuvo en el paso 1 y colóquese un signo negativo enfrente.

Los siguientes ejemplos ilustran el procedimiento:

Ejemplo 1.5: usando el complemento de 10, reste 72 532 – 3 250

M = 72532		72532
N = 03250	+	
10 complementos de N = 96 750		96750
	1	69282

Respuesta: 69282

Ejemplo 1.6: Reste $(3\ 250 - 72\ 532)_{10}$

M = 03250		03250
N = 72532	+	
10 complementos de N = 27468		57468
	No acarreo	30718

Respuesta: - 69 282 = - (10 complementos de 30 618)

Ejemplo 1.7: utilice el complemento de 2 para realizar M – N con los números binarios dados.
(a)

M = 1010100
N = 1000100
complemento de 2 de N = 0111100

```
  1010100
+
  0111100
acarreo final 1 0010000
```

Respuesta: 10 000

(b)

M = 1000100
N = 1010100
complemento de N = 0101100

```
  1000100
+
  0101100
no acarreo    1110000
```

Repuesta: - 10 000 = - (2 complemento de 1110000)

La prueba del procedimiento es: la adición de M al complemento r de N da ($M+r^n - N$). Para números que tienen una parte de n dígitos, r^n es igual a 1 en la posición (n + 1) ava (lo que se ha denominado (acarreo final"). Ya que se supone que tanto M como N son positivos, entonces:

$(M + r^n - N) > r^n$ si $M > N$, ó

$(M + r^n - N) < r^n$ si $M < N$

En el caso (a) la respuesta es positiva e igual a M – N, la cual se obtiene en forma directa descartando y se lleva cuenta final r^n. En el caso (b), la respuesta es negativa e igual a: – (N – M), este caso se detecta con la ausencia de un acarreo final. La respuesta se obtiene tomando un segundo complemento y añadiendo un signo negativo.

$-\left(r^n - (M + r - N)\right) = -(N - M)$

1.6.4 Sustracción con el complemento (r – 1)

El procedimiento para la resta con el complemento de (r – 1) es exactamente el mismo que se usó con el complemento de r, excepto por una variación, llamada “acarreo final desplazado”, como se muestra adelante. La resta de M – N, con ambos números positivos en la base r, puede calcularse en la siguiente forma:

Agréguese el minuendo M al complemento de (r – 1) del sustraendo N.
Inspecciónese el resultado que se obtuvo en el paso 1 para un acarreo final,
Si ocurre un acarreo final, agréguese 1 al dígito menos significativo (acarreo final desplazado).

Si no ocurre un acarreo final, tómese el complemento de (r – 1) del número obtenido en el paso 1 y colóquese al frente un signo negativo.

La prueba de este procedimiento es muy semejante a la dada para el caso de complementos de r y se deja como ejercicio. Los ejemplos siguientes ilustran el procedimiento.

Ejemplo 1.8: Repita los ejemplos 1-5 y 1-6 usando complementos de 9.

(a)

M = 72532
N = 03250
9 complemento de N = 96749

```
  72532
+
  96749
1 69281
+
→ 1
  69282
```

acarreo final y se pasa a la derecha

Respuesta: 69282

(b)

M = 03250
N = 72532
9 complemento de N = 27467

```
  03250
+
  27467
  30717
```

no acarreo

Respuesta: -69282 = - (9 complemento de 30717)

EJEMPLO 1-9: Repita el ejemplo 1-7 utilizando complementos de 1

(a)

M = 1010100
N = 1000100
1 complemento de N = 0111100

```
                    1010100
                  +
                    0111011
acarreo fin   1     0001111
y se pasa a la derecha
                  +       1
                    0010000
```

Respuesta: 10000

(b)

M = 1000100
N = 1010100
1 complemento de N = 0101011

```
                    1000100
                  +
                    0101011
no acarreo    _     1101111
```

Repuesta: - 10000 = - (1 complemento de 1101111)

ALGORITMOS

Objetivo particular de la unidad

Utilizar algoritmos para la solución de un problema

Tipo de aprendizaje por desarrollar

Conocimientos y habilidades

II.1 ALGORITMOS

Saber en la teoría (2 hrs.)

2.1 CONCEPTO DE ALGORITMO

El objetivo fundamental de este manual es enseñar a resolver problemas mediante una computadora. Un programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo este libro nos referiremos a la metodología necesaria para resolver problemas mediante programas, concepto que se denomina **metodología de la programación.** El eje central de esta metodología es el concepto, ya tratado, de algoritmo.
La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

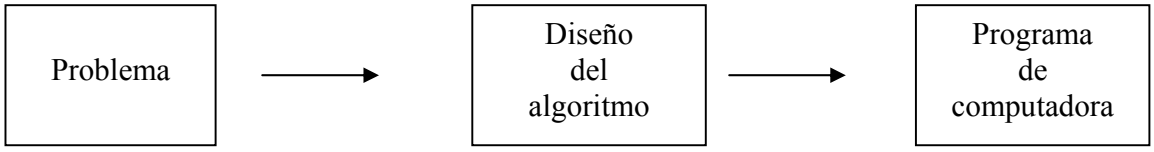


Figura: Resolución de un problema.

Los pasos para la resolución de un problema son:

l. Diseño del algoritmo que describe la secuencia ordenada de pasos -sin ambigüedades- que conducen a la solución de un problema dado. (Análisis del problema y desarrollo del algoritmo.)
2. Expresar el algoritmo como un programa en un lenguaje de programación adecuado. (Fase de codificación.
3. Ejecución y validación del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.
Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.
En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.
Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será el diseño de algoritmos. A la enseñanza y práctica de esta tarea se dedica gran parte de este libro.
El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, la solución de un problema se puede expresar mediante un algoritmo.

2.2 CARACTERÍSTICAS DE LOS ALGORITMOS

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener él mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: Entrada, Proceso y Salida. En el algoritmo de receta de cocina citada anteriormente se tendrá:

Entrada: ingredientes y utensilios empleados.
Proceso: elaboración de la receta en la cocina.
Salida: terminación del plato (por ejemplo, cordero).

Ejemplo 2.1

Un cliente ejecuta un pedido a una fábrica. La fábrica examina en su banco de datos la ficha del cliente, si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario, rechazará el pedido. Redactar el algoritmo correspondiente.

Los pasos del algoritmo son:

1. Inicio.
2. Leer el pedido.
3. Examinar la ficha del cliente.
4. Si el cliente es solvente, aceptar pedido; en caso contrario, rechazar pedido.
5. Fin.

Ejemplo 2.2

Se desea diseñar un algoritmo para saber si un número es primo o no.

Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6,4, 12, 16,20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc.

El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etcétera.

1. Inicio.

2. Poner X igual a 2 (X = 2, X, variable que representa a los divisores del numero que se busca N).

3. Dividir N por X (N/X).

4. Si el resultado de N/X es entero, entonces N no es un numero primo y bifurcar al punto 7; en caso contrario, continuar el proceso.

5. Suma 1 a X (X f- X + 1).

6. Si X es igual a NI, entonces N es un numero primo; en caso contrario, bifurcar al punto 3.

7. Fin.

Por ejemplo, si N es131, los pasos anteriores serían:

1. Inicio.
2. x = 2.
3. y a. 131/x. Como el resultado no es entero, sé continua el proceso.
5. x ← 2+1,luego x =3.
6. Como x no es 131, se bifurca al punto 3.
3. y 4. 131/x resultado no es entero.
5. x ← 3 + 1, x = 4.
6. Como x no es 131 bifurca al punto 3.
3. y 4. 131/x..., etc.
7. Fin.

Ejemplo 2.3

Realizar la suma de todos los números pares entre 2 y 1000.

El problema consiste en sumar 2 + 4 + 6 + 8 + ... + 1000.
Utilizaremos las palabras SUMA y NUMERO *(*variables, serán denominadas más tarde) para representar las sumas sucesivas (2+4), (2+4+6), (2+4+6+8), etc.

La solución se puede escribir con el siguiente algoritmo:

l. Inicio.
2. Establecer SUMA a 0.
3. Establecer NUMERO a 2.
4. Sumar NUMERO a SUMA. El resultado será el nuevo valor de la suma (SUMA).
5. Incrementar NUMERO en 2 unidades.
6. Si NUMERO =< 1000 bifurcar al paso 4; en caso contrario, escribir él ultimo valor de SUMA y terminar el proceso.
7. Fin.

2.3 LOS LENGUAJES DE PROGRAMACIÓN

Como se ha visto en el apartado anterior, para que un procesador realice un proceso se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de *interpretar* el algoritmo, lo que significa

- comprender las instrucciones de cada paso,
- realizar las operaciones correspondientes.

Cuando el procesador es una computadora, el algoritmo se ha de expresar en un formato que se denomina programa. Un programa se escribe en un lenguaje de programación y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman programación. Así pues, los len guajes utilizados para escribir programas de computadoras son los lenguajes de programación y programadores son los escritores y diseñadores de programas.
Los principales tipos de lenguajes utilizados en la actualidad son tres:

- lenguaje máquina,
- lenguaje de bajo nivel (ensamblador),.
- lenguajes de alto nivel

2.4 TRADUCTORES DE LENGUAJE

Los traductores de lenguaje son programas que traducen a su vez los programas fuente escritos en lenguajes de alto nivel a código máquina.

Los traductores se dividen en:

- compilado res,
- intérpretes.

2.5 INTÉRPRETES

Un intérprete es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta. Los programas intérpretes clásicos como BASIC, prácticamente ya no se utilizan, aunque las versiones QBasic y QuickBASIC se comercializan todavía con el Sistema Operativo DOS que corre en las computadoras personales. Sin embargo, está muy extendida la versión interpretada del lenguaje Smalltalk, un lenguaje orientado o objetos puro.

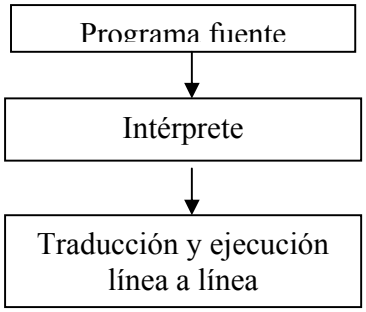


Figura 2.4. Intérprete

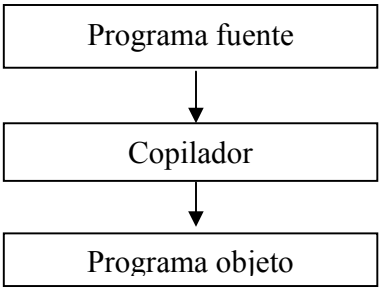


Figura 2.5. La compilación de programas.

2.6 COMPILADORES

Un compilador es un programa que traduce los programas fuente escritos en lenguajes de alto nivel-Pascal, FORTRAN, a lenguaje máquina.
Los programas escritos en lenguajes de alto nivel se llaman programas fuente y el programa traducido programa objeto o código objeto. El compilador traduce - sentencia a sentencia - el programa fuente.
Los lenguajes compiladores típicos son: C, CH, PASCAL, FORTRAN, COBOL.

2.7 LA COMPILACIÓN Y SUS FASES

La compilación es el proceso de traducción de programas fuente a programas objeto. El programa objeto obtenido de la compilación ha sido traducido normalmente a código máquina.
Para conseguir el programa máquina real se debe utilizar un programa llamado montador o en lazador (linker). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable (figura 1.8).

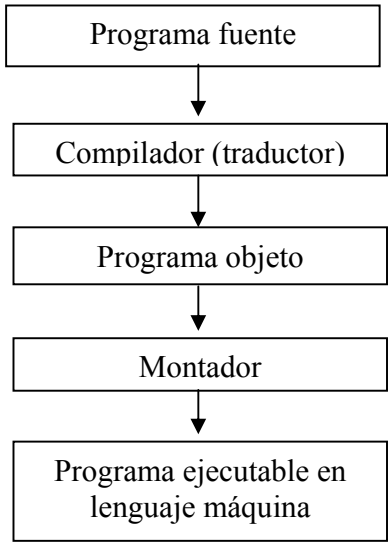


Figura 2.6. Fases de la compilación.

El proceso de ejecución de un programa Pascal, por ejemplo, tiene los siguientes pasos :

1. Escritura del programa fuente con un editor (programa que permite a una computadora actuar
2. de modo similar a una máquina de escribir electrónica y guardado en un dispositivo de almacenamiento (por ejemplo, un disco).
3. Introducir el programa fuente en memoria.
4. Compilar el programa con el compilador Pascal.
5. Verificar y corregir errores de compilación (listado de errores).
6. Obtención del. programa objeto.
7. El montador obtiene el programa ejecutable.
8. Se ejecuta el programa y, si no existen errores, se tendrá la salida del programA

El proceso de ejecución sería el mostrado en las Figuras 1.9 y 1.10.

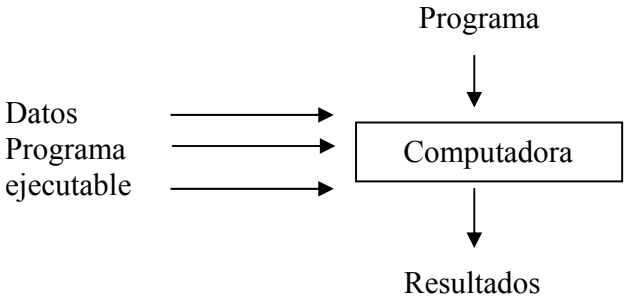


Figura 2.7. Ejecución de un programa.

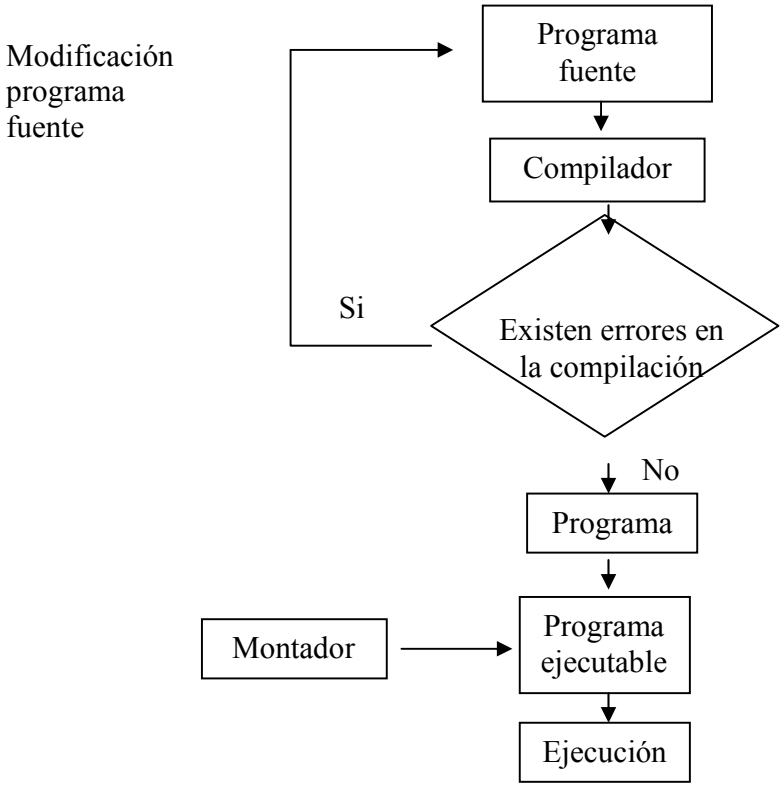


Figura 2.8. Fases de la ejecución de un programa.

2.8 REPRESENTACION GRAFICA DE LOS ALGORITMOS

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación, elegido. Ello permitirá que un algoritmo ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que la descripción pueda servir fácilmente para su transformación en un programa, es decir, su codificación.

Los métodos usuales para representar un algoritmo son:

1. Diagrama de Flujo
2. Diagrama N-S (Nassi-Schneiderman)
3. Lenguaje de especificación de algoritmos: Pseudo código
4. Lenguaje español
5. fórmulas

Los métodos 4 y 5 no suelen ser fáciles de transformar en programas. Una descripción en español narrativo no satisfactoria, ya que es prolija y generalmente ambigua. Una fórmula, sin embargo, es buen sistema de representación. Por ejemplo, la fórmula para la solución de una ecuación cuadrática en un medio sucinto el procedimiento algorítmico que se debe ejecutar para obtener las raíces.

$X1 = (- b + \sqrt{b^2 – 4ac}) / 2^a$ $x2 = (-b - \sqrt{b^2 – 4ac}) / 2^a$

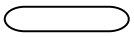
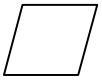
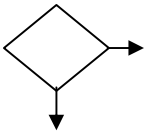
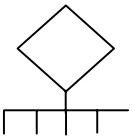
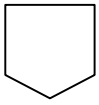
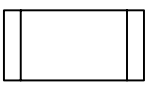
Significa:

1. Eleve al cuadrado b
2. Toma a; multiplicar por c; multiplicar por 4
3. Restar el resultado de 2 del resultado de 1, etc.

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

2.9 DIAGRAMAS DE FLUJO

Un diagrama de Flujo (flowchart) es una de las técnicas de representación de algoritmos más antigua ya la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la Figura 2.6.y que tiene los pasa del algoritmo escritos es esas cajas unidas por flechas, denominadas líneas de flujo, que indica la secuencia en que se deben ejecutar.
La Figura 2.7. En un Diagrama de Flujo básico.
El diagrama citado (Figura 2.7.) Representa la resolución de un programa que deduce el salario neto de un trabajador a partir de la lectura del nombre, horas, trabajadas, precio de la hora, y sabiendo que los impuestos aplicados son el 25 por 100 sobre el salario bruto.
Los símbolos estándar normalizados por ANSI (abreviatura de American National Standard institute) son muy variados.

Símbolos principales	Función
	Terminal (representa el comienzo, “Inicio” y el final “fin”, de una programa. Puede representar también una parada o interrupción que sea necesario realizar en un programa.
	Entrada / salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, “entrada”, o registro de la información procesada en un periférico, “salida”).
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información procesada en un periférico, “Salida”).
	Decisión 8indica operaciones lógicas o de comparación entre datos – normalmente dos – y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas – respuestas Si o No -. Pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación de se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector (sirve para enlazar dos partes cualesquiera de un organigrama a través de un conector en la salida y otro conector en la entrada. Se refiera a la conexión en la misma página del diagrama).
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector conexión (entre dos puntos del organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina en un módulo independiente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).

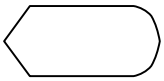
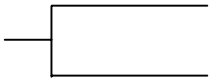
Símbolos secundarios	Función
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolo del diagrama de flujo. Se puede dibujar cualquier lado del símbolo).

Figura 2.9. Símbolos de diagrama de Flujo

En la figura 2.9 Se representa una plantilla de dibujo típica donde se contemplan la mayoría de los símbolos en el diagrama, sin embargo, los símbolos más utilizados representa:

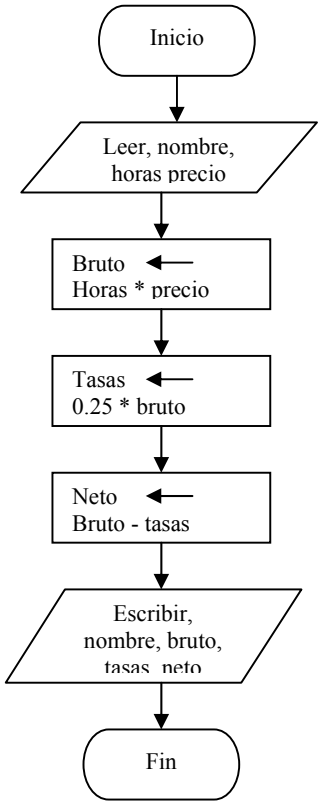


Figura 2.10. Diagrama de flujo

- Proceso
- Decisión
- Conectores
- Fin
- Entrada/salida
- Dirección del flujo

Y se resumen en la figura 2.9

En un Diagrama de Flujo:

- Existe una caja etiqueta “inicio”, que es de tipo elíptico
- Existe otra caja etiqueta “fin” de igual forma que la anterior
- Si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo (el resto de la figuras se utilizan sólo diagramas de flujo generales o de detalle y no siempre son imprescindibles).

Se puede escribir más de un paso del algoritmo en una sola caja rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujos complicados e ininteligibles.

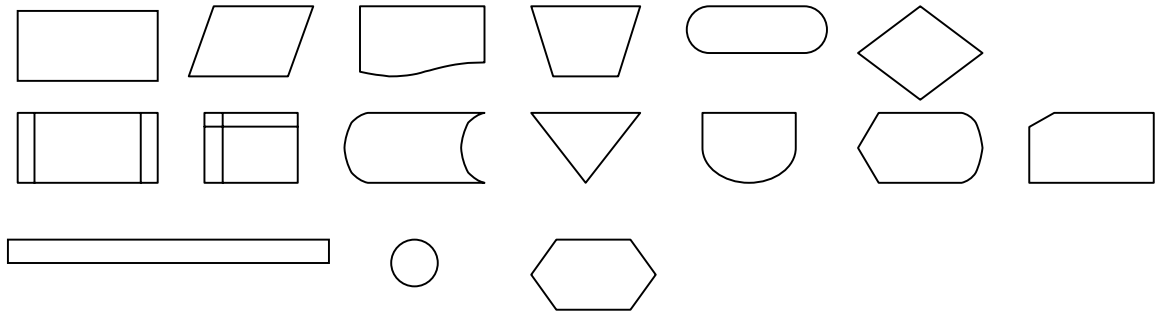


Figura 2.11. Plantilla para dibujo de diagrama de Flujo

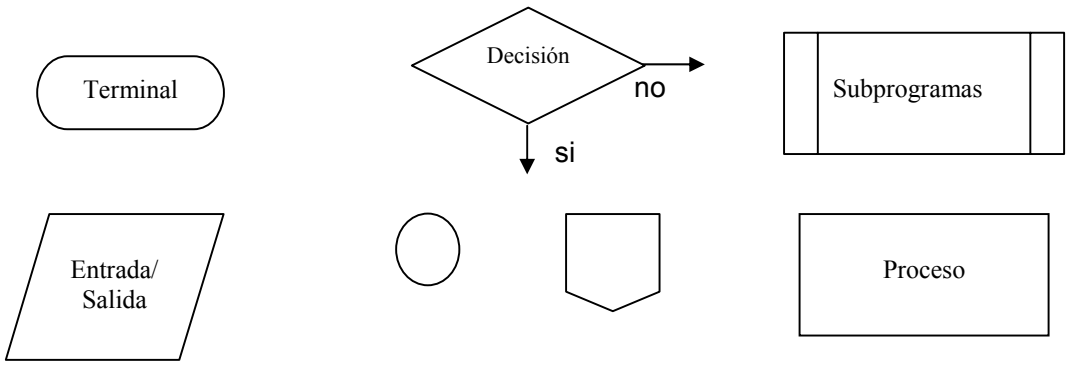


Figura 2.12. Símbolos más utilizados en los diagramas de flujo

Símbolos de Diagramas de Flujo

Cada símbolo visto anteriormente indica el tipo de operación a ejecutar y el diagrama de flujo ilustra gráficamente la secuencia en la que se ejecutan las operaciones.
Las líneas de flujo representan el flujo secuencial de la lógica del programa.
Un rectángulo significa algún tipo de proceso en la computadora, es decir, acciones a realizar (sumar dos números, calcular la raíz cuadrada de un número, etc.).
El paralelogramo es un símbolo de entrada / salida que representa cualquier tipo de entrada o salida desde el programa o sistema; por ejemplo, entrada de teclado, salida en impresión o pantalla, etc.
El símbolo rombo es una caja de decisión que representa respuestas si / no o bien diferentes alternativas 1, 2, 3, 4,…, etc.
Cada diagrama de flujo comienza y termina con un símbolo terminal.
Un pequeño círculo es un conector y se utiliza para conectar caminos, tras otras roturas previas del flujo del algoritmo.
Otros símbolos de diagramas d flujo menos utilizados de mayor detalle que los anteriores son:

Un trapezoide indica que un proceso manual se va a ejecutar en contraste con el rectángulo, que indica proceso automático.
El símbolo general de entrada / salida se puede subdividir en otros símbolos: teclado, pantalla, impresora, disco magnético, disquete o disco flexible, casete.

Saber hacer en la práctica (8 hrs.)

Planteamiento y aplicación de un algoritmo como el primer paso en la solución de un problema. Utilizar la representación gráfica en la solución de problemas. Aplicar el pseudo código para plantear la solución de problemas.

Ejemplo 2.4.

Calcular la medida de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero – como entrada – indicara que se ha alcanzado el final de serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una medir se necesita sumar y contar los valores .
Por consiguiente, nuestro algoritmo en forma descriptiva seria:

1.- Inicializar contador de números C y variable suma S.
2.- Leer número.
3.- Si el número leído es cero:
* Calcular la media;
* Imprimir la media
* Fin del proceso.

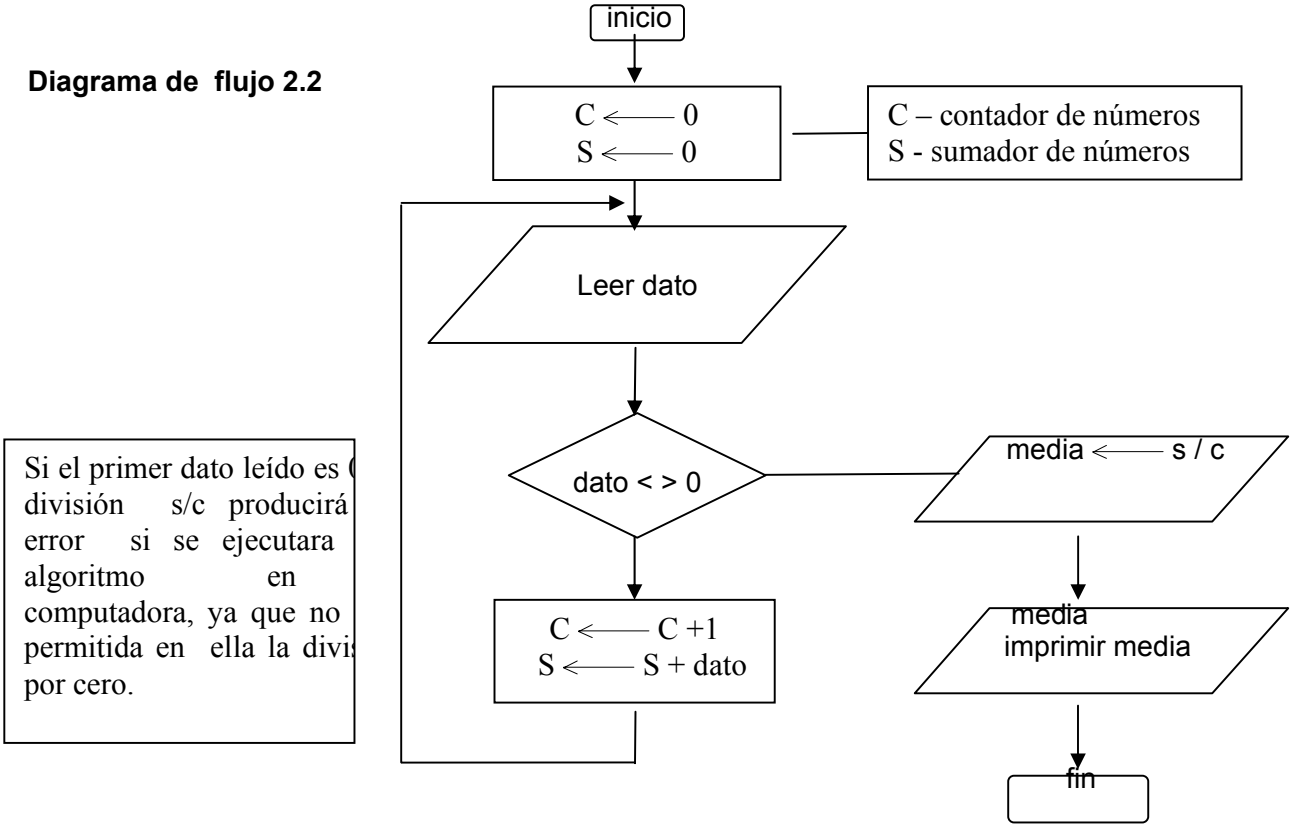
Si el número leído no es cero:

* Calcular la suma;
* Incrementar en uno el contador del número;
* Ir al paso 2.

4.- Fin.

El refinamiento del algoritmo conduce a los pasos sucesivos necesarios para realizar las operaciones de lectura de datos, verificación del último dato, suma y media de los datos.

Diagrama de flujo 2.2

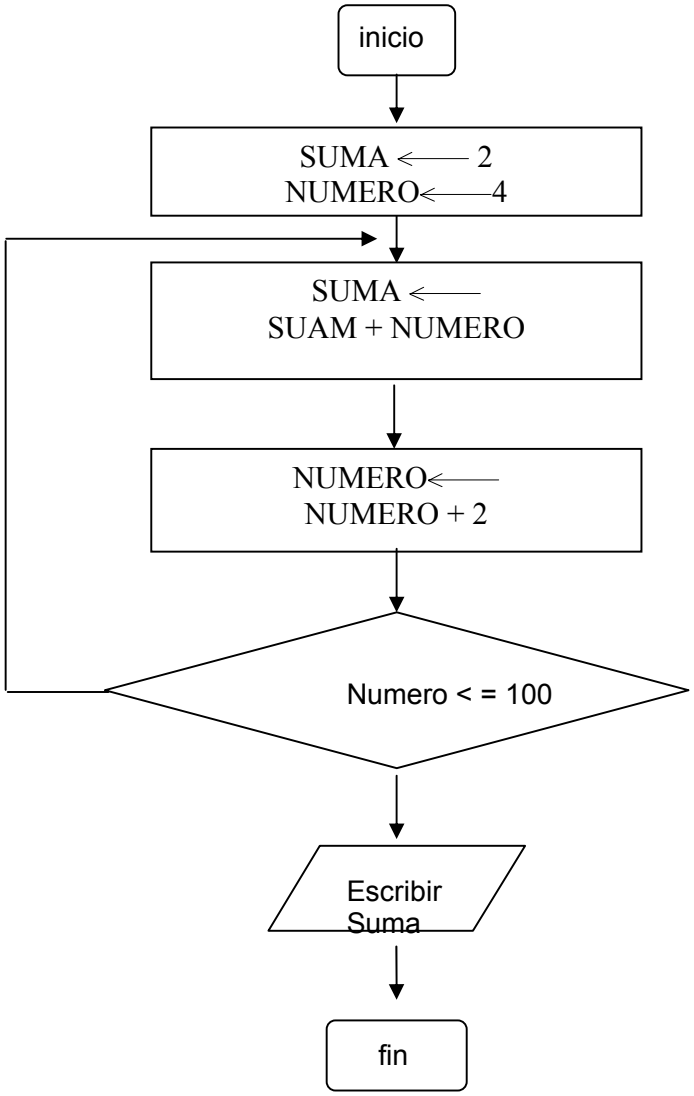


Si el primer dato leído es 0, la división S / C produciría un error si se ejecuta el algoritmo en una computadora, ya que no esta permitida en ella la división por cero.

Ejemplo 2.5

Suma de los números pares comprendidos entre 2 y 100.

Diagrama de flujo 2.3



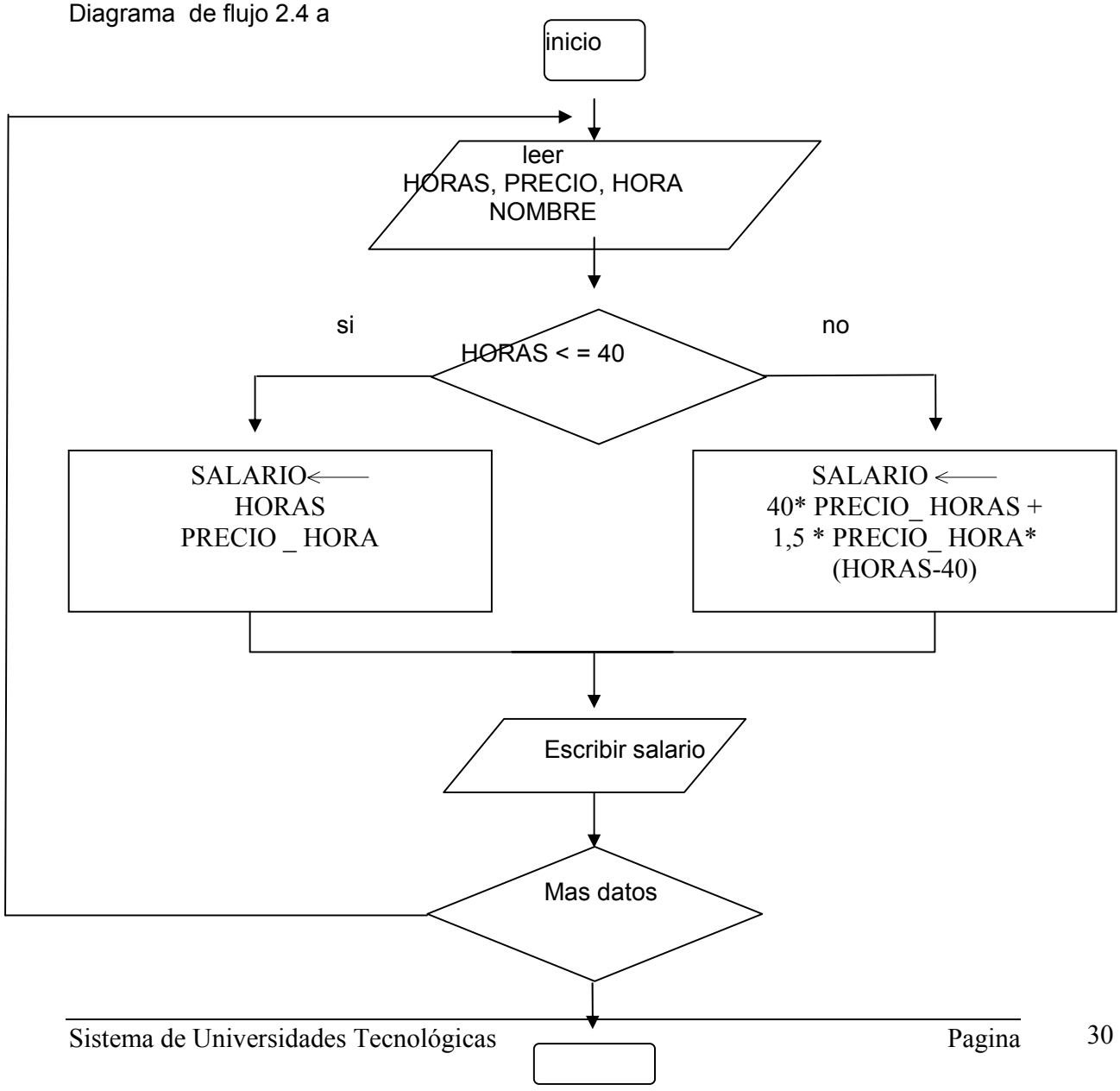
Ejemplo 2.6.
Se desea realizar el algoritmo que resuelva el siguiente problema:

Cálculo de los salarios mensuales de los empleados de una empresa , sabiendo que estos se calculan en base a las horas semanales trabajadas y de acuerdo a un precio especificado por hora . si se pasan de cuarenta horas semanales las horas extraordinarias se pagaran a razón de 1,5 veces la hora ordinaria.
Los cálculos son sencillos:

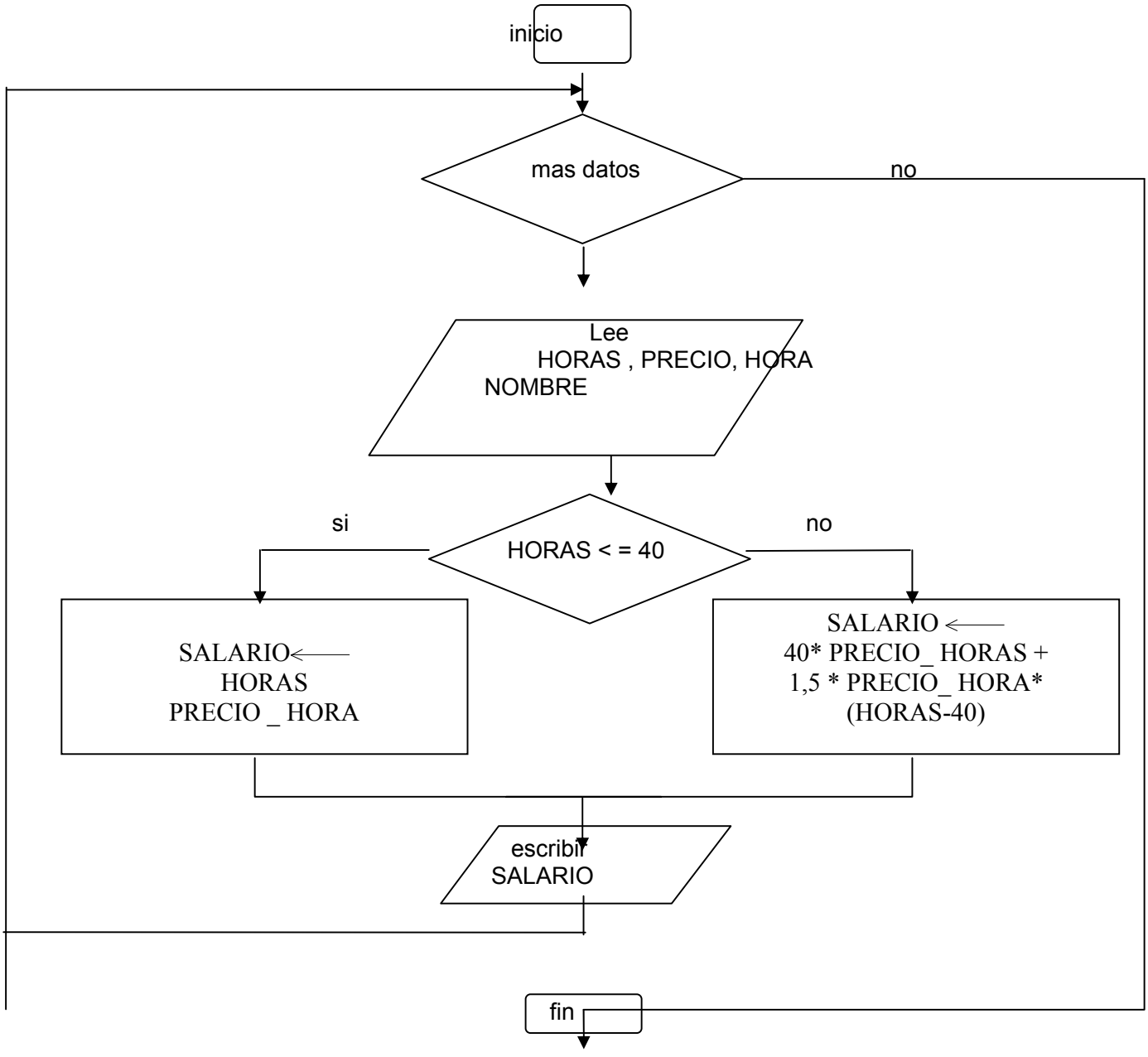
1.- Leer datos del archivo de la empresa, hasta que se encuentre la ficha final del archivo (HORAS , PRECIO _HORA , NOMBRE).
2.- Si HORA<= 40, entonces SALARIO es el producto de horas por PRECIO_ HORA.
3.- Si HORA>40, entonces SALARIO es la suma de 40 veces PRECIO _ HORA mas 1,5 veces PRECIO _ HORA por (HORAS -40).

El diagrama de flujo completo del algoritmo se indica a continuación:

Diagrama de flujo 2.4 a



O bien:
Diagrama de flujo 2.4 b



2.10 PSEUDOCODIGO

El pseudo código es un lenguaje de especificación (descripción) de algoritmos. El uso de lenguajes hace el paso de codificación final (esto es, la traducción a un lenguaje de programación relativamente fácil). Los lenguajes APL Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

El pseudo código nació como un lenguaje similar al ingles y era un medio de representar básicamente las estructuras de un control de programación estructurada que se vera en capítulos posteriores. Se considera un primer borrador. dado que el pseudo código no puede ser ejecutado por una computadora. La ventaja del pseudo código es que en su uso. En la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudo código si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que esta codificado en un lenguaje de programa. Otra ventaja de pseudo código es que puede ser traducido fácilmente a un lenguaje estructurado como pascal, C, FORTRAN 77/90, Ada 83/95, C++, etc.

El pseudo código original utiliza para presentar las acciones sucesivas palabras reservadas en ingles – similares a sus homónimas en los lenguajes de programación -, tales como **start, end, stop, if - then - else, while - end, repeat - until**, etc. La estructura de pseudo código exige normalmente la identacion (sangría en el margen izquierdo) de diferentes líneas.

La representación en pseudo código del diagrama de flujo de la figura 2.7 es la siguiente:

```
start
    // Calculo de impuestos y salario.
    read  nombre, horas, precio-hora
    Salario_ bruto    ←horas * precio por hora
    Tasas ←0,25 * salario- bruto
    Salario _ neto   ←salario bruto _ tasas
    Write nombre, salario- bruto, tasas, salario-neto
end
```

El algoritmo comienza en la palabra **start** y finaliza en la palabra **end**, e n ingles (en español, **inicio y fin**). Entre estas palabras, solo se escribe una instrucción o acción por línea.

La línea precedida por // se denomina comentario. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, solo tiene efecto de documentación interna del programa.
Algunos autores suele utilizar llaves o corchetes.
No es recomendable el uso de apóstrofos o simples comillas como representan en Basic de Microsoft los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o FORTRAN, y daría lugar a confusión.
Otro ejemplo aclaratorio en el uso de pseudo código podría ser un sencillo algoritmo del arranque matinal de un coche.

Inicio

// arranque matinal de un coche.
Introducir la llave de contacto
Tirar del estrangulador del aire
Girar la llave del contacto
Pisar el acelerador
Esperar unos instantes a que se caliente el motor
Llevar al estrangulador de aire a su posición

Fin

Por fortuna, aunque el pseudo código nació como un sustituto del lenguaje de programación y, por consiguiente , sus palabras reservadas se conservaron o fueron muy similares a las de dichos lenguajes, prácticamente el ingles , el uso de pseudo códigos se ha extraído a la hispana con términos en español como ***inicio , fin parada ,leer , escribir , si-entonces-si_no, mientras , fin _ mientras , repetir, hasta _ que,*** etc. Sin duda , el uso del terminología del pseudo código en español ha facilitado y facilitara considerablemente el aprendizaje y el uso diario de programación . En esta obra , al igual que en otras nuestras utilizaremos el pseudo código en español y daremos en su momento la estructuras equivalentes en ingles , al objeto de facilitar la traducción del pseudo código citados anteriormente deberían de ser situaciones de palabras ***start, end , read , wirte , por inicio, fin , leer, escribir*** , respectivamente.

Inicio	**start**	**leer**	**read**
.			
.			
.			
.			
fin	**end**	**escribir**	**write**

actividades de programación resueltas

2.7.- Se desea obtener el salario neto de un trabajador conocido el numero de horas trabajadas, el salario de horas y la tasa de impuestos que se debe deducir .
el algoritmo general es:

1.- Obtener el valor de las horas trabajadas , salario _ horas y tasas.
2.- Calcular salario _ bruto, total de impuestos y salario neto.
3.- Visualizar salario _ bruto, total de impuestos y salario neto.

Las entradas de algoritmos son:

Horas trabajadas , salario _ horas , tasas.

Las salidas de algoritmos son :
Paga bruta, total de impuestos y paga neta.

El refinamiento del algoritmo en pasos de nivel inferior es:

1.- Obtener valores de horas trabajadas , salario bruto y tasas .

2.- Calcular salario bruto , total de impuesto y paga neta.

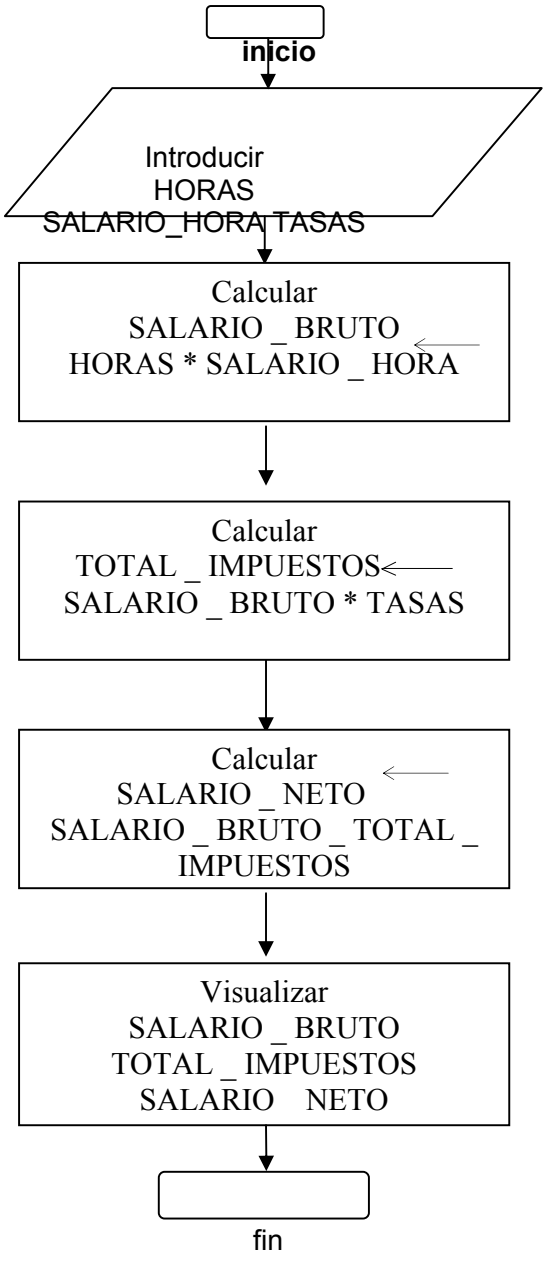
2.1. calcular salario bruto multiplicando las horas trabajadas por el salario horas.

2 .2. Calcular el total de impuestos multiplicando salario bruto por tasas (tanto porcinito de impuesto).

2 . 3 . Calcular el salario neto restando el total de impuestos de la paga bruta.

3.- visualizar salario bruto , total de impuestos , salario neto.

El diagrama de la figura de flujo 2.7 representa este algoritmo



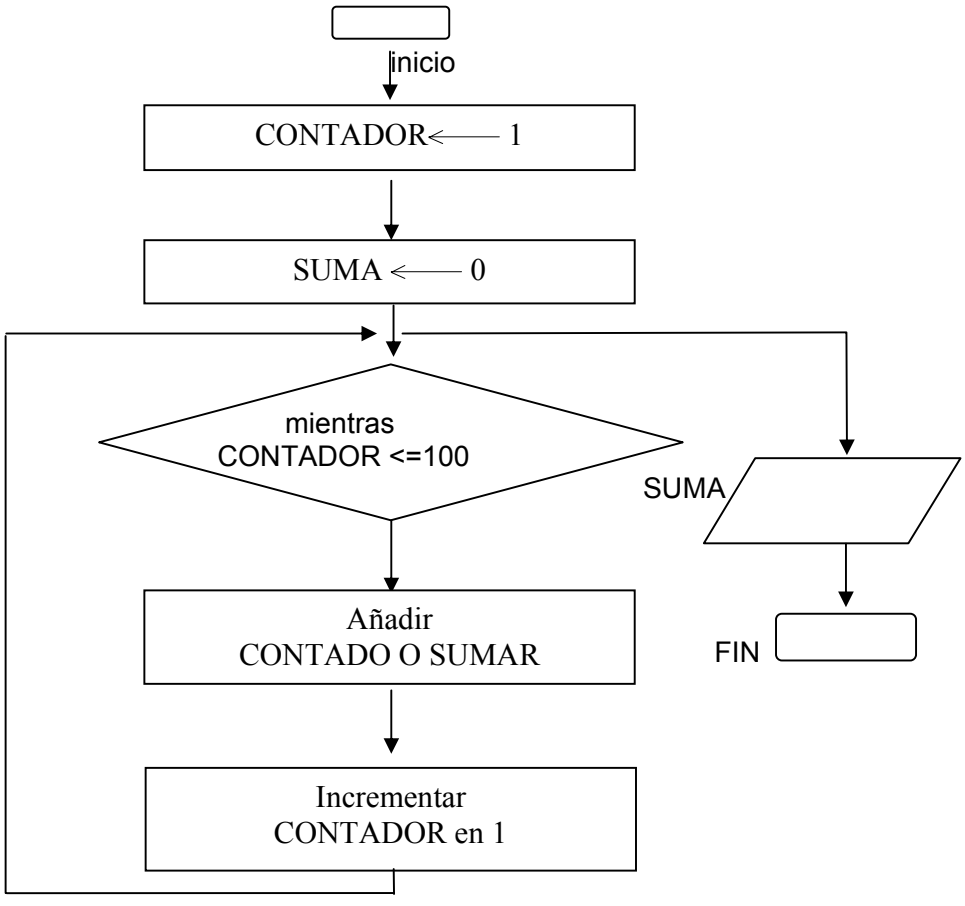
2.8 Calcular el valor de la suma 1+2+2+4+....+100.

variables *CONTADOR* (números sucesivos a contar 1, 2, 3,.....).
SUMA (totalizador de la sumas).

Pseudo código

1.- inicio.
2.- establecer contador a 1.
3.- Establecer a suma a 0.
4.- **Mientras** CONTADOR < = 100 **hacer** lo siguiente:
    4.1.- sumar contador.
    4.2.- incrementar CONTADOR en 1.
    Fin _ mientras
5.- visualizar SUMA.
6.- fin.

Este algoritmo se representa por el diagrama de flujo 2.8.



III

METODOLOGIA DE LA PROGRAMACION

Objetivo particular

Conocer el ciclo de desarrollo de un programa

Tipo de aprendizaje por desarrollar

Conocimiento y habilidades

III.1 METODOLOGIA DE LA PROGRAMACION

Saber en la teoría (3 hrs.)

3.1 ELEMENTOS BASICOS DE UN PROGRAMA

En programación se debe separar la diferencia entre el diseño del algoritmo y su implementación en un lenguaje específico. Por ello, se debe distinguir claramente entre los conceptos de programación y el medio en que ellos se implementan en un lenguaje especifico. Sin embargo, una vez que se comprendan los conceptos de programación, como utilizarlos, la enseñanza de un nuevo lenguaje es relativamente fácil.

Los lenguajes de programación—como los restantes lenguajes—tienen elementos básicos que se utilizan como bloques contractivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan *sintaxis* del lenguaje. Solamente las instrucciones sintáctica mente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la maquina.

Los elementos básicos constitutivos de un programa o algoritmo son:

- *Palabras reservadas (***inició, fin, si-entonces….,** *etc.),*
- *identificadores* (nombres de variables esencialmente),
- *Caracteres especiales* (coma, apostrofo, etc.),
- *Constantes,*
- *Variables,*
- *Expresiones,*
- *Instrucciones*.

Además de estos elementos básicos, existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para el correcto diseño de un algoritmo y naturalmente la codificación del programa.

Estos elementos son:

- *Bucles,*
- *Contadores,*
- *Acumuladores,*
- *Interruptores,*
- *Estructuras:*
  ---secuénciales,
  ---selectivas,
  ---repetitivas.

El amplio conocimiento de todos los elementos de programación y el modo de su integración en los programas constituyen las técnicas de programación que todo buen programador debe conocer.

3.2. Bucles

Un *bucle* o *lazo* (*loop*)1 es un segmento de un algoritmo o programa, cuyas instrucciones se repiten un numero determinado de veces mientras se cumple una determinada *condición* (existe o es verdadera la condición). Se debe establecer un mecanismo para determinar las tareas repetitivas. Este mecanismo es una condición que se puede ser verdadera o falsa y se comprueba una vez a cada *paso* o *iteración* del bucle (total de instrucciones que se repiten en el bucle).

Un bucle consta de tres partes:

- *Decisión,*
- *Cuerpo del bucle,*
- *Salida del bucle.*

El bucle de la figura 3.1 es infinito, ya que las instrucciones (1), (2) y (3) se ejecutan indefinidamente, pues no existe salida del bucle, al no cumplirse una determinada condición.

Si tras la lectura de la variable N se coloca una condición, el bucle dejara de ser infinito y tendrá fin cuando la condición sea verdadera.

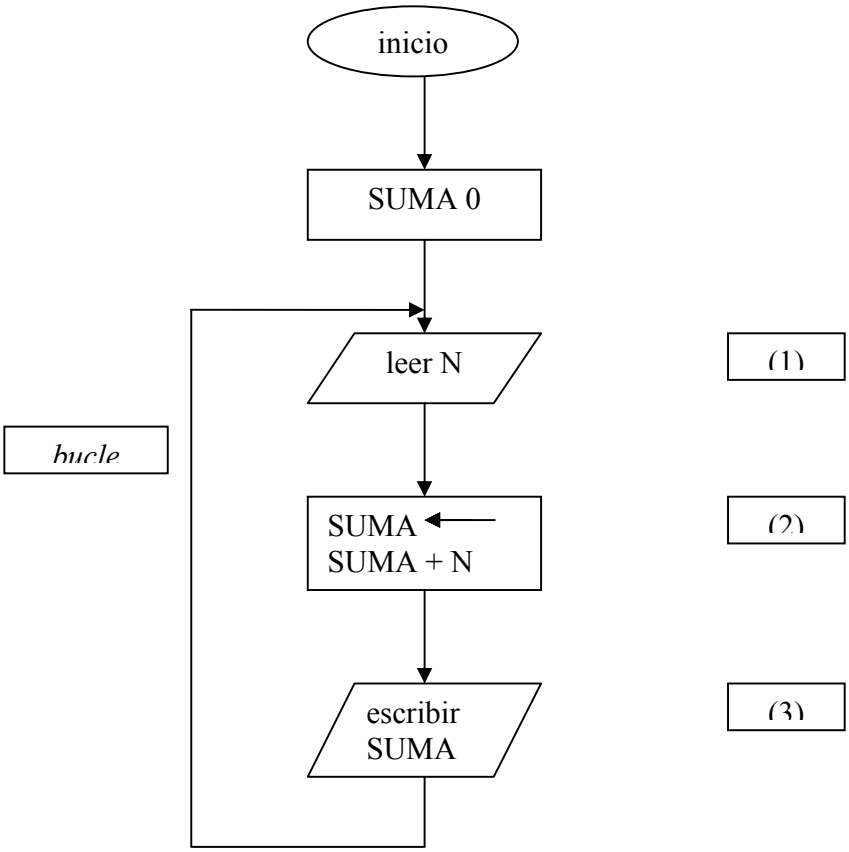


Figura 3.1. Bucle infinito**.**

El diagrama de flujo escrito en ***pseudocódigo*** es:

```
inicio
   SUMA ←  0
    1: leer (N)
    si N = 0 entonces
       escribir (SUMA)
       ir_a fin
    si_no
       SUMA ← SUMA  + N
    Fin_si
     Ir a 1
fin
```

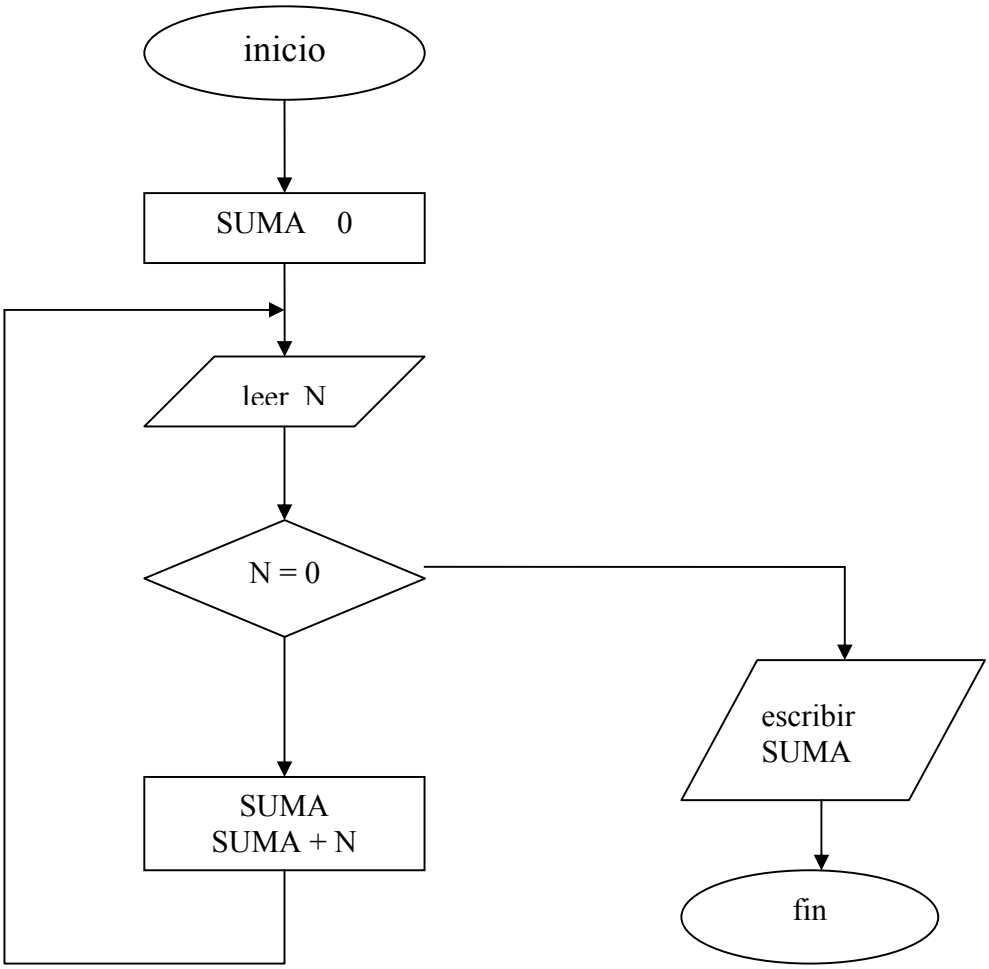


Figura 3.2. Bucle con fin.

3.3 Bucles anidados

En un algoritmo pueden existir varios bucles. Los bucles pueden ser *anidados* o independientes.

Los bucles son ***anidados*** cuando están dispuestos de tal modo que unos son inferiores a otros;

los bucles son *independientes* cuando son externos unos a otros (Figura 3.3).

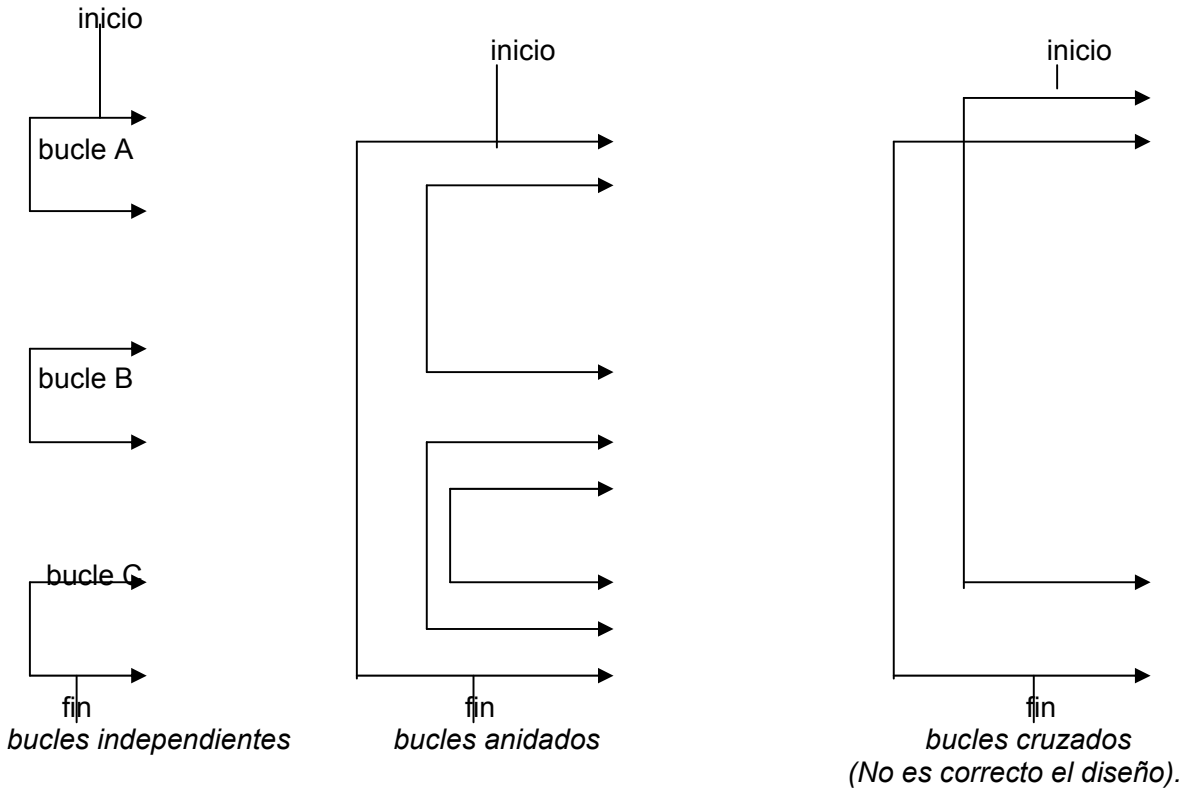


Figura 3.3. Tipos de bucles.

3.3. Contadores

Los procesos repetitivos son la base del uso de las computadoras. En estos procesos se necesitan normalmente contar los sucesos o acciones internas del bucle, como pueden ser los elementos de un fichero, él numero de iteraciones a realizar por el bucle, etc., Una forma de controlar un bucle es mediante un contador.

Un *contador* es una variable cuyo valor se incrementa o decrementa en una cantidad constante en cada iteración.

La Figura 3.6 presenta un diagrama de flujo para un algoritmo que se desea repetir 50 veces; el contador se representa en este ejemplo con la variable CONT. La instrucción que representa a un contador es la asignación CONT CONT + 1.

La Figura 3.7 es otro ejemplo de un diagrama de flujo con contador; en este caso, negativo. Se dice también *descontar.*

El contador puede ser positivo (incrementos, uno en uno) o negativo (decrementos, uno en uno). En la Figura 3.6 el contador cuenta desde 1 a 50 y deja de contar cuando la variable CONT toma el valor de 51 y se termina el bucle. En la Figura 3.7 el contador cuenta negativamente, o lo que es lo mismo, descuenta o decrementa; comienza a contar un *n* y se va decrementando hasta llegar a cero, en cuyo caso se termina el bucle y se realiza la acción escribir.

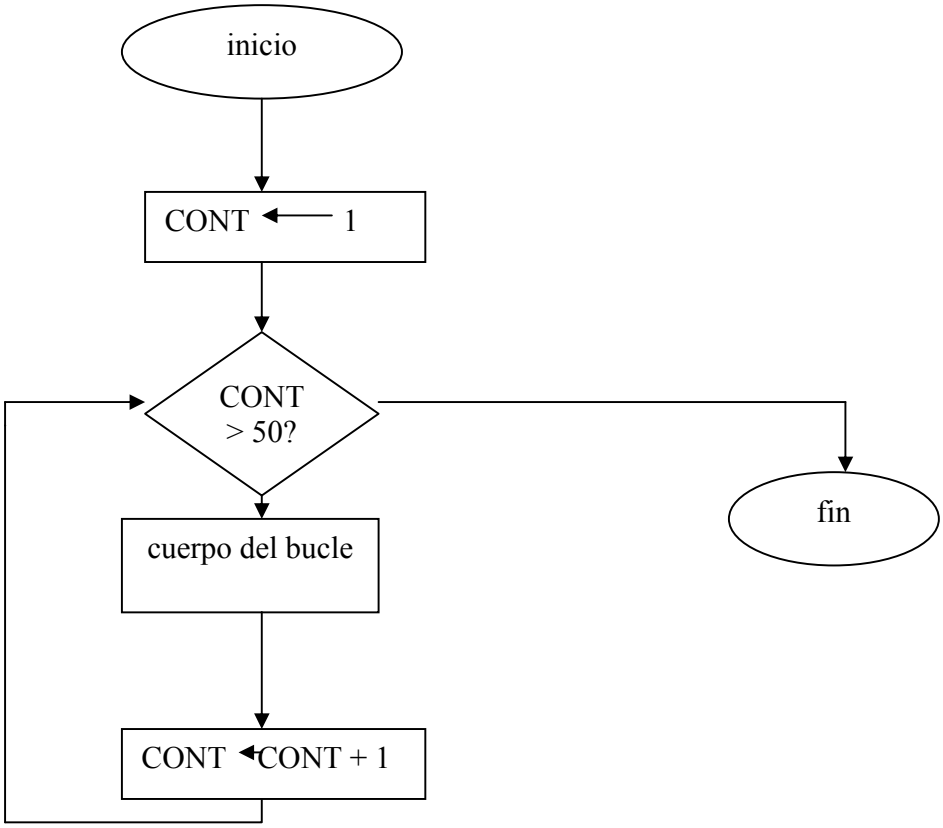


Figura 3.4. Ejemplo de contador positivo

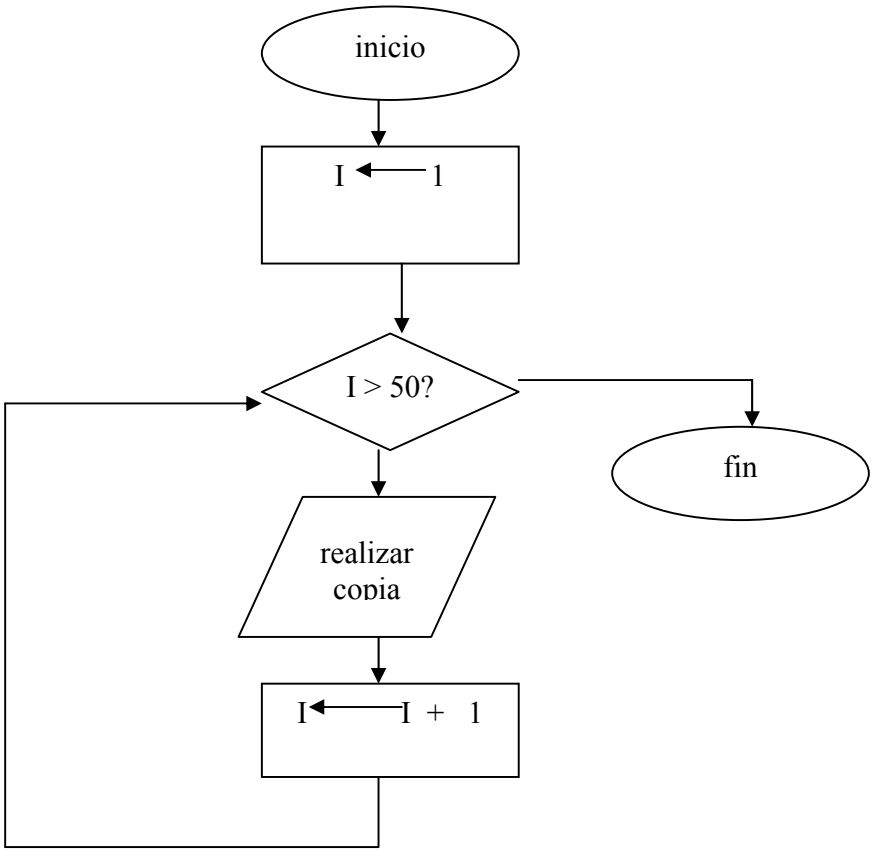
Saber hacer en la práctica: (2 hrs.)

Aplicar el proceso de desarrollo de un programa, para realizar un programa en la Cantidad mínima de tiempo y con los menores errores.

Ejemplo 3.1

Si desea hacer cincuenta copias de un informe de ventas, ¿Cuál seria el diagrama de flujo que resolverá el problema?

La tarea implica realizar cincuenta veces la impresión de una copia del informe. Dado que el contador I sé inicializa a 1, el valor de este contador coincidirá con el número de copias que se hayan realizado. Cuando I tome el valor 51, significara que se habrán realizado cincuenta copias.


inicio
I ← 1
I > 50?
fin
realizar
copia
I ← I + 1

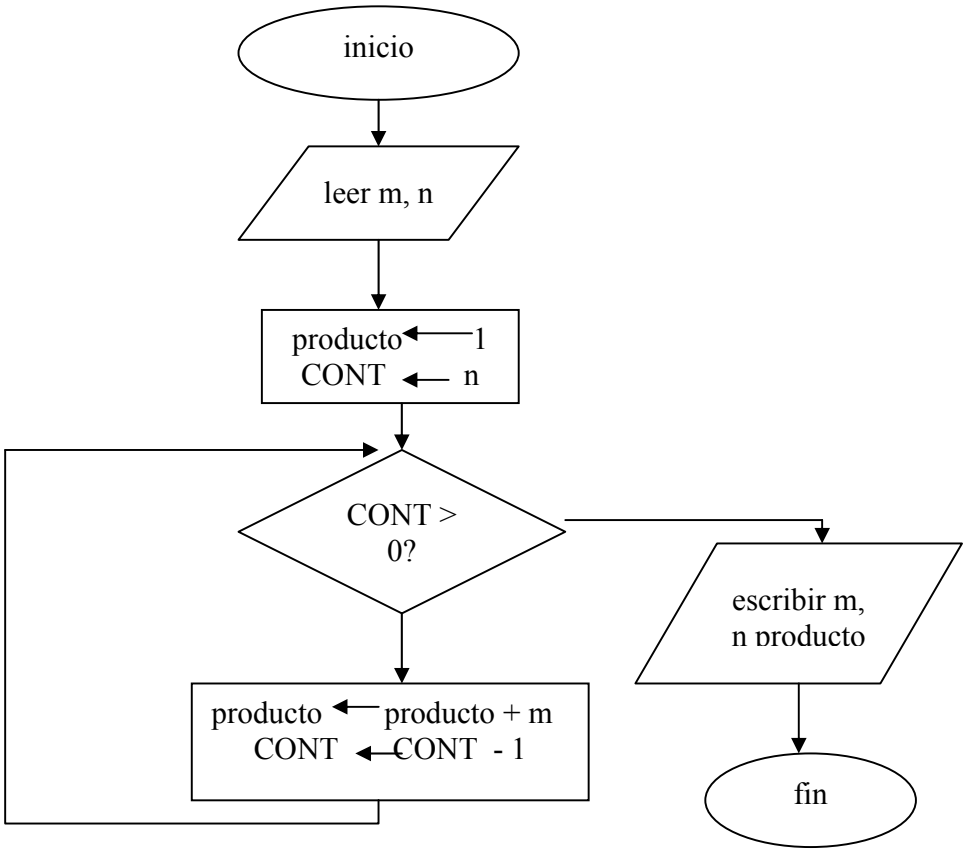


Figura 3.7. Bucle contador negativo.

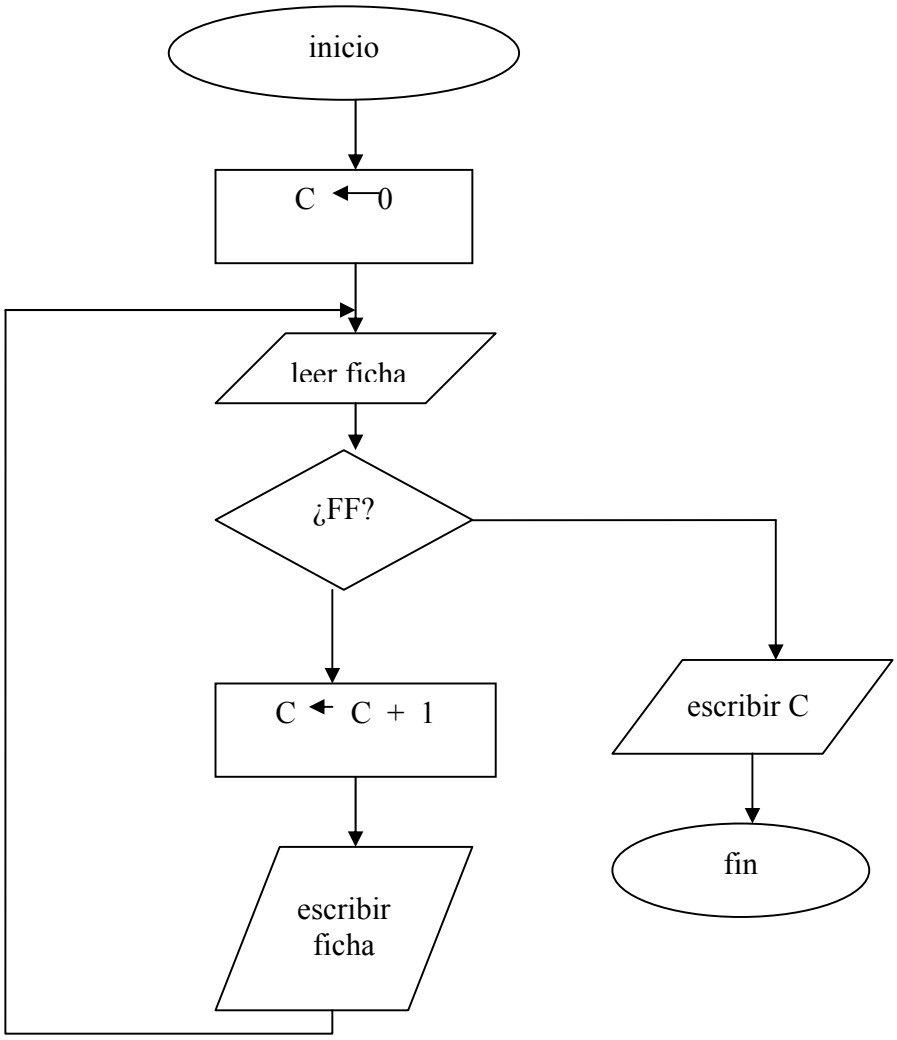
Ejemplo 3.2

Leer todas las fichas de un archivo e imprimir él numero total de ellas.

Para realizar estas operaciones es usual poner en todos los archivos una ficha denominada fin de fichero o de archivo (FF), que se sitúa al final del archivo, con lo cual cuando se lee el archivo y se encuentra esta ficha (FF) se sabe que se ha alcanzado el final del archivo.

El *algoritmo* en lenguaje natural seria:

1. **inicio**
2. C ← 0
3. **leer** (ficha)
4. **si** (FF) **ir_a** (8)
5. C ← C + 1
6. **escribir** (ficha)
7. **ir _ a** (3)
8. **escribir** (C)
9. **fin**



3.4. Acumulador

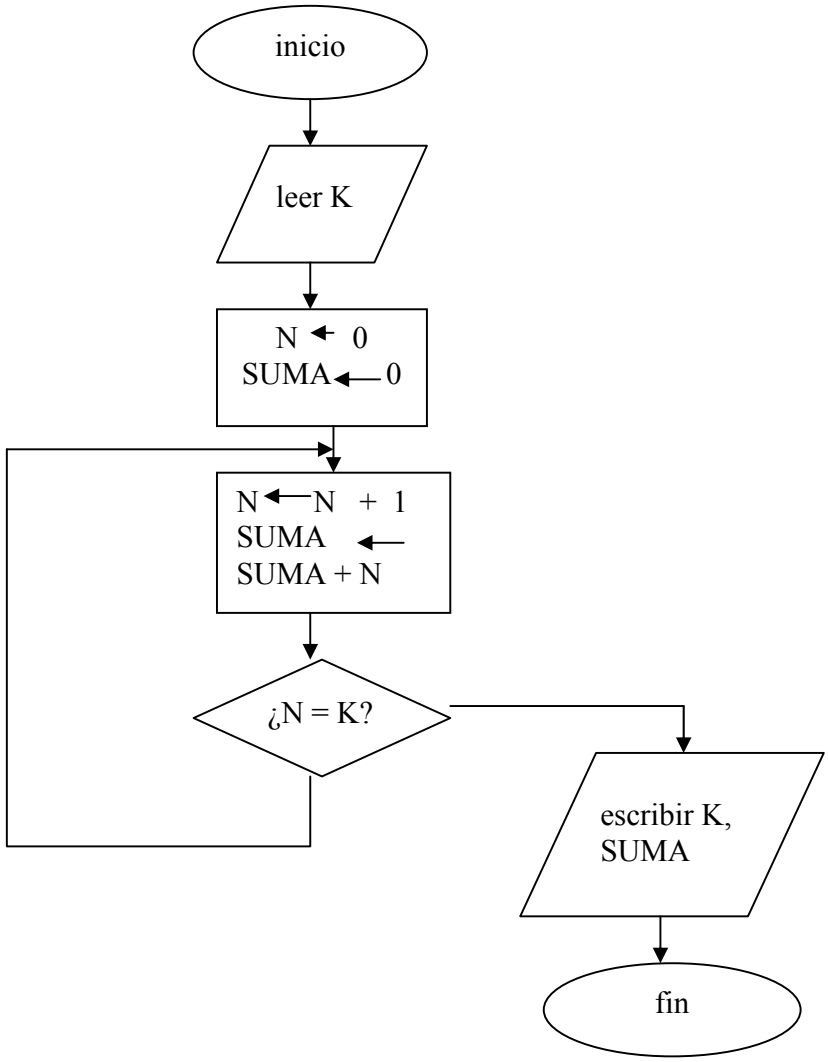
Un *acumulador* o *totalizador* es una variable cuya misión es almacenar cantidades variables resultantes de sumas sucesivas. Realiza la misma función que un contador, con la diferencia de que el incremento o decremento de cada suma es variable en lugar de constante, como en el caso del contador.

Se representa por la instrucción S ← S + N, donde N es una variable y no una constante.

Ejemplo 3.3

Sumar los primeros k números enteros. El algoritmo correspondiente es:

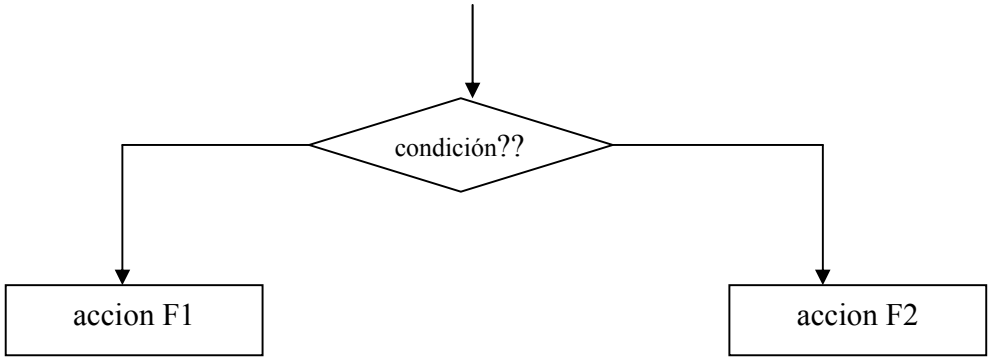
1. **inicio**
2. **leer** (K)
3. N ← 0
4. SUMA ← 0
5. N ← N + 1
6. SUMA ← SUMA + N
7. **si** N = K, **ir_a** (9)
8. **ir_a** (5)
9. **escribir** (K, SUMA)
10. **fin**



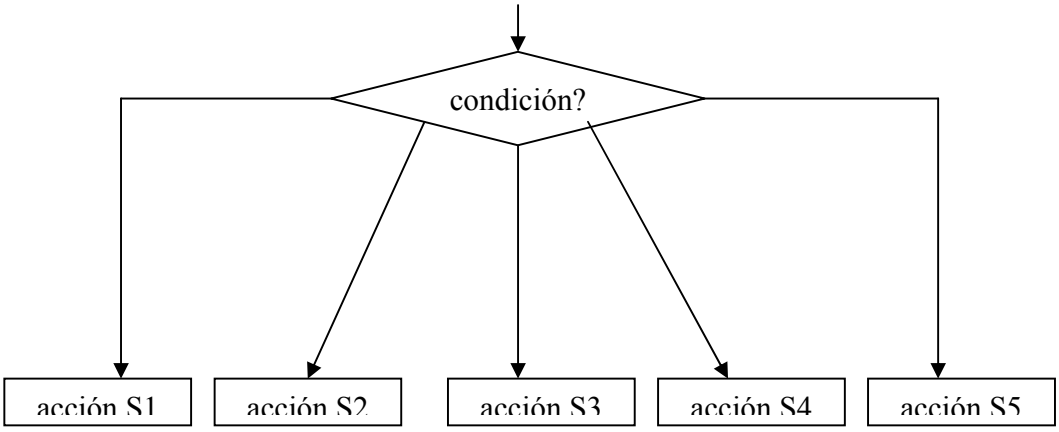
3.5. Decisión o selección

Cuando el programador desea especificar dos o más caminos alternativos en un algoritmo (o un programa), se deben utilizar estructuras de decisión o selección.

Una instrucción de decisión o selección evalúa una condición y, en función del resultado de esa condición, se bifurcara a un determinado punto.

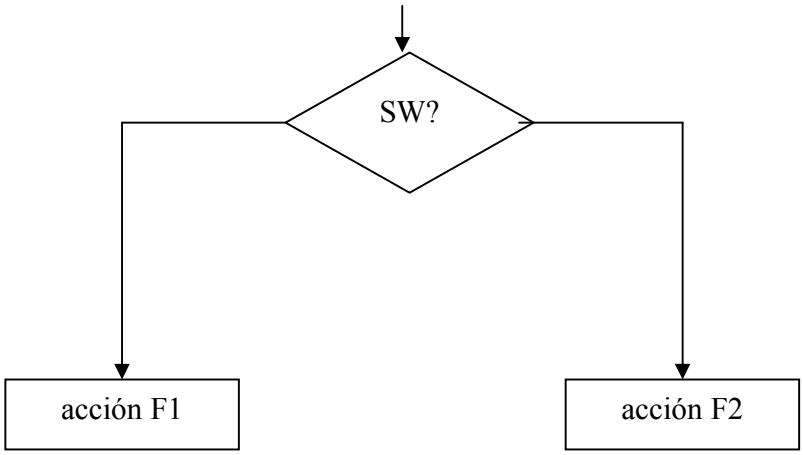


En el caso anterior, se tiene una instrucción alternativa con dos posibles caminos: si la condición es verdadera, se ejecuta la acción S1, y si la condición es falsa, se ejecuta la acción S2. Es posible también realizar diferentes acciones para el caso de alternativas múltiples. En el caso de que la condición ofreciese cinco posibles soluciones, se representaría como

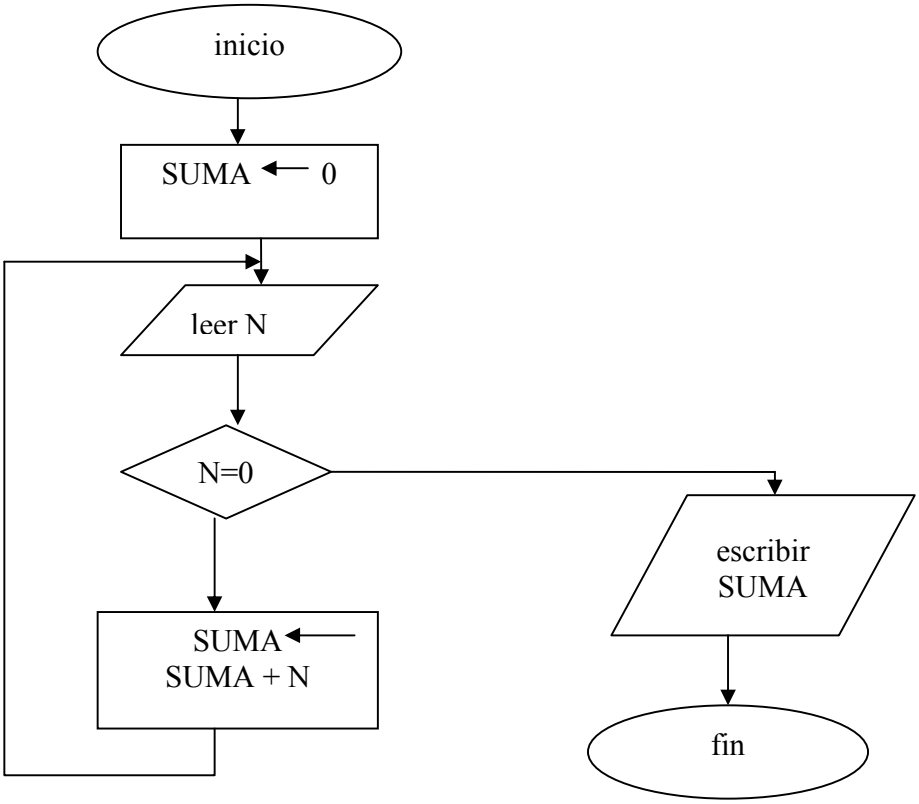


3.6. Interruptores

Un *interruptor* o *conmutador* (*swithc*) –a veces se les denomina *indicador, bandera* (*flag)* – es una variable que puede tomar diversos valores a lo largo de la ejecución del programa y que permite comunicar información de un parte a otra del mismo. Los interruptores pueden tomar dos valores diferentes 1 y 0. (de ahí su nombre de interruptor, “encendido” / “apagado”, “abierto” / “cerrado”.)



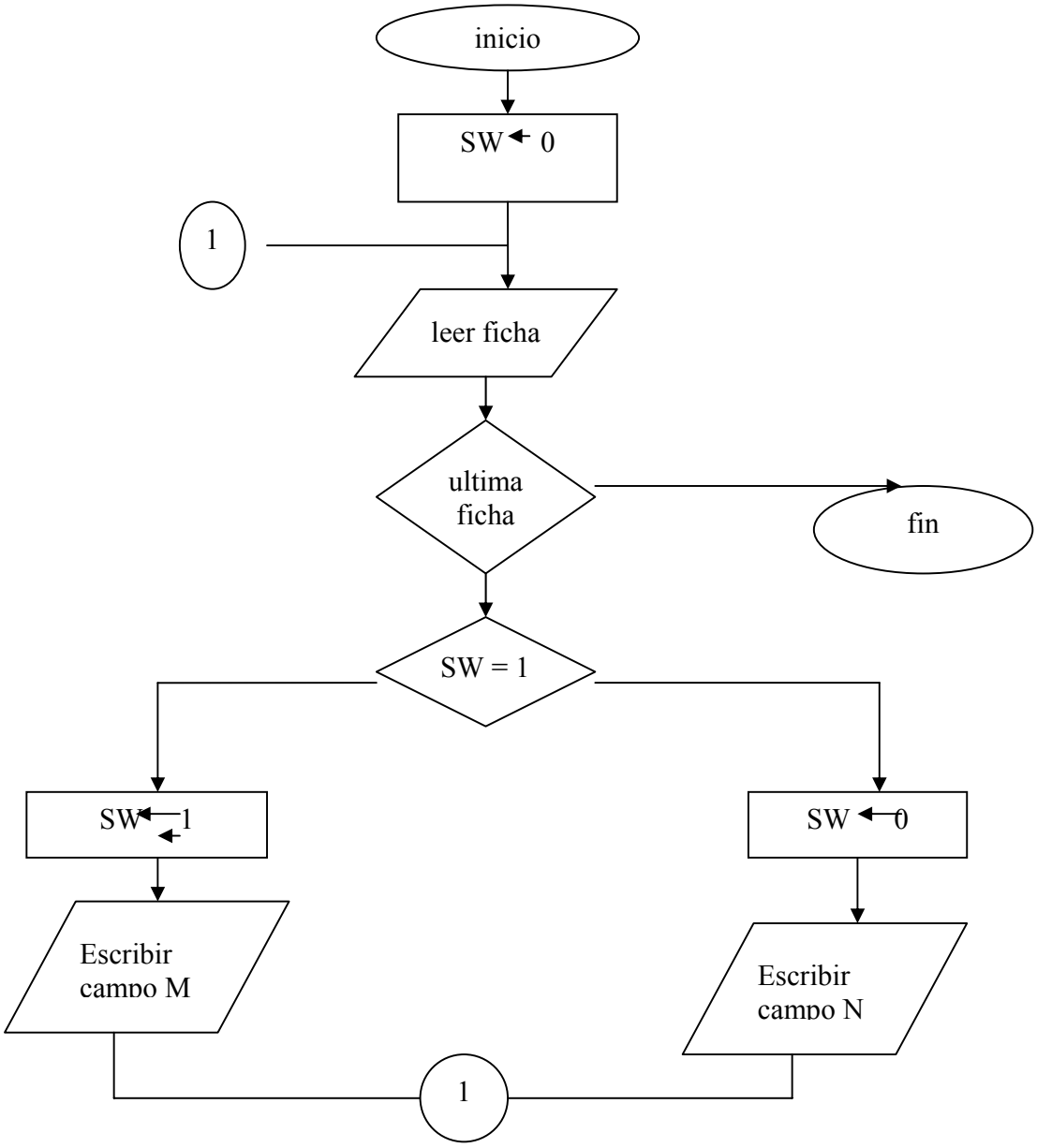
Si SW es igual a 1, se ejecuta la acción S1, y si SW es igual a 0, se ejecuta la acción S2.



Ejemplo 3.4

Se tiene un archivo con un conjunto de fichas con dos campos, M y N. se desea listar el campo M de las fichas pares y el campo N de las fichas impares.

Los interruptores –SW-- suelen iniciar a un valor determinado (0 o 1) y luego se ha de modificar su valor alternativamente a medida que se leen las fichas impares y pares. De este modo, cuando SW = 0, se leerán las fichas impares y cuando SW = 1, se leerán las fichas pares.



IV
LENGUAJES DE PROGRAMACIÓN

Objetivo particular de la unidad
Elaborar programas en un lenguaje de programación

Tipo de aprendizaje por desarrollar
Habilidades

IV.1 LENGUAJES DE PROGRAMACION

Saber en la teoría (2 hrs.)

Sentencias de control de flujo, construcción de expresiones

4.1 SENTENCIAS IF Y SWITCH.

Introducción

Los programas definidos hasta este punto se ejecutan de modo esencial, es decir, una sentencia después de otra. La ejecución comienza con la primera sentencia de la función y prosigue hasta la última sentencia, cada una de las cuales se ejecuta una sola vez. Esta forma de programación es adecuada para resolver problemas sencillos. Sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuales son las sentencias que se ejecutan en que momentos. Las estructuras o construcciones de control controlan la sentencia o flujo de ejecución de las sentencias. Las estructuras de control se dividen en tres grandes categorías en función de flujo de ejecución: secuencia, selección y repetición.

Esta unidad considera las estructuras selectivas o condicionales –sentencias **if** y **switch**- que controlan si una sentencia o una lista de sentencias se ejecutan en función del cumplimiento o no de una condición. Para soportar estas construcciones, el estándar **ANSI/ISO C++** soporta el tipo lógico **bool**.

Conceptos clave.

- Estructura de control.
- Tipo de dato **bool**.
- Sentencia compuesta.
- Estructura de control selectiva.
- Sentencia **if**.
- Sentencia **switch**.
- Sentencia **break**.
- Sentencia **enum**.

Estructuras de control.

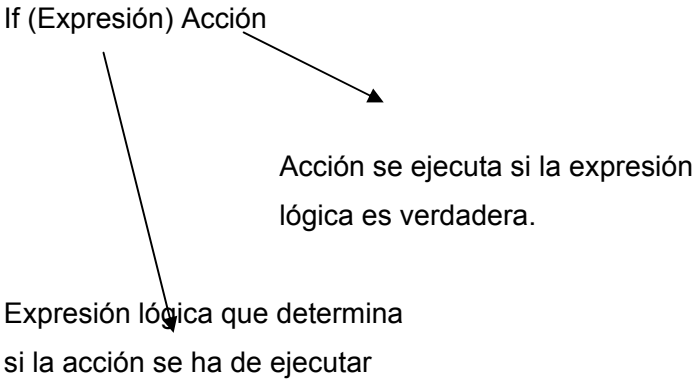
Las **estructuras de control** controlan el flujo de ejecución de un programa o función. Las estructuras de control permiten combinar instrucciones o sentencias individuales en una simple unidad lógica con un punto de entrada y un punto de salida.
Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecución: secuencia, selección (decisión) y repetición. Hasta este momento solo se ha utilizado el flujo secuencial. Una **sentencia compuesta** es un conjunto de sentencias encerradas entre llaves ({ y }) que se utiliza para especificar un flujo secuencial.

```
{

    Sentencia1;
    Sentencia2;
     .
     .
     .
    Sentencia n;
}
```

El control fluye de la sentencia sentencia1 a la sentencia2, y así sucesivamente. Sin embargo, existen problemas que requieren etapas con dos o más opciones o alternativas a elegir en función del valor de una condición o expresión.

4.2 LA SENTENCIA IF

En C++, la estructura de control de selección principal es una sentencia **if**. La sentencia if tiene dos alternativas o formatos posibles. El formato más sencillo tiene la sintaxis siguiente:



La sentencia **if** funciona de la siguiente manera. Cuando se alcanza la sentencia **if** dentro de un programa, se evalúa la expresión entre paréntesis que viene a continuación del **if**. Si **Expresión** es verdadera, se ejecuta acción; en caso contrario no se ejecuta acción (en su formato más simple, Acción es una sentencia simple, y en los restantes formatos, es una sentencia compuesta). En cualquier caso la ejecución del programa continúa con la siguiente sentencia del programa.

EJEMPLO 4.1

PRUEBA DE DIVISIBILIDAD

```
void main ( )
{
int n,  d;
cout   <<   “introduzca dos  enteros:” ;
cin >> n >>  d;
if (n %d  = = 0)  cout  << n <<  “es divisible por”   << d <<
end1;
}
```

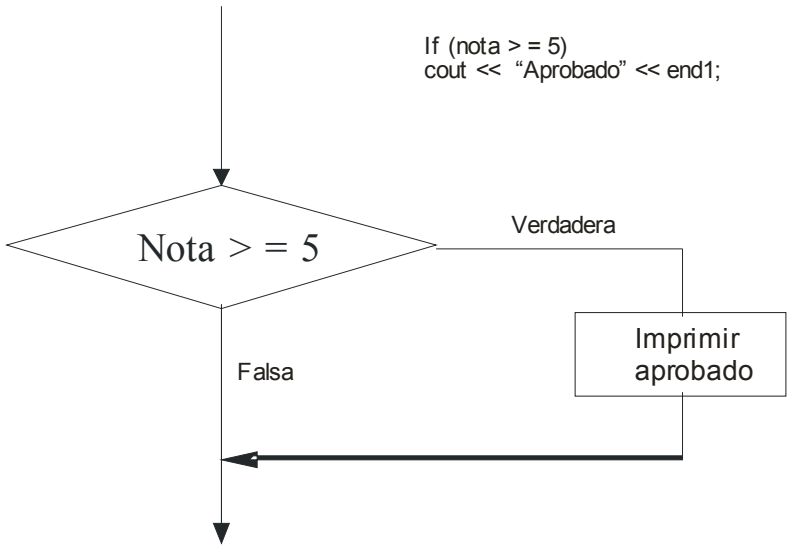
Ejecución

Introdusca dos enteros: **36 4**
36 es divisible por **4**

Este programa lee dos números enteros y comprueba cual es el valor del resto de la división **n** entre **d (n / d).** si el resto es cero, **n** es divisible por **d.** (En nuestro caso 36 es divisible por 4; ya que 36 : 4 = 9 y el resto es 0.)

EJEMPLO 4.2

REPRESENTAR LA SUPERACION DE UN EXAMEN (NOTA > = 5, APROBADO)



EJEMPLO 4.3

```
/ / programa demo  if
# incluye <iostream.h>          / /  E / S de C++

void main ( )
{

        float numero;
        / /   obtener  numero  introducido por usuario
        cout << "introduzca  un  numero  positivo  o  negativo: ";
        cin>>  numero;
        / /  compara  numero con cero
        if  (numero  >  0)
            cout  << numero  >>  "es mayor que cero"  <<  endl;
```

La ejecución de este programa produce

```
Introduzca un numero positivo o negativo:  10.15
10.15 es mayor que cero
```

Si en lugar de introducir de introducir un número positivo se introduce un número negativo, ¿Qué sucede? Nada. El programa es tan simple que solo puede comprobar si el número es mayor que cero.

EJEMPLO 4.4

```
/ / programa  demo2  if
#  include <iostream.h>

void main ( )
{
        float numero;
        / /  obtener numero introducido por usuario
        cout  “introduzca un numero positivo o negativo;”;
        cin  >>  numero;
        / / comprar numero a cero
        if  (numero > 0)
            cout << numero  <<  “es mayor que cero”  << endl;
        if (numero < 0)
            cout << numero  <<  “es mayor que cero”  << endl;

        if (numero  = = 0 )
          cout <<  numero  <<  “ es igual a cero”  << endl;
}
```

Este programa simplemente añade otra sentencia if que comprueba si el numero introducido es menor que cero. Realmente, una tercera sentencia if se añade también que comprueba si el número es igual a cero.

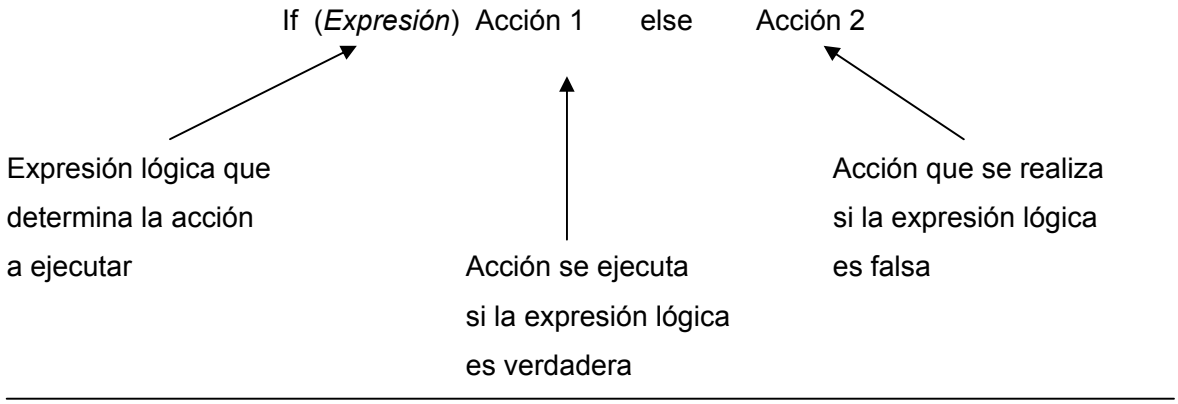
EJEMPLO 4.5

Visualizar el valor absoluto de un número leído del teclado.

```
/ / programa de calculo de valor absoluto de la entrada
#include <iostream.h>
#include <string.h >
int main ( )   {
        cout << “introduzca un numero:”;
        int Valor;
        if (Valor)  <  0 )
           Valor =  - Valor;
        cout <<  Valor  << “es positivo”  <<  endl;
        return 0;
}
```

4.3 SENTENCIA if , DOS ALTERNATIVAS : if – else

Un segundo formado de la sentencia if es la sentencia if – else. Este formato de la sentencia if tiene la siguiente sintaxis:



En este formato acción1 y acción2 son individualmente o bien una única sentencia que termina en un punto y coma (;) o un grupo de sentencias encerrado entre llaves. Cuando se ejecuta la sentencia if – else, se evalúa *expresión.* Si *expresión* es verdadera, se ejecuta la accion1 y en

caso contrario se ejecuta la acción2. la figura muestra la semántica de la sentencia de la sentencia if – else.

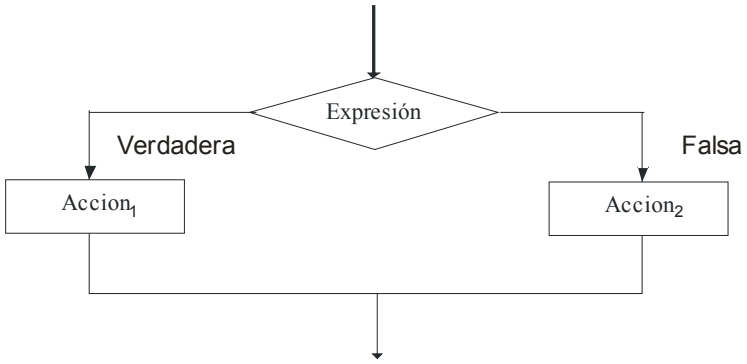


Diagrama de flujo de la representación de una sentencia if – else.

EJEMPLO 4.6

1. if (salario > = 100. 000)
   salario_neto = salario – impuestos;

else
   salario_neto = salario;

Si salario es mayor que 100 000, se calcula el salario neto, restándole los impuestos; en caso contrario (else), el salario neto es igual al salario (bruto).

2. if (Nota > = 5)
   cout << “aprobado” << endl;

else
   cout << “reprobado” << endl;

Formatos

1.
```
if (expresión_logica)
    sentencia
```

2.
```
if (expresión_logica)
    sentencia
else
    sentencia
```

3.
```
if (expresión_logica) sentencia
```

4.
```
if (expresión_logica) sentencia1 else sentencia2
```

Si *expresión lógica* es verdadera, se ejecuta sentencia o bien sentencia1, si es falsa (*si no,* en caso contrario), se ejecuta *sentencia2.*

EJEMPLO 4.7

```
1. if  (x > 0.0)
       producto = producto *  x;
2. if  (x = 0.0 )
       producto = producto * x;
/ / se ejecuta la sentencia de asignación cuando x no es igual a 0.
/ / en este caso producto se multiplica por x y el nuevo valor se
/ /  guarda en producto reemplaza el valor antiguo.
/ / si x es igual a 0, la multiplicación no se ejecuta.
```

EJEMPLO 4.8

Prueba de divisibilidad (igual que 1, al que se ha añadido la cláusula else)

```
void main ( )
   {
      int n, d;
      cout << "introduzca dos enteros:";
      cin >> n >> d;
      if (n % d  = = 0 ) cout << n << "es divisible por" << d << endl;
      else
```

```
        cout << n << "no es divisible por" << d << endl;
}
```

Ejecución

Introduzca dos enteros 36 5

36 no es divisible por 5

Comentario

36 no es divisible por 5, ya que 36 dividido entre 5 produce un resto 1 (n % d = = 0, es falsa, y se ejecuta la cláusula else).

EJEMPLO 4.9

Calcular el mayor de dos números leídos del teclado y visualizarlo en pantalla.

```
void main ( )
{
    int x, y;
    cout << "introduzca dos enteros:";
    cin >> x >> y;
    if (x  >  y) cout <<  x  << endl;
    else
        cout <<  y   << endl;
}
```

Ejecución

Introduzca dos enteros: 17 54

54

4.4 SENTENCIAS if- else ANIDADAS:

Hasta este momento, las sentencias *if* implementan decisiones que implican una o dos alternativas. En esta sección se mostrará cómo se puede utilizar la sentencia if para implementar decisiones que impliquen diferentes alternativas.

Una sentencia *if* es anidada cuando la sentencia de la rama verdadera o la rama falsa es a su vez una sentencia *if.* Una sentencia *if* anidada se puede utilizar para implementar decisiones con varias alternativas o Multi-alternativas.

Sintaxis:

```
if       (condición)
    sentencial
else  if  (condición2)
    sentencia2


else  if  (condición,)
    sentencia n
else
   sentencia
```

EJEMPLO 4.10

```
// incrementar contadores de números positivos, números
// negativos o ceros

 if (x > 0)
     num-pos = num-pos + 1;
else
     if (x < 0)

         num_neg = num_neg + 1;
    else

        num_ceros = num_ceros + 1;
```

La sentencia if anidada tiene tres alternativas. Se incrementa una de las tres variables (num-pos, num-neg. y num_ceros) en 1, dependiendo de que x sea mayor que cero, menor que cero o igual a cero, respectivamente. Las cajas muestran la estructura lógica de la sentencia if anidada; la segunda sentencia if es la acción o tarea falsa (a continuación de else) de la primera sentencia if.
La ejecución de la sentencia if anidada se realiza como sigue: se comprueba la primera condición (x > 0); si es verdadera, num_pos se incrementa en 1 y se salta el resto de la sentencia if. Si la primera condición es ~ falsa, se comprueba la segunda condición (x < o); si es verdadera num_neg se incrementa en uno; en caso contrario se incrementa num_ceros en uno. Es importante considerar que la segunda condición se comprueba *sólo si* la primera condición es falsa.

Sangría en las sentencias if anidadas

El formato multibifurcación se compone de una serie de sentencias if anidadas, que se pueden escribir en cada línea una sentencia if. La sintaxis multibifurcación anidada es:

Formato 1:

```
if (expresión-lógica1)
    sentencial
else
    if (expresión-lógica2)
    else
        if (expresión-lógica3)
                sentencia]
            else
                if (expresión-lógica4)
                   sentencia.
                else
                    sentencias
```

Formato 2:

```
if (expresión-lógica l)
    sentencia 1
else if (expresión-lógica2)
    sentencia 2
else if (expresión lógica3)
     sentencia 3
else if (expresión-lógica 4)
    sentencia 4
else
    sentencias
```

EJEMPLO 4.11

```
l. if (x > 0}
        if (y > 0)
            z = sqrt (x} + sqrt (y};

2. if (x > 0)
        if   (y > 0}
            z = sqrt (x} + sqrt (y}. ;
   else
        cerr« "\n *** Imposible calcular z" « endl;
```

EJEMPLO4.12

```
/ / comparación _if
// ilustra las sentencias compuestas if-else
#include <iostream.h>
Void main ()
    cout « if introduzca un número positivo o negativo: ";
    cin » número;
```

```
 / / comparar número a cero
if (numero >0)

    cout «número « " es mayor que cero" « endl;
    cout « "pruebe de nuevo introduzca un numero negativo" « endl;
 }
else if (número < O)
    (
   cout «número « " es menor que cero" « endl;
   cout« "pruebe de nuevo introduciendo un número negativo" « endl;
    }
   else
    {
       cout « número « " es igual a cero" « endl;
       cout «" ¿por qué no introduce un número negativo?« endl;
   }
}
```

Comparación de sentencias if anidadas y secuencias de sentencias if

Los programadores tienen dos alternativas: (1) usar una secuencia de sentencias if; (2) una única sentencia if anidada. Por ejemplo, la sentencia if del Ejemplo 4.6 se puede reescribir como la siguiente secuencia de sentencias if.

```
if (x > 0)


   números = números + 1;

if (x < 0)

   num_neg = num_neg + 1;

if (x == O)

   num_ceros = num_ceros + 1;
```

Aunque la secuencia anterior es lógicamente equivalente a la original, no es tan legible ni eficiente. Al contrario que la sentencia i f anidada, la secuencia no muestra claramente cuál es la sentencia a ejecutar para un valor. determinado de x. Con respecto a la eficiencia, la sentencia i f anida- da se ejecuta más rápidamente cuando x es positivo ya que la primera condición (x > o) es verdadera, lo que significa que la parte de la sentencia if a continuación del primer else se salta. En contraste, se comprueban siempre las tres condiciones en la secuencia de sentencias if. Si x es negativa, se comprueban dos condiciones en las sentencias *i* f anidadas frente a las tres condiciones de las secuencias de sentencias if.

EJEMPLO 4.13

Calcular el mayor de *tres números enteros.*

```
void main ()
 {
   int a, b, c, mayor;

   cout « "Introduzca tres enteros:";
   cin» a » b » c;

   if (a > b)
```

```
  if (a> c) mayor = a;
    else mayor = c;
else
    if (b < c) mayor = b;
    else mayor = c;
cout « "El mayor es " «  mayor « endl;
```

Ejecución

```
Introduzca tres enteros: 77 54 85
El mayor es 85.
```

Análisis

Al ejecutar el primer if, la condición (a > b) es verdadera, entonces se ejecuta la segunda i f. En el segundo i f la condición (a > c) es falsa, en consecuencia el primer else y mayor igual a 85 y se termina la sentencia if y se ejecuta la última línea y se visualízale mayor es 85.

4.5 SENTENCIA DE CONTROL SWITCH

La sentencia switch es una sentencia C++ que se utiliza para selecciona una de entre múltiples alternativas. La sentencia switch es especialmente útil cuando la selección se basa en el valor de una variable simple o de una expresión simple denominada *expresión de controlo selector.* El valor de esta expresión puede ser de tipo int o char, pero no de tipo doble.

Sintaxis

```
switch (selector)
{
    Case etiqueta l: sentencias l;
                        break;
    case etiqueta 2: sentencias2;
                        break;


    case etiqueta  n: sentencias n;
                        break;
    default: sentencias d; // opcional
}
```

La expresión de control o *selector* se evalúa y se compara con cada una de las etiquetas de case. La expresión *selector* debe ser un tipo ordinal (por ejemplo, int, char, bool. pero no float o string). Cada *etiqueta* es" un valor único, constante, y cada etiqueta debe tener un valor diferente de los otros. Si el valor de la expresión selector es igual a una de las etiquetas case -

por ejemplo, *etiqueta,-* entonces la ejecución comenzará con la primera sentencia de la secuencia *secuencia;* y continuará hasta que se encuentra una sentencia break (o hasta que se encuentra el final de la sentencia de control switch).
El tipo de cada etiqueta debe ser el mismo que la expresión de *selector.* Las expresiones están permitidas como etiquetas pero sólo si cada operan- do de la expresión es por sí misma una constante -por ejemplo, 4 + 8 o bien m * 15, siempre que m hubiera sido definido anteriormente como constante con nombre.
Si el valor del selector no está listado en ninguna etiqueta case, no se ejecutará ninguna de las opciones a menos que se especifique una acción por defecto (omisión). La omisión de una etiqueta default puede crear un error lógico difícil de prever. Aunque la etiqueta default es opcional, se recomienda su uso a menos que se esté absolutamente seguro de que todos los valores de *selector* estén incluidos en las etiquetas case.
Una sentencia break consta de la palabra reservada bread seguida por un punto y coma. Cuando la computadora ejecuta las sentencias siguientes a una etiqueta case, continúa hasta que se alcanza una sentencia bread. Si la computadora encuentra una sentencia bread, termina la sentencia switch. Si se omiten las sentencias bread, después de ejecutar el.código de case, la computadora ejecutará el código que sigue a la siguiente case.

EJEMPLO 4.14

```
// programa de ilustración de la sentencia switch
#include <iostream.h>

int main ()
 { .
    char nota;
    cout « "Introduzca calificación (A-H) y pulse Intro:";
     cin » nota;

    switch (nota)
     {
        Case 'A': cout« "Excelente."
                              « "Examen superado\n";
                  break;
         Case 'B': cout« "Notable.";
                  Cout « "Suficiente\n";
                  break;
         Case 'C': cout« "Aprobado\n";
                  break;
         Case 'D':
         Case 'F': cout« "reprobado\n";
                  break;
                                                                    -
            cout "no es posible esta nota";
     }
     cout « "Final de programa" « endl;
    return O;
```

Cuando se ejecuta la sentencia switch, se evalúa nota; si el valor de la expresión es igual al valor de una etiqueta, entonces se transfiere el flujo de control a las sentencias asociadas con la etiqueta correspondiente... Si ninguna etiqueta coincide con el valor de nota se ejecuta la sentencia default y las sentencias que vienen detrás de ella. Normalmente la última sentencia de las sentencias que vienen después de una case es una sentencia bread. Esta sentencia hace que el flujo de control del programa salte a la última sentencia de switch. Si no existiera break, se ejecutarían también las sentencias restantes de la sentencia switch.

EJEMPLO 4.15

```
int tipo_vehículo;
cout « "Introduzca tipo de vehículo:";
cin » tipo_vehículo, peaje;

switch (tipo_vehículo)
{
    case 1:
        cout «"turismo";
        peaje = 500;
        break;..Si se omite esta break, el vehículo primero será turismo y luego autobús.
    Case 2:
        Cout « "autobús";
        Peaje = 3000;
        break;
    Case 3:
        cout « "motocicleta";
        Peaje = 300;
        break;
  default:
        cout « "vehículo no autorizado";
    }
```

Cuando la computadora comienza a ejecutar una sentencia case, no detiene su ejecución hasta que se encuentra o bien una sentencia brek o bien una sentencia switch.

LA SENTENCIA WHILE:

Un bucle es cualquier construcción de programa que repite una sentencia o secuencia de sentencias un numero de veces. La sentencia (o grupo de sentencias) que se repite en un bloque se denomina **cuerpo** del bucle y cada repetición del cuerpo del bucle se llama **iteración** del bucle. Las dos principales cuestiones de diseño en la construcción del bucle son: ¿cuál es el cuerpo del bucle? ¿Cuántas veces se itera el cuerpo del bucle?

Un bucle *while* tiene una *condición* del bucle (una expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que un bucle *while* es un bucle *pretest* de modo que cuando se ejecuta el mismo, se evalúa la condición antes de que se ejecute el cuerpo del bucle.

Sintaxis

```
1 while (condición _ bucle)
        sentencia; ──────► cuerpo
2 while (condición _ bucle)
{
    sentencia-1;
    sentencia-2
        .
        .                  cuerpo
        .
    sentencia-n;
}
```

While ——————— palabra reservada C++
Condición _bucle ——— expresión lógica o booleana
Sentencia ——————— sentencia simple o compuesta

El comportamiento de una sentencia (bucle) *while* es:

1.- se evalúa la condicion_bucle
2.- si condicion_bucle es verdadera:
   a) la sentencia especificada, denominada el cuerpo del bucle, se ejecuta.
   b) Vuelve el control al paso 1
3.- en caso contrario:
   El control se transfiere a la sentencia siguiente al bucle o sentencia *while*.

> El cuerpo del bucle se repite **mientras** que la expresión lógica (condición del bucle) sea verdadera. Cuando se evalúa la expresión lógica y resulta falsa, se termina y se sale del bucle y se ejecuta la siguiente sentencia de programa después del cuerpo de la sentencia *while*.

```
// cuenta hasta 10 (1 a 10)
int. X = 1;
while (X<=10)
     cout << "X :" << ++X;
```

EJEMPLO 4.16

```
// visualizar n asteriscos
contador = 0;                  inicialización
while (contador < n)           prueba /condición
{
cout << " * ";
contador ++;                   actualización (incrementa en 1 contador)
} //                           fin de while
```

La variable que representa la condición del bucle se denomina también **variable de control del bucle** debido a que su valor determina si el cuerpo del bucle se repite. La variable de control del bucle debe ser: (1) inicializada, (2) comprobada, (3) actualizada para que el cuerpo del bucle se ejecute adecuadamente. Cada etapa se resume así:

1.- ***Iniciación:*** contador se establece a un valor inicial (se inicializa a cero, aunque podría ser otro su valor) antes de que alcance la sentencia while.

2***.- prueba / condición:*** se comprueba el valor de contador antes de que comienza la repetición de cada bucle.

3.- ***actualización:*** contador se actualiza (su valor se incrementa en 1 mediante el operador ++) durante cada iteración.

Si la variable de control no actualiza el bucle, este se ejecutara <<siempre>>. Tal bucle se denomina **bucle infinito**. En otras palabras un bucle infinito se producirá cuando la condición del bucle permanece y no se hace falsa en ninguna iteración.

```
//     bucle infinito
contador = 1;
while (contador < 100)
{
cout <<contador << endl;
contador --;  ←————————  decrementa en 1 contador.
}
```

Contador se inicializa a 1 (menor de 100) y como contador --decrementa en 1 el valor de contador en cada iteración, el valor de contador nunca llegara a 100, valor necesario de contador para que la condición del bucle sea falsa. Por consiguiente, la condición contador < 100 siempre será verdadera, resultando un bucle infinito, cuya salida será:

```
1
0
-1
-2
-3
-4
.
.
.
```

EJEMPLO 4.17

```
//     bucle de muestra con while
int. main ( )
{
int  contador = 0;          //    inicializa la condición
while (contador < 5)     //     condición de prueba
{
contador ++;              //     cuerpo del bucle
cout << “contador: “ << contador << “ \ n”;
}
cout << “terminado  contador:” <<contador << “.\n”;
return 0;
}
```

ejecución

```
contador: 1
contador: 2
contador: 3
contador: 4
contador: 5

terminado contador: 5
```

4.7 OPERADORES DE INCREMENTO Y DECREMENTO (++, --)

C++ ofrece los operadores de incremento (++) y decremento (--) que soporta una sintaxis abreviada para añadir o restar 1 al valor de una variable. Recordemos del capitulo 3 la sintaxis de ambos operadores:

```
++ nombrevariable        //preincremento
nombrevariable ++        //postincremento

-- nombrevariable        //predecremento
nombrevariable --        //postdecremento
```

EJEMPLO 4.18

Si i es una variable entera cuyo valor es 3, las variables k e i toman los valores sucesivos que se indican en las sentencias siguientes:

```
k = i++;          //asigna el valor 3 a k y 4 a i
k = ++i;          //asigna el valor 5 a k y 5 a i
k = i--;          //asigna el valor 5 a k y 4 a i
k = --i;          //asigna el valor 4 a k y 3 a i
```

EJEMPLO 4.19

uso del operador de incremento ++ para controlar la iteración de un bucle.

```
//programa calculo de calorías
#include <iostream.h>
int main ( )
{
int. num_de_elementos, cuenta,calorias_por_alimento, calorias_total;
cout <<"¿cuántos alimentos ha comido hoy?";
cin>> num_de_elementos;
calorías_total = 0;
cuenta = 1;
cout<<"introducir el numero de calorías de cada alimento";
     <<num_elementos <<"alimentos tomados :\n";
while (cuenta ++ <= num_de_elementos)
{
cin>> calorías_por_alimento;
calorías_total = calorías_total + calorías_por_alimento;
}
cout<< "las calorías totales consumidas hoy son = ";
     <<calorías_total <<endl;
return 0;
}
```

ejecución de muestra

¿cuántos alimentos ha comido hoy? **8**
introducir el numero de calorías de cada uno de los 8 alimentos tomados:
500 50 1400 700 10 5 250 100
las calorías totales consumidas hoy son = **3015**

4.8 TERMINACIONES ANORMALES DE UN CICLO

Un error típico en el diseño de una sentencia while se produce cuando el bucle tiene solo una sentencia en lugar de varias sentencias como se planeo.
El código siguiente:

```
contador = 1;
while (contador < 25)
      cout<< contador <<endl;
      contador ++;
```

Visualizará infinitas veces el valor 1. es decir, entra en un bucle infinito del que nunca sale por que no se actualiza la variable de control *contador.*

La razón es que el punto y coma al final de la línea *cout<<contador <<endl;* hace que el bucle termine en ese punto y coma, aunque aparentemente el sangrado pueda dar la sensación de que el cuerpo de while contiene dos sentencias: cout ... y contador ++.

El error se hubiera detectado rápidamente se el bucle se hubiera escrito correctamente con una sangría.

```
contador = 1;
while (contador < 25 )
     cout<< contador <<endl;
contador ++;
```

La solución es muy sencilla, utilizar las llaves de la sentencia compuesta:

```
contador = 1;
while (contador < 25 )
{
cout<< contador <<endl;
contador ++;
}
```

4.9 DISEÑO EFICIENTE DE BUCLES.

Una cosa es analizar la operación de un bucle y otra diseñar eficientemente sus propios bucles. Los principios a considerar son: primero, analizar los requisitos de un nuevo bucle con el objetivo de determinar su inicialización, prueba (condición) y actualización de la variable de

control del bucle; segundo, desarrollar patrones estructurales de los bucles que se utilizan frecuentemente.

4.10 BUCLES *WHILE* CON CERO ITERACIONES.

El cuerpo de un bucle no se ejecuta nunca si la prueba o condición de repetición del bucle no se cumple (es falsa) cuando se alcanza *while* la primera vez.

```
contador = 10
while (contador > 100)
{
      ...
}
```

El bucle anterior nunca se ejecutara ya que la condición del bucle (contador > 100) es falsa la primera vez que se ejecuta. El cuerpo del bucle nunca se ejecutara.

4.11 BUCLES CONTROLADOS POR CENTINELA.

Normalmente, no se conoce con exactitud cuantos elementos de datos se procesan antes de comenzar su ejecución. Esto se produce bien por que hay muchos datos a contar manualmente o por que el numero de datos a procesar depende de cómo prosigue el proceso de calculo.

Un medio para manejar esta situación es instruir al usuario a introducir un único dato definido y especificado denominado *valor centinela* como ultimo dato. L condición del bucle comprueba cada dato y termina cuando se lee el valor centinela. El valor centinela se debe seleccionar con mucho cuidado y debe ser un valor que no pueda producirse como dato. En realidad el centinela es un valor que sirve para determinar el proceso del bucle.

EJEMPLO 4.20

```
//  entrada de datos numéricos
//  centinela -1
const int. centinela = -1
cout<<"introduzca la primera nota";
cin>>nota;
while (nota ! = centinela)
{
cuenta ++;
suma += nota;
cout<< "introduzca la siguiente nota:":
cin>> nota;
} //fin de while
cout<< "final";
```

Si se lee el primer valor de nota, por ejemplo 25, y luego se ejecuta el bucle, la salida podría ser esta:

Introduzca primera nota: **25**
Introduzca siguiente nota: **30**
Introduzca siguiente nota: **90**
Introduzca siguiente nota: **-1** **//valor del centinela**
Final.

4.12 BUCLES CONTROLADOS POR INDICADORES (BANDERAS).

Las variables tipo bool. se utilizan con frecuencia como indicadores o banderas de estado para controlar la ejecución de un bucle. El valor del indicador se inicializa (normalmente a falso "*false"*) antes de la entrada al bucle y se redefine (normalmente a verdadero "*true"*) cuando un suceso especifico ocurre dentro del bucle. Un bucle controlado por bandera o indicador se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.
Se desea leer diversos datos tipo carácter introducidos por el teclado mediante un bucle while y se debe terminar el bucle cuando se lea un dato tipo dígito (rango '0' a '9').

La variable bandera digito_leido se utiliza como un indicador que representa cuando un dígito ha sido introducido por el teclado.

Variable bandera	significado
digito_leido	su valor es falso antes de entrar en el bucle y mientras el dato leído sea un carácter y es verdad cuando el dato leído es un dígito.

El problema que se desea resolver es la lectura de datos carácter y la lectura debe detenerse cundo el dato sea numérico (un dígito de '0' a '9'). Por consiguiente, antes de que el bucle se ejecute y se lean los datos de entrada, la variable *digito_leido* se inicializa a falso. Cuando se ejecuta el bucle este debe continuar ejecutándose mientras el dato leído sea un carácter y en consecuencia la variable *digito_leido* toma el valor falso y se debe detener el bucle cuando el dato leído sea un dígito y en este caso el valor de la variable *digito_leido* se debe cambiar a verdadero. El bucle while será similar a:

```
char car.;
digito_leido = false;                    //    no se ha leído ningún dato
while (! digito_leido)
{
cout<< "introduzca un carácter :" ;
cin>> car. ;
digito_leido = (('0' <= car.) && (car. <= '9'));
...
}          //fin de while.
```

El bucle funciona de la siguiente forma:

1.- Entrada del bucle: la variable digito_leido tiene por valor <<falso>>.
2.- La condición del bucle ! digito_leido es verdadera, por consiguiente se ejecutan las sentencias del interior del bucle.

3.- Se introduce por teclado un dato que se almacena en la variable car. Si el dato leído es un carácter la variable digito_leido se mantiene con valor falso.
4.- El bucle se termina cuando se lee un dato tipo dígito (‘0’ a ‘9’) ya que la condición del bucle es falsa.

4.13 LA SENTENCIA BREAK EN LOS BUCLES.

La sentencia break se utiliza para realizar una terminación anormal del bucle. Dicho de otro modo, una terminación antes de lo previsto. Su sintaxis es:
break;

La sentencia break se utiliza para realizar la salida de un bucle while o do-while, aunque también se puede utilizar dentro de una sentencia *switch*, siendo este su uso mas frecuente:

```
while (condicion)
{
if  (condicion 2)
break;
//   sentencias
}
```

EJEMPLO 4.21

El siguiente código extrae y visualiza valores de entrada desde el flujo de entrada cin hasta que se encuentra un valor especificado.

```
int clave;
cin>>clave;
int entrada;
while (cin>>entrada) {
if (entrada ! = clave)
cout <<entrada <<endl;
else
break;
}
```

4.14 BUCLES WHILE (TRUE)

La condición que se comprueba en el bucle while puede ser cualquier expresión valida C++. Mientras que la condición permanezca verdadera, el bucle while continuara ejecutándose. Se puede crear un bucle que nunca termine utilizando el valor true para la condición que se comprueba.

EJEMPLO 4.22

```
1:  //       listado while (true)
2:  #include <iostream.h>
3:  int. main ()
```

```
4:  {
5:  int  contador = 0
6:  while (contador = 0);   //    tambien, while (true)
7:  {
8:  contador ++;
9:  if  (contador > 10)
10:  break;
11:  }
12:  cout<< “contador :” <<contador <<”\n”;
13:  return 0;
14:  }
```

salida

contador : 11

analisis

en la línea 6, un bucle while se establece con una condición que nunca puede ser falsa. El bucle incrementa la variable contador en la línea 8, y a continuación la línea 9 comprueba a ver si el contador es mayor que 10. si no es así el bucle se itera de nuevo. Si el contador es mayor que 10, la sentencia break de la línea 10 termina el bucle while, y la ejecución del programa pasa a la línea 12.

4.15 REPETICIÓN: EL BUCLE FOR:

La sentencia for (bucle for) es un método para ejecutar un bloque de sentencias un número fijos de veces. El bucle for se diferencia del bucle while en que las operaciones de control del bucle se sitúan en un solo sitio: la cabecera de la sentencia.

El bucle for contiene las cuatro partes siguientes:

- Parte de inicialización, que inicializa las variables de control del bucle. Se pueden utilizar variables de control del bucle simple o múltiples.
- Parte de iteración, que contiene una expresión lógica que hace que el bucle realice las iteracciones de la sentencias, mientras la expresión sea verdadera.
- Parte de incremento, que incrementa o decrementa la variable o variables de control del bucle.
- Sentencias, acciones o sentencias que se ejecutarán por cada iteracción del bucle.

EJEMPLO 4.23

```
//      Imprimir hola 10 veces

for     (int j = 0; j < 10; j++)
        cout << “HOLA”;
```

EJEMPLO 4.24

```
for (int j = 0; j < 10; j++)
{
cout << “Hola!” << endl;
cout << “El valor de j es:”<< j <<endl;
}
```

Existen dos formas de implementar la sentencia for que se utilizan normalmente para implementar bucles de conteo: formato ascendente, en el que la variable de control se incrementa y formato descendente en el que la variable de control se decrementa.

Formato ascendente:

```
for (int var_control = valor_inicial; var_control < =   valor_limite: exp incremento)
        sentencia
```

Formato descendente:

```
for (int var_control = valor_inicial;  var_control < =  valor_limite; exp_decremento)
        sentencia
```

Ejemplo de formato ascendente:

```
for (int n = 1; n <= 10; n++)
        cout << n << “t” << n * n << endl;
```

Ejemplo de formato descendente:

```
for (int n = 10; n > 5; n --)
        cout << n << “ t “ << n * n << endl ;
```

PRECAUCIONES EN EL USO DEL FOR:

Un bucle for se debe de construir con gran precaución, asegurándose de que la expresión de inicialización, la condición del bucle y la expresión de incremento harán que la condición del bucle se convierta en falsa en algún momento. Por ejemplo la ejecución de la siguiente instrucción:

```
int limite = 1;
for (int n = 0; n <= limite; n++)
{
        cout << n << endl;
        limite++;
}
```

Produce una secuencia infinita de enteros, ya que a cada iteración la expresión límite++ se incrementa limite en 1. A consecuencia de ello, la condición del bucle i = limite siempre es cierta y se producirá la siguiente salida.

0
1
2
.
.
.

4.16 SENTENCIAS BREAK Y CONTINUE:

Las sentencias break y continue terminan la ejecución de un bucle while de modo similar a los otros bucles.

EJEMPLO 4.25

```
// Listados
// Sentencias bucles for vacios

#include <iostream.h>

int main ( )
{
        int contador = 0;
        int maximo;
        cout << “cuantos holas”;
        cin>>maximo;

        for (;;)                    // bucle que no termina nunca
        {
        if (contador < maximo)
        {
        cout<<”Hola!”;
        contador++;
        }
        else
                break;
}
return 0;
}
```

La salida que se visualizara será la siguiente:

Hola!
Hola!
Hola!

4.17 REPETICIÓN EL BUCLE DO_ WHILE.

La sentencia do_while se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener de que una determinada acción se ejecuta una o varias veces , pero al menos una vez.

Sintaxis

Acción(sentencia) a ejecutar Almenas una vez

expresiones lógicas que determinan si la acción se repite

1.

Do sentencia while (expresión)

2.

Do Sentencia While (expresión)

La construcción do comienza ejecutando sentencia. Se evalúa a continuación expresión. expresión es verdadera, entonces se repite la ejecución de sentencia. Este proceso continua hasta que la expresión es falsa .

EJEMPLO 4.26

```
do
{
cout<<introduzca un digito (0-9):";
cin>>digito;
}while ((digito < ' 0 ') | | ( ' 9 '< digito)) ;
```

Este bucle se realiza mientras se introduce dígitos y se termina cuando se introduzca un caracter que no sea un digito de ' 0 ' a ' 9 '.

EJEMPLO 4.27

Aplicación simple de un bucle while: seleccionar una opción de saludo al usuario dentro de su programa.

```
# include < iostream.h >
int main  ( )
{
char opcion;
do
{
```

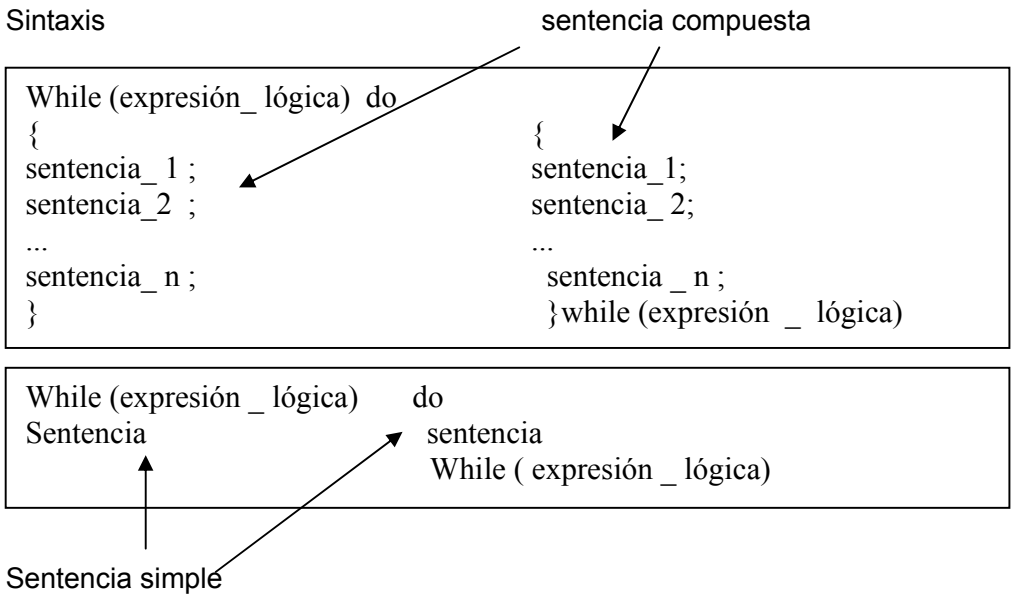
```
cout << ” hola ” <<  end ;
cout  << “ deseas otro tipo de saludo \n “ ;
cin  >> opcion ;
}while (opcion      ‘ s ‘ || opcion == ‘S ‘;
cout << “ adios \ n “ ;
return 0 ;
}
```

salida de muestra

hola
¿deseas otro tipo de saludo?
Pulse s para si y n para no
Y continuación pulse intro: s hola
¿ desea otro tipo de saludo?
Pulse s para si y n para no
Y a continuación pulse intro: n
Adiós

Diferencia entre while y do_ while

una sentencia do while es similar a una sentencia while, excepto que el cuerpo del bucle se ejecuta siempre al menos una vez.



EJEMPLO 4.28

```
/ / cuenta a 10
int x = 0 ;
cout  < < “x : “ << x++ ;
while (x < 10)
```

EJEMPLO 4.29

Visualizar la potencias de 2 cuerpos cuyos valores están en el rango 1 a 1.000.

```
/ / ejercicio con while
potencia = 1 ;
while ( potencia < 1000)
{
cout << potencia << end;
potencia * = 2
} / / fin de while
```

```
/ / ejercicio con do – while
potencia = 1 ;
{
cout << potencia << endl;
potencia * = 2 ;
} while ( potencia < 1000);
```

Comparación de bucles while, for y do- while

C++ proporciona 3 sentencias para el control de bucles: while, for y do while. El bucle while se repite mientras su condición de repetición del bucle es verdadera; el bucle for se utiliza normalmente cuando el conteo este implicado, o bien el control del bucle for en donde el numero de intenciones requeridas de seguir el numero de veces en los que un seguidor en particular tiene lugar . el bucle do while se ejecuta de un modo similar a while excepto que las sentencias del cuerpo del bucle se ejecutan al menos una vez.

En C++, el bucle for es el mas frecuentemente utilizado de los tres. Es relativamente fácil rescribir un bucle do- wuile como un bucle while, insertando una asignación inicial de la variable condicional. Sin embargo no todos los bucles while se pueden expresar de modo adecuado como bucles do- while, ya que un bucle do- while se ejecutara siempre amenos una vez y el bucle while puede no ejecutarse. Por esta razón un bucle while suele preferirse a un bucle do while a menos que esta claro que se debe de ejecutar una interacción como mínimo.

Comparación de tres bucles

```
cuenta =  valor – inicial

while ( cuenta < valor _ parada )
{
...
cuenta ++;
} / / fin de while

for  ( cuenta = valor _ inicial ; cuenta <valor _ parada ; cuenta ++)
{
```

```
...
} / / fin de for
cuenta = a valor  _ inicial;
if ( valor _ inicial < parada )
do
{
...
cuenta ++;
} while  ( cuenta  < valor _ parada );
```

Diseño de bucles

El diseño de un bucle necesita diseñar tres cosas :
1: el cuerpo de un bucle
2. las sentencias de inicialización.
3.las condiciones para la terminación del bucle .

Bucle para diseño de sumas y productos

Muchas tareas frecuentes implican la lectura de una lista de números y calculan su suma. Si se conoce cuantos numeraos habrá, tal tarea se puede ejecutar fácilmente por el siguiente seudo código. El valor de la variable total es el numero de números que se suman. La suma se acumula en la variable suma.

```
Suma = 0
Repetir lo siguiente total veces:
cin>> siguiente ;
Suma  = Suma + siguiente ;
Fin_ bucle
```

Este código se implementa fácilmente con un bucle for .

```
int suma  = 0;
for  ( int cuenta = 1 ; cuenta <= total ; cuenta + + )
{
cin >> siguiente;
Suma = Suma + siguiente;
}
```

obsérvese que la variable suma se espera tome un valor cuando se ejecuta la siguiente sentencia.

```
Suma = Suma  + siguiente;
```

Dado que suma debe tener un valor la primera vez que la sentencia se ejecuta , suma debe estar inicializada a algún valor antes de que se ejecute el bucle. Con el objeto de determinar el valor correcto de inicialización de suma se debe pensar sobre que sucede después de una iteración de bucle. Despides de añadir el primer numero, el valor de suma debe ser ese numero. Esto es , la primera ves que se ejecute el bucle el valor de sume + siguiente sea = a

siguiente. Para hacer esta operación trae el valor de suma debe ser inicializado a 0 . si en el lugar de suma, se sesea realizar productos de una lista de números , la técnica a utilizar es:

```
int producto = 1;
for  ( int cuenta = 1; cuenta < = total ; cuenta ++)
{
cin << siguiente;
producto = producto * siguiente ;
}
```

la variable producto debe tener un valor inicial. No se debe de suponer que todas las variables se deben inicializar a 0. si el producto se hiciera a 0 seguiría siendo 0 después de que el bucle anterior se terminara.

Fin del bucle

Existen cuatro métodos utilizados normalmente para terminar un bucle de entrada. Estos 4 métodos son :

1. lista encabezada por tamaño
2. preguntar antes de la interacción
3. lista terminada por un valor centinela
4. agotamiento de la entrada

Lista encabezada por el tamaño:

Si su programa puede determinar el tamaño de una lista de entrada por anticipado, bien preguntado al usuario o por algún otro mete , se puede usar un bucle repetir “n veces para leer la entrada exactamente n veces donde n es el tamaño de la lista.

Preguntar antes de la interacción

El segundo método de la interacción de un bucle de entrada es preguntar simplemente al usuario después de cada interacción de l bucle , si el bucle debe ser o no liberado de nuevo.

EJEMPLO 4.30

```
suma = 0;
cout<< “ ¿ existen números en la lista ?: /n”

<<” tecle s para si, n para no y final , intro) : “ ;
char resp;
cin>> resp ; while (( resp = = ‘ s ‘ )  || ( resp = = ‘ s ‘ ))
{
cout <<”introduzca un numero : “ ;
```

```
cin>>  numero ;

suma = suma + numero ;

cout << " ¿ existen mas números ? : / n" ;

<< " s para si , n para no . final con intro : ";

cin >> resp ;
}
```

Este método es muy tedioso para listas grandes de números. Cuando se lee una lista grande es preferible incluir una única señal de parada , como se incluye en el método siguiente .

Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es con un valor centinela. Un valor centinela es aquel que es totalmente distinto de todos los valores posibles de la lista que se esta leyendo y de este método sirve para indicar el final de la lista. Un ejemplo típico se presenta cuando se lee una lista de números positivos ; un numero – se puede usar como valor centinela para indicar el final de la lista.

EJEMPLO 4.31

```
/ /  ejemplo de valor centinela ( numero negativo)
...
cout << " introduzca una lista de enteros positivos " >> endl ;
<< " termina la lista con un numero negativo " << end ;
suma = 0 ;
cin ++ numero ;
white  ( numero > = 0 )
{
suma = suma + numero ;
cin  >>numero ;
}
cout << " la suma es : " < < suma ;
```

si al ejecutar el segmento del programa anterior se introduce la lista

4 8 15 -99

el valor de la suma será 27. es decir , -99, ultimo numero de entrada de datos no se añade a suma. – 99 es el ultimo dato de la lista que actúa como centinela y no forma parte de la lista de entrada de números.

Agotamiento de la entrada .

Cuando se leen entradas de un archivo, se puede usar un valor centinela. Aunque el método mas frecuente es comprobar simplemente las entradas del archivo se han leído y se alcanza el final del bucle cuando no hay mas entradas a leer este es método usual en la lectura de

archivos que suele usar una marca al final de archivo. En el capitulo de archivo se dedicara una tensión especial a la lectura de archivos con marca de final de archivo.

Otras técnicas de terminación de bucle

Las técnicas mas usuales para la terminación de bucle de otro tipo son :

1. bucles controlados por contador
2. preguntar antes de la iteración.
3. salir con una condición de bandera

Un **bucle controlado por contador** es cualquier bucle que determina el numero de iteraciones antes de que el bucle comience la continuación, repite (litera) el cuerpo del bucle es aliteraciones . la técnica de la lista encabezada por tamaño es un ejemplo de un bucle controlado por controlador la técnica de preguntar antes de iterar se puede usar para bucles distintos de los bucles de entrada, pero el uso mas común de esta técnica es para procesar la entrada . la técnica de va, lo centinela es una técnica conocida también como salida con una condición bandera o señalizadota. Una variable que cambia su valor para indicar que algún suceso o evento a tenido lugar se denomina normalmente bandera o indicador. El ejemplo anterior de suma de números, la variable bandera es número de modo que cuando toma un valor negativo significa que la lista de entrada a terminado

Bucles for vacíos

La sentencia nula (;) es una sentencia que esta en el cuerpo del bucle y no hace nada .

```
1.  . / /  listado
2.   : / /  demostración de la sentencia nula
3.  : / /  como cuerpo del bucle for
4.  .
5.  # include < iostream.h >
6.  : int main ( )
7.  {
8.  for ( int i = 0 ;  i <  < 5);  cout <<" i : " <<  i  ++  >> endl ;
9.  ;
10. return 0 ;
11.  }
```

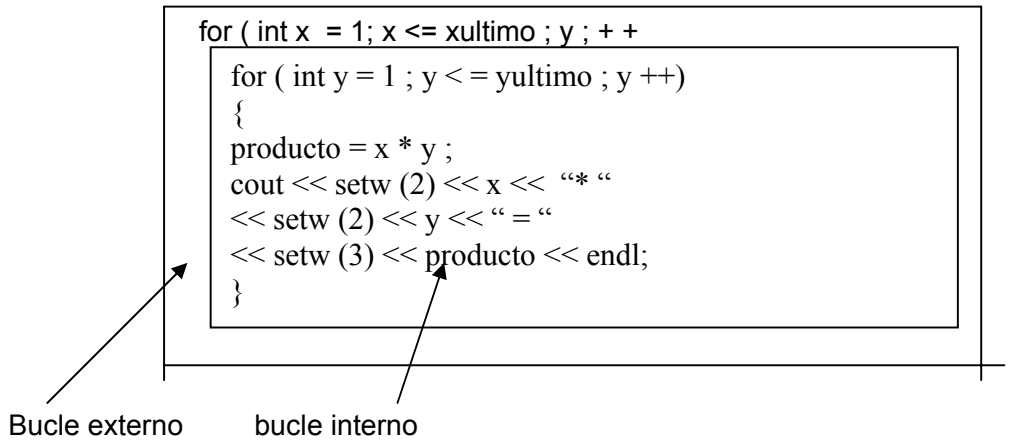
salida

```
i :  0
i :  1
i :  2
i :  3
i :  4
```

Bucles anidados

Es posible anidar bucles. Los bucles anidados constan de un bucle externo con uno o más bucles interno. Cada vez que se repite el bucle externo, los bucles internos se repiten se revalúan los componentes de control y se ejecutan todas las interacciones requeridas.
El segmento de programas siguiente visualiza una tabla de multiplicación por calculo y visualización de productos de la forma x * y para cada x en el rango de 1 a x ultimo y desde cada y en el rango 1 a y ultimo (donde x ultimo y y ultimo son enteros prefijados). La tabla que se desea obtener es:

```
1 * 1 =  1
1*2 =    2
1*3=     3
1*4=     4
1*5=     5
2*1=     2
2*2=     4
2*3=     6
2*4=     8
2*5=    10
```



El bucle que tiene x como variable de control se denomina bucle externo y el bucle que tiene y como variable de control se denomina bucle interno.

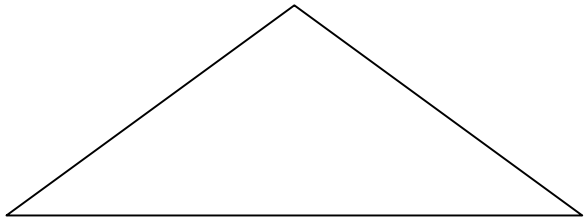
EJEMPLO 4.32

```
/ /  Aplicación de bucles anidados

# include < iostream. h>                    / / necesario para cin y cout
# include < iomanip.h>                     / / necesario para setw

void main ()
{
/ / cabecera de impresión
cout < < setw (12) < < " i " endl ;
for ( int i =  o ;  i < 4 ; i + +)
{
cout<< " externo " < < setw ( 7 ) << end;
for ( int y = 0 ; y < 4 ; i ++)
}
```

la salida del programa es

```
                    i          j
externo             0
externo             1
interno                        0
externo             2
interno                        0
interno                        1
externo             3
interno                        0
interno                        1
interno                        2
```

EJEMPLO 4.33. Escribir un programa que visualice un triangulo isósceles.



El triangulo isósceles se realiza mediante un bucle externo y 2 bucles internos. El bucle externo se repite 5 veces (5 filas) el numero de repeticiones realizadas por bucles internos se basa en el valor de la variable fila. El primer bucle visualiza los espacios en blanco el segundo bucle interno visualiza 1 o mas asteriscos .

```
/ / archivo triangulo . cpp
# include < iostream . h >
void main  ( )
{
datos locales ...
const  int num _ lineas = 5;
cont  char blanco = ‘ , ‘ ;
const char asterisco = ‘ * ’ ;
/ /    comienzo de una nueva línea
 cout < < endl ;
/ /    dibujar cada línea: bucle externo
for  ( int blancos = num_ líneas- filas ; blancos  0;
blancos - cout << blanco ;
ford ( int cuenta _ has = 1; cuenta _ as < 2 ‘ fila ;
cout < < asterisco ;
/ / terminar línea
cout << endl ;
{ / /
fin del bucle externo
}
```

El bucle externo se repite 5 veces, 1 por línea o fila ; el numero de repeticiones ejecutadas por los bucles se basa en el valor de fila .
la primera fila consta de un asterisco y 4 blancos, la segunda fila consta de 3 blancos y 3 asteriscos y así sucesivamente la 5 fila tendrá 9 asteriscos (2 * 5 – 1).

4.18 FUNCIONES.

Una función es un mini programa dentro de un programa. Las funciones contienen varias sentencias bajo un solo nombre, que un programa puede utilizar una o más veces para ejecutar dicha sentencia. Las funciones ahorran espacio, reduciendo repeticiones y haciendo más fácil la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables. En otros lenguajes como BASIC o ensamblador se

denomina subrutinas; en Pascal, las funciones son equivalentes a funciones y procedimientos.

4.19 CONCEPTO DE FUNCION.

C++ se puede utilizar como el lenguaje de programación estructurada, también conocida como programación modular. Por esta razón para escribir un programa se divide este en varios módulos, en lugar de uno solo largo. El programa se divide en muchos módulos (rutinas pequeñas denominadas funciones), que producen muchos beneficios: aislar mejor los problemas, escribir programas correctos más rápido y producir programas que son más fáciles de mantener.

Así, pues, un programa C++ se compone de varias funciones, cada una de las cuales realiza una tarea principal. Por ejemplo, si esta escribiendo un programa que obtenga una lista de caracteres del teclado, los ordena alfabéticamente y los visualice a continuación en la pantalla, se puede escribir todas estas tareas en un único gran programa (función **main()**).

```
int main ()
{
        //Codigo C++ para obtener una lista de caracteres
        ...
        //Código C++ para alfabetizar los caracteres
        ...
        //Código C++ para visualizar la lista por orden alfabético
        ...
        return 0;
}
```

Sin embargo, este método no es correcto. El mejor medio para escribir un programa es escribir funciones independientes para cada tarea que haga el programa. El mejor medio para escribir el citado programa seria el siguiente:

```
int main ()
{
   obtener  caracteres ( ); // Llamada a una función que obtienen los
//números
   alfabetizar ();                  // Llamada la función que ordenan
                        //alfabéticamente las letras
   ver  letras ();                  //Llamada la función que visualiza
                        // letras en la pantalla
   return ( );              // retorno a DOS


void  obtener caracteres ( )
```

```
{
   //...
   // código de C++ para obtener una lista de caracteres
   return 0;                //Retorno a main ( )
}
void  alfabetizar ( )
{
   //...
   // código de C++ para alfabetizar los caracteres
  //...


         return 0;                // Retorno a main ( )
   }
   void   ver letras ( )
   {
         //...
         // codigo de C++ para visualizar lista alfabetizada
         //...

         return  0;               // Retorno a main( )
   }
```

Cada función realiza una determinada tarea y cuando se ejecuta return se retorna al punto en que fue llamado por el programa o función principal.

4.20 ESTRUCTURA DE UNA FUNCION.

Una función es, sencillamente, un conjunto de sentencias que se pueden llamar desde cualquier parte de un programa. Las funciones permiten al programador un grado de abstracción en la resolución de un problema.
Las funciones no se pueden anidar. Esto significa que una función no se puede declarar dentro de una función. La razón para esto es permitir un acceso muy eficiente a los datos. En C++ todas las funciones son externas o globales, es, decir, pueden ser llamadas desde cualquier punto del programa.
La estructura de una función en C++ se muestra figura siguiente:

```
tipo_de_retorno nombre de Función  (lista de parámetros)
{
  cuerpo de la función
  return expresión
}
```

tipo_de_retorno tipo de valor devuelto por la función o la palabra reservada void si la función no devuelve ningún valor
nombreFunción identificador o nombre de la función
lista de Parámetros lista de declaraciones de los parámetros de la función separados por comas

expresión valor que devuelve la función.

Los aspectos más sobresalientes en el diseño de una función son:

- Tipo de resultado. Es el tipo de dato que devuelve la función C++ y aparece antes del nombre de la función.
- Lista de parámetros. Es una lista de parámetros tipificados (con tipos) que utilizan el formato siguiente:
  tipo1 parametro1, tipo2 parametro2,.....
- Cuerpo de la función. Se encierra entre llaves de apertura ({) y cierre (}). No hay punto y coma después de la llave de cierre.
- Paso de parámetro. Posterior mente se vera que el paso de parámetros en C++ se pude hacer por valor y por referencia.
- No se pueden declarar funciones anidadada.

- Declaración local. Las constantes tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- Valor devuelto por la función. Mediante la palabra reservada return se pude devolver el valor de la funcion.

EJEMPLO 4.34

Las funciones cuadrado ( ) y suma ( ) muestran dos ejemplos típicos de ellas.

```
// función que calcula los cuadros de números enteros
// sucesivos a partir de un numero dado (n), parámetro
// de la función, hasta obtener un cuadrado que sea
// mayor de 1000

int cuadrado (int  n)
{
int   cuadrado = 0;
while ( cuadrado <=1000)          //          el cuadrado ha de ser menor de 1000
          cuadrado = n*n;
          cout << “El cuadrado de:” << n << “ es ”  << cuadrado
                << endl ;
          n++;
   }
   return 0 ;
}

// Calcula la suma de un número dado (parámetro) de elementos de un
// array de datos

float suma (int num_elementos)
{
          int indice;
          float total = 0.0 ;
          for (indice = 0 ; indice <= num_elementos ; indice ++)
              total += Datos [indice];
          return total ;
}
```

4.21 NOMBRE DE UNA FUNCION

Un nombre de una función comienza con una letra o un subrayado (_) y puede contener tantas letras, números o subrayados como desee. El compilador ignora, sin embargo, a partir de una cantidad dada (32 en Borland / Turbo C++, 248 en Microsoft). C++ es sensible a mayúsculas, lo que significa que las letras mayúsculas y minúsculas son distintas a efectos del nombre de la funcion.

```
int max (int x, int y)                    //nombre de la función  max
double media (double x1, double x2)       // nombre de la función media
```

Tipo de datos de retorno

Si la función no devuelve un valor int, se debe especificar el tipo de dato devuelto (de retorno) por la función. El tipo debe de ser uno de los tipos simples de C++ , tales como int, char o float, o un puntero a cualquier tipo de C++, o un tipo struct.

```
int max (int x, int y )          // devuelve un tipo de int
double  media (double x1, double x2)      // devuelve un tipo double
float func0 () {...} //devuelve un float
char *func1() {...}                       // devuelve un puntero a char
int *func3() {...}                        // devuelve un puntero a int
char *func 4 () {...}                  // devuelve un puntero a un array char
int func 5 ( ) {...}                      //devuelve un  int   [es opcional]
```

Si una función no devuelve un resultado se puede utilizar el tipo void, que se considera como un tipo de dato especial.
Algunas declaraciones de funciones que devuelven distintos tipos de resultados son:

```
int calculo_ kilometraje (int litros, int kilómetros);
char mayúsculas (char car);
float DesvEst  (void);
struct Inforpesona BuscarRegistro (int num_ registro);
```

En C++ como se vera mas adelante, el uso de las palabras reservada struc es opcional para funciones que devuelven un tipo estructura. Se puede utilizar solamente el nombre de la estructura, como en la ultima declaración:
Muchas funciones no devuelven resultados. La razón es que se utiliza como subrutinas para realizar una tarea correcta. Una función que no devuelve un resultado, a veces se domina procedimiento. Para identificar al compilador que una función no devuelve resultado, se utiliza el tipo de retorno void, como en este ejemplo:

```
void visualizar resultados  (float Total, int num_ elementos);
```

Si se omite un tipo de retorno para una función como en

```
void VerResultados (float Total, int longitud)
```

El compilador supone que el tipo de datos devuelto es int. Aunque el uso de int es opcional, por razones de claridad y consistencia se recomienda su uso. Así, la función anterior se puede declarar también:

int VerResultados (float Total, int longitud)

Resultados de una función

Una función puede devolver un único valor. El resultado se muestra como una sentencia return cuyas sintaxis es:

```
return (expresión)

return (a+b+c);
return;
```

El valor devuelto (expresión) pude ser cualquier tipo de datos excepto una función o un array. Se puede ver valores múltiples devolviendo un puntero a una estructura o un array. El valor de retorno debe seguir las mismas reglas que se aplican a un operador designación. Por ejemplo, no se puede devolver un valor int si el tipo de retorno es un puntero. Sin embargo, si se devuelve un int el tipo de retorno es float, se realiza la conversión automáticamente.
Una función puede tener cualquier numero de sentencia return. Tan pronto como el programa encuentra cualquiera de las sentencias return, se retorna a la sentencia llamadora. La ejecución de una llamada a la función terminada si no se encuentra ninguna sentencia return; en este caso, la ejecución continua hacia la llave final del cuerpo de la función.
Si el tipo de retorno es void, la sentencia return se puede escribir como return; sin ninguna expresión de retorno, o bien, de modo alternativo, se puede omitir la sentencia return.

```
void func1 (void)
{
cout <<"Esta funcion no devuelve valores";
}
```

El valor devuelto se suele encerrar entre paréntesis, pero su uso es opcional. En algunos sistemas operativos, como DOS, se puede devolver un resultado al entorno llamador. Normalmente el valor 0 se devuelve en estos casos.

```
int main ()

{

cout <<"Hola mundo" << endl ;

return  0;

}
```

4.22 LLAMADA A UNA FUNCION.

Las funciones, para poder ser ejecutadas, han de ser llamadas o invocadas. Cualquier expresión puede contener una llamada a una función que redirija el control del programa a la función nombrada, Normalmente la llamada a una función se realizara desde la función principal **main (),** aunque naturalmente también podrá ser desde otra función.

La función llamada que recibe el control del programa se ejecuta desde el principio y cuando termina (se alcanza la sentencia return, o la llave de cierre (}) si se omite return) el control del programa vuelve y retorna a la función llamadora si no es main.

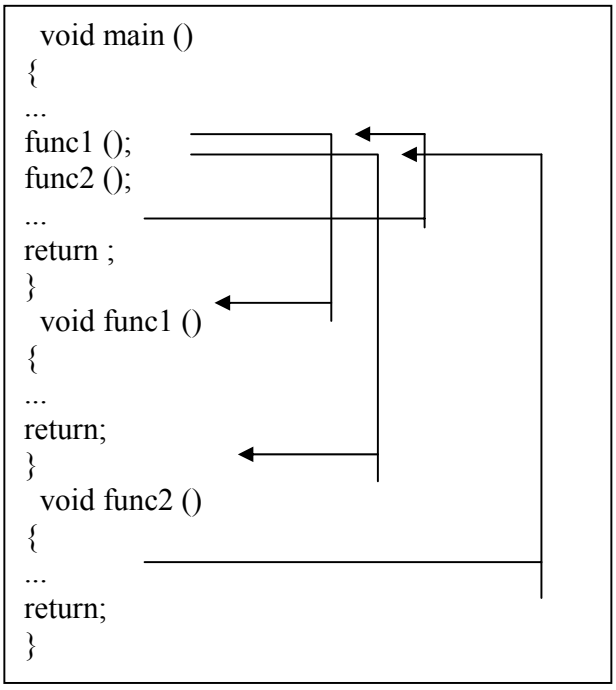


Figura: Traza de llamadas de funciones.

EJEMPLO 4.35

```
# include <iostream.h>
void func1 (void)
{
cout <<" Segunda funcion \n ";
return;
}
void func2 (void)
{
cout <<" Tercera funcion \n ";
return;
}
int main ()
```

```
{
cout << " Primera funcion llamada main () \n ";
func1 ( );        //        Segunda funcion llamada

func2 ( );        //        Tercera funcion llamada
cout << " main se termina ";
return 0;         //        Se devulve el control a DOS
}
```

La salida de es te programa es;

```
Primera función llamada main ( )
Segunda función
Tercera función
main termina
```

Se puede llamar a una función y no utilizar el valor que se devuelve. En el ejemplo

```
func ( );
```

el valor de retorno no se considera. El formato **func ( )** sin argumentos es el más simple. Para indicar que la llamada a una función no tiene argumentos si se sitúa una palabra reservada void entre paréntesis en la declaración de la función y posteriormente en lo que se denomina prototipo.

```
void main ( )
{
func  ( );
...
}
void func (void)
{
cout << " Hola mundo \n ";
}
```

EJEMPLO 4.36

La función max devulve el numero mayor de dos enteros.

```
int max (int x, int y)
{
if ( x < y)  return y;
else
return x;
}
void main ()
{
int m,n;
do {
cin  >> m >> n;
cout  << max (m,n)  << end1;          //Llamada a max
} while (m  !=0);
}
```

EJEMPLO 4.37

Calcular la media aritmética de dos números.

```
# include <iostream.h>
double media (double x1, double x2)
{
return (x1 + x2) / 2;
}
void main ()
{
double num1 , num2 , med;
cout << “Introducir dos números reales:”;
cin >> num1 >> num2;
med = media (num1, num2);
cout << “El valor medio es “ << med << end1;
}
```

4.23 PROTOTIPOS DE LAS FUNCIONES.

C++ requiere que una función se declare o defina antes de su uso. La declaración de una función se denomina prototipo. Los prototipos de una función contienen la misma cabecera de la función, con la diferencia de que los prototipos terminan con un punto y coma. Específicamente un prototipo consta de los siguientes elementos: nombre de la función, una lista de argumentos encerrados entre paréntesis y un punto y coma. La inclusión del nombre de los parámetros es opcional.

SINTAXIS.

tipo_retorno nombre_funcion (lista_de_declaración_parámetros);

tipo_retorno	tipo del valor devuelto por la funcion o palabras reservadas void si no devuelve un valor
nombre_funcion	nombre de la funcion
lista_declaración_parámetros	lista de declaración de los parámetros de la funcion, separados por comas (los nombres de los parámetros son opcionales, pero es buena practica incluirlos para indicar los parámetros que presentan)

Un prototipo declara una función y proporciona una información suficiente al compilador para verificar que la función esta siendo llamada correctamente, con respecto al numero y tipo de los parámetros y el tipo devuelto por la función. Es obligatorio poner un punto y coma al final del prototipo de la función con el objetivo de convertirlo en una sentencia.

```
double FahrACelsius (double temFahr);      //prototipo valido

int max (int , int);                       //prototipo valido
```

Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la primera funcion main (). El compilador utiliza los prototipos para validar que el numero y los tipos de datos de los argumentos reales de la llamada a la función son los mismos que el numero y el tipo de argumentos formales en la función llamada. Si se detecta una inconsistencia, se visualiza un mensaje de error. Sin prototipos, un error puede ocurrir sin un argumento con un tipo de datos incorrecto se pasa a una función. En programas más complejos, este tipo de errores son difícil de detectar.

En C++, la diferencia entre los conceptos declaración y definición es preciso tenerla clara. Cuando una entidad se declara, se proporciona un nombre y se lista sus características. Una definición proporciona un nombre de entidad y reserva espacio de memoria para esa entidad. Una definición indica que existe un lugar en un programa donde ' existe' realmente la entidad definida, mientras que una declaración es solo una indicación de que algo existe en alguna posición.

Una declaración de la función contiene solo la cabecera de la función debe existir en algún lugar del programa; por ejemplo, antes o después de main.

```
int max (int , int);
 // programa principal
void main ()
{
int m ,n;
do {
cin >> m >>n;
cout <<max (m , n) <<end1;
} while (m !=0);
}
//devulve el entero mayor
int max (int x int y)                    // Definicion de max
{
if (x ,y) return y ;
else return x;
}
```

```
# include <iostream.h>
double media (double x1, double x2);     // Declaracion de media

main ()
{
med = media (num1 , num2);
...
}
double media (double x1 , double x2)     // Definición
{
return (x1 + x2) /2;
}
```

DECLARACIONES DE UNA FUNCION

- Antes de que una función pueda ser invocada, debe ser declarada.
- Una declaración de una función contiene solo la cabecera de la función (llamado también prototipo).

tipo_resultado nombre (tipo1 param1, tipo2 param2,.....);

- Los nombres de los parámetros se pueden omitir.

Double mon (mod , mod);

Prototipos con un número no especificado de parámetros.

Un formato especial de prototipo es aquel que tiene un número no especificado de argumentos, que se representa por puntos suspensivos (…). Por ejemplo.

Int muestras (int a, …);
Int printf (constante char *formato, …);
Int scanf (constante char *formato, …) ;

4.24 PARÁMETROS DE UNA FUNCIÓN.

C++ proporciona dos métodos para pasar variables (parámetros) entre funciones. Una función puede usar parámetros por valor y parámetros por referencia, o puede no tener parámetros. Esta sección examina el mecanismo que C++ utiliza para pasar parámetros a funciones cómo optimizar el paso de parámetro, dependiendo del tipo de datos que se utiliza. Suponiendo que se tenga la declaración de una función circulo con tres argumentos-

void circulo (int x, int y, int diámetro);

Cuando se llama a circulo se deben pasar tres parámetros a esta función. En el punto de llamada cada parámetro puede ser una constante, una variable o una expresión, como en el siguiente ejemplo:

circulo (25, 40, vueltas*4);

Paso de parámetros por valor.

Paso por valor (también llamado paso por copia) significa que cuando C++ compila la función y el código que llama a la función, la función recibe una copia de los valores de los parámetros. Si se cambia el valor de un parámetro variable local, el cambio sólo afecta a la función y no tiene efecto fuera de la función.
La figura 6.3 muestra la acción de para un argumento por valor. La variable real “i” no se pasa, pero el valor de “i”, 6, se pasa ala función receptora.
En la técnica de paso de parámetro por valor, la función receptora no puede modificar la variable de la función (parámetro pasado).

NOTA: El método por defecto de pasar parámetros es por valor, a menor que se pasen arrays. Los arrays se pasan siempre por dirección.

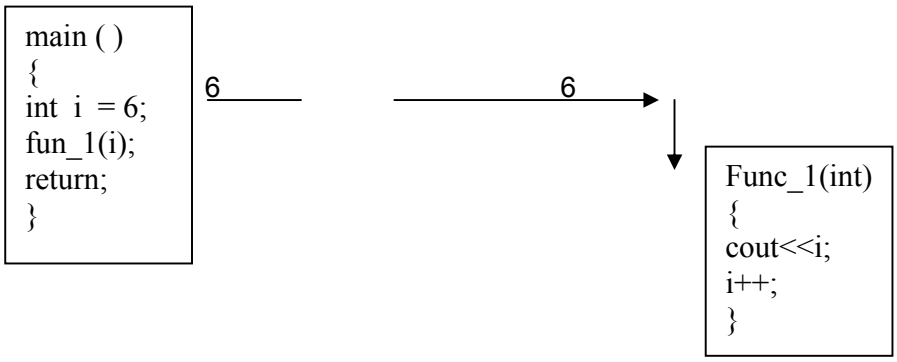


Figura: Paso de variables “i” por valor.

El listado PARVALOR. UNO muestra el mecanismo de paso de parámetros por valor.

```
//  PARVALOR.UNO
//  Muestra el paso de parámetros por valor
//  Se puede cambiar la variable del parámetro en la función pero su modificación
//  Pero su modificación no puede salir al exterior

#include<iostream.h>
void DemoLocal (int valor);
void main (void)
{
int n=10;
cout<< “antes de llamar a DemoLocal, n =*”<<n<<endl;
DemoLocal (n);
cout<<”después de llamada a DemoLocal, n*”<<n<<endl;
cou<”pulse intro (enter) para continuar”;
cin.get ( )
}
void DemoLocal (int valor)
{
cout<<”dentro de DemoLocal, valor = * << valor << endl;
valor = 999;
cout<<”dentro de DemoLocal, valor =* << valor <<endl;
}
```

Al ejecutar este programa se visualiza la salida:
Antes de llamar a DemoLocal, n= 10
Dentro de DemoLocal, valor = 10
Dentro de DemoLocal, = 999
Después de llamar a DemoLocal, n = 10

4.25 PASO DE PARAMETROS POR REFERENCIA.

Cuando una función debe modificarse el valor del parámetro pasado y devolver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por referencia o dirección.
En este método el compilador pasa la dirección de memoria del valor del parámetro a la función. Cuando se modifica el valor del parámetro (la variable local), este valor queda almacenado en la misma dirección de la memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado.
Para declarar una variable parámetro como paso por referencia, el símbolo "&" debe preceder al nombre de la variable.
C++ permite utilizar punteros para implementar parámetros por referencia. Este método es utilizado en "C" y se conserva en "C++" precisamente por cuestión de compatibilidad.

```
// método de paso por referencia, mediante punteros
// estilo C
void intercambio (int* a, int* b)
{
int aux = *a;
*a = *b;
*b = aux;
}
```

La función intercambio ( ) utiliza las expresiones *a y *b para acceder a los enteros referenciados por las direcciones "int" de las variables "i" y "j" en la llamada de prueba siguiente:

```
it i = 3, j= 50
cout << "i = "<< i << "y j = "<< j << endl;
intercambio (&I, &j);
cout <<"I = "<< i << "y j =" << j << endl;
```

La llamada a la función intercambio ( ) debe pasar la dirección de las variables intercambiadas. La versión de la función intercambio ( ) que utiliza parámetro referencia es:

```
void intercambio (int m, int& n)
{
int aux = m;
m = n;
n = aux;
{
```

Los parámetros por referencia. Por consiguiente, cualquier cambio efectuado en el interior de la función se transmitirá al exterior de la misma. Para llamar a la función intercambio ( ) por referencia, simplemente se pasan las variables a intercambiar, "i" y "J".

```
int i = 3, j = 50;
cout << "i = "<< I <<"y   j ="<<j<< endl;
intercambio (i, j);
cout <<"I = "<< I << "y   j= "<<j<< endl;
```

Diferencia entre los parámetros por valor y por referencia.

Las reglas que se han de seguir cuando se utilizan parámetros valor y referencia son las siguientes:

- Los parámetros valor (declarados sin &) reciben copias de los valores de los argumentos que se les pasan;
- La asignación a parámetros valor de una función nunca cambia el valor del argumento original pasado a los parámetros;
- Los parámetros referencia (declarados con &) reciben la dirección de los argumentos pasados;
- En una función, las asignaciones a parámetros referencia cambian los valores de los argumentos pasados.

Por ejemplo, la escritura de una función intercambio ( ) para intercambiar los contenidos de dos variables, requiere que los datos puedan ser modificados.

Llamada por valor

```
int uno, dos;

litercambio 1
(int primero, int segundo)
{
……..
}
```

Llamada por referencia

```
int uno, dos;

intercambio 2
(int& primero, int& segundo)
{
……..
}
```

Sólo en el intercambio 2 los valores de uno y dos se cambiarán. Veamos una aplicación completa de ambas funciones:

EJEMPLO 4.38

```
#include<iostream.h>
void intercambio 1 (int, int);
void intercambio 2 (int&, int&);
void main ( )
{
int a, b;
a= 10; b= 20;
intercambio 1 (a, b);
cout << a << b << endl;
}
void intercambio 1 (int primero, int segundo)
{
int aux;
aux = primero;
primero = segundo;
segundo = aux;
}
```

```
void intercambio 2 (int& primero, int& segundo)
{
int aux;
aux = primero;
primero = segundo;
segundo = aux;
}
```

La ejecución del programa producirá:

```
10   20
20   10
```

Nota: Todos los parámetros en C sé pasan por valor, C no tiene equivalente al parámetro referencia en C++.

Parámetros const de una función.

Con el objeto de añadir seguridad adicional a las funciones, se pueden añadir a una descripción de un parámetro el especificador const, que indica al compilador que sólo es de lectura en el interior de la función. Si se intenta escribir en este parámetro se producirá un mensaje de error de compilación.

```
void falso (const int item, const char& car)
{
item = 123;       // fallo en tiempo de compilación
car = 'A'         // fallo en tiempo de compilación
}
```

La tabla 6.1 muestra un resumen del comportamiento de los diferentes tipos de parámetros.

Parámetro Especificado Como:	**Item Pasado por**	**Cambia item Dentro de La función**	**Modificar Parámetros Al exterior**
Itm item	valor	si	No
Const int item	valor	no	no
Int& item	referencia	si	si
Const int& item	referencia	no	no

4.26 ARGUMENTOS POR OMISIÓN.

Cuando una función tiene un cierto número de parámetro, normalmente el mismo número de argumento deben indicarse cuando se llama a al función.
En C++, sin embargo, es posible omitir algún argumento.
Una característica poderosa de la función C++ es que en ella pueden establecer valores por omisión o ausencia (<<por defecto>>) para los parámetros. Se pueden asignar argumentos por defecto a los parámetros de una función. Cuan do se omite el argumento de un parámetro

que es un argumento por defecto, se utiliza automáticamente éste. La única restricción es que se deben incluir todas las variables desde la izquierda hasta el primer parámetro omitido. Si se pasan valores a los argumentos

omitidos, se utiliza ese valor; si no se pasa un valor a un parámetro opcional, se utiliza el valor por defecto como argumento. El valor por defecto debe ser una expresión constante.

El ejemplo siguiente muestra cómo utilizar argumentos por defecto, en una función cuyo prototipo es:

void asteriscos (int fila, int col, int num, char c = '*') ;

Se puede llamar a la función asterisco de dos formas equivalentes;

Asteriscos (4, 0, 40);

O bien

Asteriscos (4, 0, 40,'*');

Sin embargo, si sé cambiar el carácter utilizada por la función, se puede escribir

Asteriscos (4, 0, 40, '#');

En donde el argumento explícito (#) anula el carácter por omisión (*).

Otro ejemplo es la función funcdef ( ).

Char funcdef (int arg 1 = 1, char c = 'A', flota f_val = 45.7f);

Se puede llamar a funcdef con cualquiera de las siguientes sentencias:

```
funcdef (9, 'Z', 91.5);    //anula los tres argumentos por defecto
funcdef (25, ´W');         //anula los dos primero argumentos
funcdef (50);              //anular el primer argumento
funcdef ( );               //utiliza los tres argumentos por defecto
```

Las sentencias anteriores son equivalentes a:

```
funcdef (9, 'Z', 91.5);
funcdef (25, 'W', 45.7);
funcdef (50, 'A', 45.7);
funcdef (1, 'A', 45.7);
```

Sin embargo, no se puede omitir un argumento a menos que se omita todos los argumentos a su derecha. Por ejemplo, la siguiente llamada a la función no es correcta:

funcdef (, 'Z', 99.99);
Se debe tener cuidado y situar cualquier argumento que tenga valores por defecto a la derecha de una función.

```
f ( )
f (a);
```

```
f (a, b);
f(a, b, c);
```

La función escribir_car tiene dos parámetros. El primero indica el carácter a escribir (visualizar) y el segundo indica él numero de veces que ese carácter debe escribirse.

```
void escribir_car (char c, int num = 1)
{
for (int i = 1; i<= num; 1++)
cout <<c;
}
```

Al parámetro num se le ha dado el valor por defecto de 1 se utilizará automáticamente si se omite el argumento correspondiente en una llamada. Si no se omite el argumento, se utilizará ese valor. Llamadas validas son:

```
Escribir_car ('x', 4)        // se escribe cuatro 'x'
Escribir_car  ('y');         // se escribe una 'y'
```

El programa ARGDEFEC.CPP muestra cómo asignar valores por omisión a variables parámetros.

```
//ARGDEFEC.CPP
// Muestra el uso de valores de parámetros por defecto

#include <<iostream.h>
void f(int a=10, int b=20, int c=30)
{
cout <<"a="<<a<<endl
      <<"b="<<b<<endl
      <<"c="<<c<<endl;
}
void main (void)
{
f ( )
f (1)
f (1, 5)
f (1, 2, 3);
cout <<"pulse intro (Enter) para continuar";
cin.get ( );
}
```

Reglas de construcción de argumentos por defecto:

- Los argumentos por defecto se deben pasar por valor. Un nombre de un argumento por defecto no se puede pasar por referencia.
- Los valores de los argumentos por defecto pueden ser valores literales o definiciones const. No pueden ser variables. En otras palabras, si redeclara int n y a continuación int x=n, se rechazará n como parámetro por defecto. Sin embargo, si n se declara con, const int n=1 entonces sé acepta la declaración.

- Todos los argumentos por defecto deben colocarse al final en el prototipo de la función. Después del primer argumento por defecto, todos los argumentos posteriores deben incluir también valores por defecto.

Sea la función

void f (int anchura, flota v = 3.14159, char x = '*') ;

Las llamadas siguientes son laterales:

F (10);
F (10, 7.5);
F (10, 7.5, '$');

La llamada siguiente no es legal:

F (10, , '^') ;

Ya que se ha intentado saltar uno de los parámetros por defecto, otro ejemplo puede ser la función h ( ):

void h(int a = 1, int b = 2, int c = 3, iny d = 4);

Y llamadas validas son:

H ( );
H (9);
H (9, 18);
H (9, 18, 25); h (9, 18, 25, 4);

Y llamadas no validas:

H (9, , , 4);

EJEMPLO 4.39

```
void escribir_car (char c, int num = 1, bool lineas_indep = false)
{
for (int i = 1; i <num; i ++)
{
cout <<c;
if (lineas_indep)
cout <<endl;
}
}
```

Algunas llamadas validas son:

```
Escribir_car ('x');                 // se escribe una x
Escribe_car ('#', 5);               // se escribe 5 * en la misma línea
Escribir_car ('%', 5, trae);        // cinco % en líneas independientes
```

4.27 FUNCIONES EN LÍNEA (INLINE)

Cuando el compilador construye el código ejecutable de un programa C++ existen dos opciones disponibles para la generación del código de las funciones: funciones en línea y fuera de línea.

Las funciones en línea (**inline**) sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función se utiliza muchas veces en el programa y su código es pequeño. Existe una diferencia grande en el comportamiento en ambos tipos de funciones.

Una función normal es un bloque de código que se llama desde otra función. El copilador genera código para situar la dirección de retorno en la pila. La dirección de retorno es la dirección de la sentencia que sigue a la instrucción que llama a la función. A continuación, el compilador genera códigos que sitúan cualquier argumento de la función en la pila a medida que se requiera. Por último, el compilador genera una instrucción de llamada que transfiere el control a la función.

Para una función en línea (in line9, el compilador inserta realmente el código para la función en el punto en que se llama la función. Esta acción hace que el programa se ejecute más rápidamente, ya que no ha de ejecutar el código asociado con la llamada a la función.

Sin embargo, cada instancia de la función en línea puede requerir tanta memoria como se requiera para contener la función completa. Por esta razón, el programa incrementa su tamaño, aunque es mucho más rápido en su ejecución. Si se llama a la función en línea diez veces en un programa, el compilador inserta diez copias de la función en le programa. Sí la función

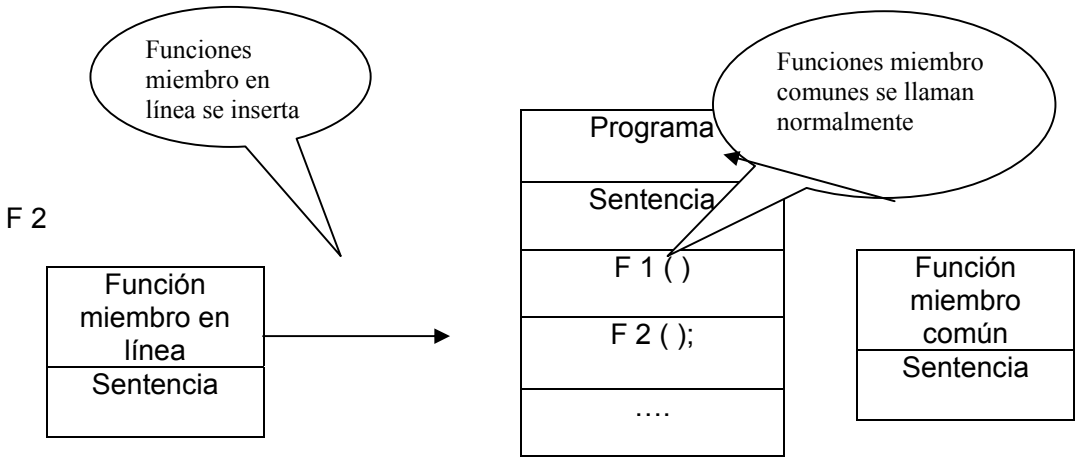


FIGURA: Código generado por una función fuera de línea.
Ocupa 1K, el tamaño de su programa se incrementa en 10K (10.240 bytes).
Por el contrario, si se llama a la misma función con una función normal (no inline) diez veces, y el codigo de llamada suplementario es 25 bytes, la función requiere sólo 1.274 bytes (1K+25*10,1024+250,1.274):

A continuación se ilustra la sintaxis general de una función inline.

Regla: Comience declarando las funciones en - línea como funciones ordinarias cuando desarrolle sus programas. Las funciones en línea son más fáciles de depurar. Una vez que su programa esté funcionando, inserte la palabra reservada inline donde sea necesario.

Código generado por una función en línea.

La tabla siguiente resume las ventajas y desventajas de situar un código de una función en línea o fuera de línea.

TABLA Ventajas y desventajas de la función en línea.

	Ventajas	**Desventajas**
Funciones en línea	Rápida de ejecutar	Tamaño de código grande
Funciones fuera de línea	Pequeño tamaño de código	Lenta de ejecución

4.28 ARREGLOS

Un arreglo (lista o tabla es una esencia de objetos del mismo tipo. Los objetos sec llaman elementos del arreglo y se numeran consecutivamente 0,1,2,3.....el tipo de elementos almacenados en el arreglo pueden ser de cualquier tipo de dato de C++, incluyendo estructuras definidas por el usuario, como se describirá más tarde. Normalmente el arreglo se utiliza para almacenar tipos tales como char , int, o float

Un arreglo puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de personas que residen en cada una de las diecisiete comunidades autónomas españolas. Cada ítem del arreglo se denomina elemento.
Los elementos de un arreglo se numeran, como ya se ha comentado, consecutivamente 0,1,2,3... estos números se denominan valores índice o subíndice del arreglo. El termino subíndice se utiliza ya que se especifica igual que en matemáticas como una secuencia tal comoa_0, a_1, a_2.....estos números localizan la posición del elemento dentro del arreglo, proporcionando acceso directo al arreglo.
Si el nombre del arreglo es a, entonces a[0] es el nombre del elemento que esta en la posición 0, a[1] es el nombre del elemento que esta en la posición 1, etc. En general el elemento i-enésimo está en la posición i-l. De modo que si el arreglo tiene n elementos, sus nombres son a [0], a[1],.....a[n-1]. Gráficamente se representa así el arreglo a con seis elementos.
El arreglo a tiene seis elementos: a[0] contiene 25.1, a[1] contiene 34,2, a[2] contiene 5,25, a[3] contiene 7,45, a[4] contiene, a[4] contiene 6,09 y a[5] contiene 7,54
El diagrama representa una región de la memoria de la computadora ya que un arreglo se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua.

a

25,1	34,2	5.25	7.45	6.09	7.54

0 1 2 3 4 5

arreglo de seis elementos

4.29 DECLARACION DE UN ARREGLO:

Al igual que en cualquier tipo de variable, se debe declarar un array antes de utilizarlo. Un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar el compilador el tamaño o longitud del array. Para indicar el compilador el tamaño o longitud del array se debe hacer seguir al nombre, el tamaño encerrado entre corchetes. La sintaxis para declarar un array de una dimensión determinada es:

Tipo nombre array [numero de elementos];

Por ejemplo parta crear un array (lista) de diez variables enteras, se escribe:
int numeros[10];
Esta declaración hace que el compilador reserve espacio suficiente para contener diez valores enteros. En C++ los enteros ocupan, normalmente, 2 bytes de memoria la figura muestra el esquema de un array de diez elementos; cada elemento puede tener su propio

Array de datos enteros: a

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

Almacenamiento de un array en memoria

Un array de datos enteros se almacena en bytes consecutivos de memoria. Cada elemento utiliza dos bytes. Se accede a cada elemento de array mediante un índice que comienza en cero. Así, el elemento quinto (a[4]] del array ocupa los bytes 9' y 10'..

Se puede acceder a cada elemento del array utilizando un índice en el array. Por ejemplo,

cout<< numeros [4] endl;

Visualiza el valor del elemento 5 del array. Los arrays siempre comienzan en el elemento 0. así pues, el array *numeros* contiene los siguientes elementos individuales:

numeros[0] numeros[1] numeros[2] numeros[3]

numeros[4] numeros[5] numeros[6] numeros[7]
numeros[8] numeros[9]

precaución: C++ no comprueba que los índices del array están dentro del rango definido. Asi, por ejemplo se puede intentar acceder a *números [12]* y el compilador no producirá ningún error, lo que puede producir un fallo en su programa, dependiendo del contexto en que se encuentre el error.

4.30 SUBINDICES DE UN ARREGLO:

El índice de un array se denomina con frecuencia, *subíndice del array*. Él termino procede de las matemáticas, en las que un subíndice se utiliza para representar un elemento determinado.

$numeros_0$ equivale a números [0]

$numeros_3$ equivale a números [3]

El método de numeración del elemento *i-enésimo* con el índice o subíndice *i-l* se denomina *indexación basada en cero.* Su uso tiene el efecto de que el índice de un elemento del array es siempre el mismo que el número de <<pasos>> desde el elemento inicial a[0] a ese elemento. Por ejemplo, a[3] esta a tres pasos o posiciones del elemento a[0]. La ventaja de este métodos se vera de modo más evidente al tratar las relaciones entre arrays y punteros.

EJEMPLO 4.40

int edad[5]; array edad contiene 5 elementos: el primero, edad [0] y el ultimo edad [4].

int pesos[25], longitudes[100] ; declara 2 arrays de enteros.

float salarios[25] ; declara un array de 25 elementos float.

double temperatura[50]; declara un array de 50 elementos double.

char letras [15]; declara un array de caracteres.

En los programas se pueden referenciar elementos utilizando formulas para los subíndices. Mientras que el subíndice puede evaluar a un entero, se puede utilizar una constante, una variable o una expresión para el subíndice. Así. Algunas referencias individuales son:

edad[4]

ventas[total +5]

bonos[mes]

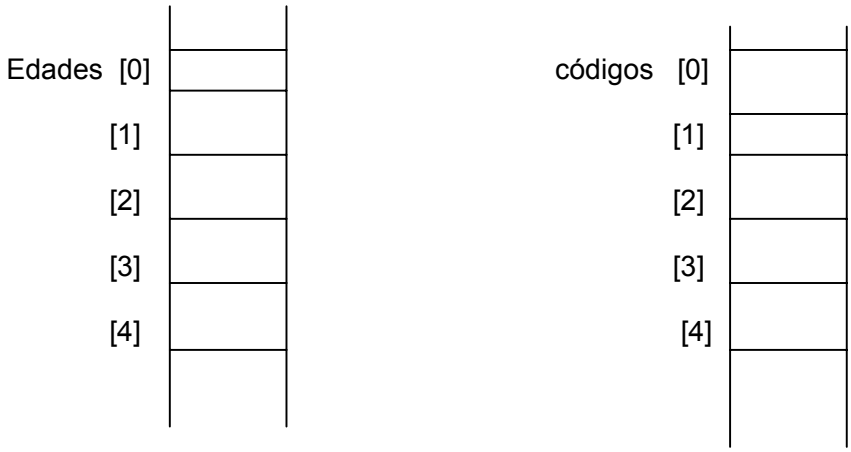
salario [mes [i] *5]

4.31 ALMACENAMIENTO EN MEMORIA DE LOS ARREGLOS

Los elementos de los arrays se almacenan en bloques contiguos. Así, por ejemplo, los arrays.

```
int edades[5];
char  códigos[5];
```

Se representan gráficamente en memoria.



4.32 EL TAMAÑO DE LOS ARREGLOS (sizeof)

La función sizeof ( ) devuelve el numero de bytes necesarios para contener su argumento. Si se usa sizeof() para solicitar el tamaño de un array, esta función devuelve el numero de bytes reservados para el array completo.
Por ejemplo, supongamos que se declara un array de enteros de 100 elementos denominados edades; si se desea producir el tamaño del array, se puede utilizar una sentencia similar a:
n=sizeof(edades);

donde *n* tomara el valor 200. si se desea solicitar el tamaño de un elemento individual del array, tal como

n=sizeof (edades [6]);

n almacenara el valor 2 (numero de bytes que contienen un entero).

4.33 VERIFICACION DEL RANGO DEL INDICE DE UN ARREGLO

C++ al contrario que otros lenguajes de programación-por ejemplo, pascal-no verifica el valor del índice de la variable que representa el array, así, por ejemplo, en pascal si se define un array m con índices 0 a 5, entonces a[6] hará que el programa se <<rompa>> (<<produzca un error>>).

EJEMPLO 4.41

Protección frente a errores en el intervalo(rango) de valores de una variable de índice que representa un array.

```
double suma (const double a [ ], const   int  n)
{
if (n  *  sizeo f ( double ) >  sizeof  ( a )
    return 0 ;
double S = 0.0 ;
for ( int  i =0 ; i < n ;  i++)
       S+ =  a [ i ];
return  S;
}
```

4.34 INICIALIZACION DE UN ARREGLO

Se deben asignar valores a los elementos del array antes de utilizarlos, tal como se asignan valores a variables. Para asignar valores a cada elemento del array de enteros precios, se puede escribir:

```
Precios[0]=10;
Precios[1]=20;
Precios[2]=30;
Precios[3]=40;
....
```

la primera sentencia fija precios[0] al valor 10, precios[1] al valor 20, etc. Sin embargo, este método no es practico cuando el array contiene muchos elementos. El método utilizado normalmente es *inicializar* (iniciar) el array completo en una sola sentencia.
Cuando se inicializa un array, el tamaño del array puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos.

```
int numeros[6]= {10, 20, 30 ,40, 60};
int n[ ] = {3, 4, 5}                    //declara un array de 3 elementos
char  c[  ] {`L´, ´u´, ´i´, ´s´};       //declara un array de 4 elementos
```

El array números tiene 6 elementos, n tiene 3 elementos y el array c tiene 4 elementos. Los arrays de caracteres se pueden inicializar con una constante de cadena, como en.

```
char s [ ] ={“mortimer”};
```

Nota: C++ puede dejar los corchetes vacíos, solo cuando se asignan valores al array, tal como
int cuenta [ ] = {15, 25, -45, 0,50};
El compilador asigna automáticamente cinco elementos a cuenta.

El método de inicializar arrays mediante valores constantes después de su definición es adecuado cuando el número de elementos del array es pequeño. Por ejemplo, para inicializar un array (lista) de 10 enteros a los valores 10 a 1, y a continuación visualizar dichos valores en un orden inverso, se puede escribir:

```
int cuenta[10]= {10, 9, 8, 7, 6,, 5, 4, 3, 2, 1, 0};
for  (i = 9;  i >=1; i-)
        cout<<"\ n cuenta descendente" <<i<< "="  <<cuenta[i];
```

se puede asignar constantes simbólicas como valores numéricos, de modo que las sentencias siguientes son validas:

```
const int ENE= 31, FEB=28, MAR=31, ABR=30, MAY=31, JUN=30, JUL=31, AGO=31,
SEP=30,OCT=31, NOV= 30, DIC= 31;

……
int mes[12]= { ENE, FEB,  MAR, ABR, MAY, JUN, JUL, AGO, SEP,OCT, NOV, DIC};
```

se pueden asignar valores a un array utilizando un bucle for o while /do-while, y este suele ser el sistema más empleado normalmente. Por ejemplo para inicializar todos los valores del array números el valor 0, se puede utilizar la siguiente sentencia:

```
for (i= 0; i<= 5; i++)
     números[I ]= 0;
```

Debido a que el valor del subíndice i varia de 0 a 5, cada elemento del array números se inicializa y se establece a cero.

EJEMPLO 4.42

El programa siguiente asigna ocho enteros a un array nums, mediante cin; a continuación, visualiza el total de los números.

```
#include<iostream.h>

const NUM 8;

main ( )

{

int nums[NUM];

int total= 0;

for (int j = 0; j < NUM;  j++)

cout<< "por favor, introduzca el número";

cin>>nums [  j ];

total + = nums [ j ];
}

cout<<  "el total de números es" <<total<<endl;
```

```
return 0;
}
```

4.35 ARREGLO DE CARACTERES Y CADENAS DE TEXTO

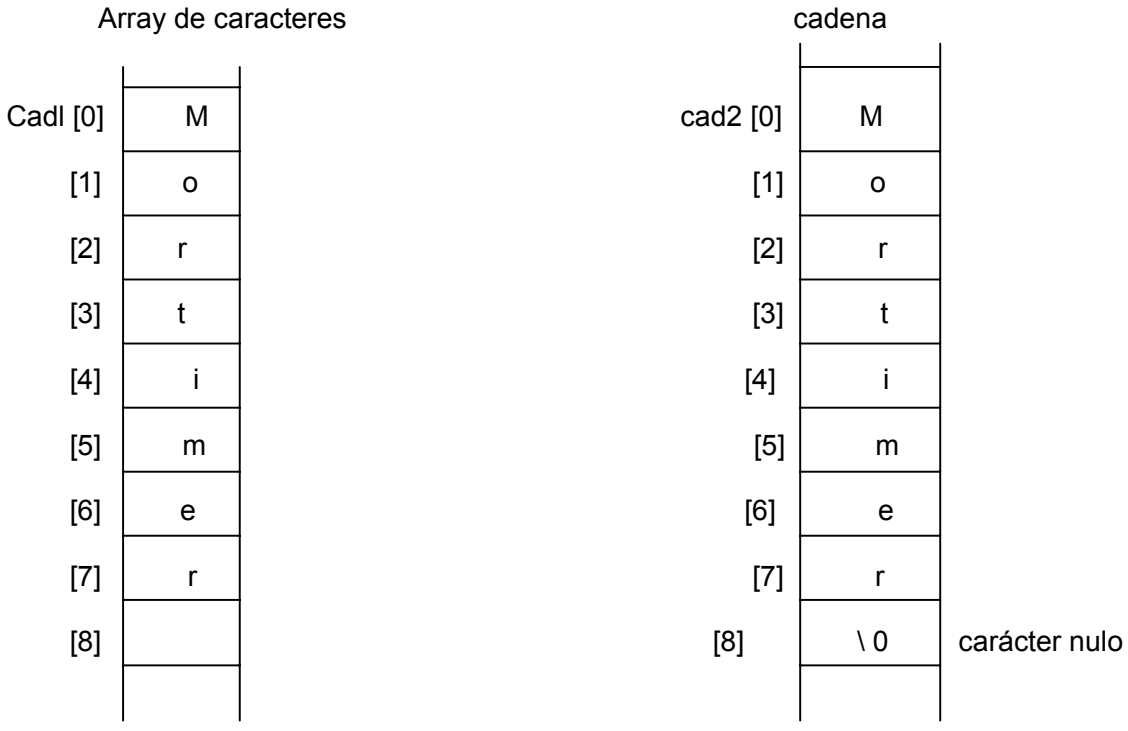
Una cadena de texto es un conjunto de caracteres, tales como <<ABCDEFG>>.
C++ soporta cadenas de texto utilizando un array de caracteres que contenga una secuencia de caracteres:

***c**har cadena [ ] =”*ABCDEFG”;

Es importante comprender la *diferencia* entre un array de caracteres y una cadena de caracteres. Las *cadenas* contienen un carácter nulo al final del array de caracteres.

Las cadenas se deben almacenar en arrays de caracteres, pero no todos Los arrays de caracteres contienen cadenas.

La figura siguiente muestra una cadena de caracteres y un arreglo de caracteres:



EJEMPIO 4.43

Rellenar los elementos de un arreglo con datos procedentes del teclado.

```
// Rellenado de datos de una lista
#include < iostream.h >

const int MAX = 10;
int muestra  MAX ;
void main ( )
{
        cout << “introduzca una lista de” << MAX << “”elementos y pulse la tecla intro”;
        for (int j = 0; j < MAX; ++j)
        cin >> Muestra  [ j ];
}
```

EJEMPLO 4.44

Visualizar un arreglo después de introducir datos en el mismo.

```
//Visualizar los elementos de un arreglo.
#include <iostream.h>
//Constantes y variables globales
const int MAX = 10;
        int Muestra  [MAX] ;
void main ( )
{
        for (int j = 0; j < MAX; ++j)
                Muestra [ j ]  = j * j;
        for (j = 0; j < MAX; ++j;
                cout << Muestra [ j ];
}
```

4.36 ARREGLOS MULTIDIMENSIONALES

Los arreglos vistos anteriormente se conocen como arreglos de una dimensión y se caracterizan por tener un solo subíndice. Estos arreglos se conocen también por el término listas. Los arreglos multidimensionales son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arreglos más usuales son los de dos dimensiones, conocidos también por el nombre de **tablas o matrices.** La sintaxis para la declaración de un arreglo de dos dimensiones es la siguiente:

Tipo de elemento nombre del arreglo Numero de filas Numero de columnas

Algunos ejemplos de declaración de tablas son:

```
char pantalla [25] [80];
int puestos [ 6 ] [ 8 ];
int equipos [ 4 ] [ 30 ];
int matriz  [ 4 ] [ 2 ];
```

Un arreglo de dos dimensiones es en realidad un arreglo de arreglos, los elementos de los arreglos se almacenan en memoria de modo que el subíndice más próximo al nombre del arreglo es la fila y el otro subíndice, la columna, por ejemplo:

int tabla [4] [2];

4.37 INICIALIZACION DE ARREGLOS MULTIDIMENSIONALES

Los arreglos multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por coma y encerradas entre llaves, como en el siguiente ejemplo:

int tabla [2] [3] = { 51, 52, 53, 54, 55, 56 };

4.38 ACCESO A LOS ELEMENTOS DE UN ARREGLO BIDIMENSIONAL

Se puede acceder a los elementos de un arreglo bidimensional de igual forma que a los elementos de un arreglo unidimensional. La diferencia reside en que en los elementos bidimensionales deben especificarse los índices de las filas y las columnas. El formato general para asignación directa de valores a los elementos es:

Inserción de elementos
< Nombre del arreglo> [índice fila] [índice columna] = valor elemento;

Extracción de elementos
< Variable > = <nombre del arreglo> [índice fila] [índice columna];

Algunos ejemplos de inserciones pueden ser:

Tabla [2] [3] = 4.5;
Asientos libres [5] [12] = 5;

y de extracción de valores:

Ventas = Tabla [1] [1];
Día = semana [3] [6];

4.39 LECTURA Y ESCRITURA DE ELEMENTOS DE ARREGLOS BIDIMENSIONALES

La sentencia cin y cout se utilizan para extraer elementos del arreglo. Por ejemplo:

```
cin   Tabla  [2]  [3];
cout    Tabla  [1]  [1];
cin  resistencias  [2]  [4]
if  (Asientos libres  [3]  [1]  )
        cout  “verdadero”;
else
        cout  “falso”;
```

EJEMPLO 4.45

Lectura y visualización de un arreglo de dos dimensiones.

```
// La función leer lee un arreglo (tabla) de dos dimensiones y la función visualizar presenta
//  la tabla en pantalla.

//generar un arreglo de dos dimensiones
#include <iostream.h>
#include <conio.h>
void leer (int a[][5]);
void visualizar (const int a[][5]);

main()
{
        int a[3][5];
  leer (a);
  visualizar(a);
}

void leer (int a[][5])
{
        cout<<"introduzca 15 numeros enteros, 3 por fila"<<endl;
  for( int i=0; i<3; i++)
  {
  cout<<"fila"<<i<<":";
  for (int j=0; j<5; j++)
  cin>>a[i][j];
  }
}

void visualizar (const int a[][5])
{
for (int i=0; i<3; i++)
        {
  for (int j=0; j<5; j++)
        cout<<" "<<a[i][j];
  cout<<endl;
  getch();
  }
}
```

EJEMPLO 4.46

Encontrar el número mayor de una lista

```
//Encontrar el mayor valor del arreglo
#include <iostream.h>
#include <conio.h>
#define LONG 15
main()
{
int lista[LONG] = { 5, 2, 4,990, 14, 4, 2, 85, 11,45,25,12,45,6,99};

int valor_mayor, i;
valor_mayor = lista[0];

for (i=1; i<LONG; i++)
{
if (lista[i] > valor_mayor)
{
valor_mayor = lista[i];
}
}
cout<<"El numero mayor de la lista es:"<<valor_mayor<<"\n";
getch();
return 0;
}
```

V

CANALES DE COMUNICACIÓN

Objetivo particular de la unidad

Elaborar programas que utilicen los puertos como medio de comunicación entre la cpu y un dispositivo periférico.

Tipo de aprendizaje por desarrollar

Conocimiento y habilidades

V.1 CANALES DE COMUNICACION

Saber en la teoría (5 hrs.)

5.1 PUERTO SERIE

El puerto serie RS-232C, presente en todos las computadoras actuales, es la forma mas comúnmente usada para realizar transmisiones de datos entre computadoras. El RS-232C es un estándar que constituye la tercera revisión de la antigua norma RS-232, propuesta por la EIA (Asociación de Industrias Electrónicas), realizándose posteriormente una versión internacional por el CCITT, conocida como V.24. Las diferencias entre ambas son mínimas, por lo que a veces se habla indistintamente de V.24 y de RS-232C (incluso sin el sufijo "C"), refiriéndose siempre al mismo estándar.
El RS-232C consiste en un conector tipo DB-25 de 25 pines, aunque es normal encontrar la versión de 9 pines DB-9, mas barato e incluso mas extendido para cierto tipo de periféricos (como el ratón serie del PC). En cualquier caso, los PCs no suelen emplear mas de 9 pines en el conector DB-25. Las señales con las que trabaja este puerto serie son digitales, de +12V (0 lógico) y -12V (1 lógico), para la entrada y salida de datos, y a la inversa en las señales de control. El estado de reposo en la entrada y salida de datos es -12V. Dependiendo de la velocidad de transmisión empleada, es posible tener cables de hasta 15 metros.

Cada pin puede ser de entrada o de salida, teniendo una función específica cada uno de ellos. Las más importantes son:

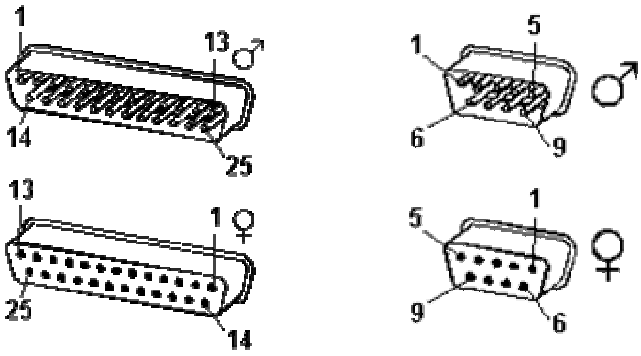
Pin	Función
TXD	(Transmitir Datos)
RXD	(Recibir Datos)
DTR	(Terminal de Datos Listo)
DSR	(Equipo de Datos Listo)
RTS	(Solicitud de Envío)
CTS	(Libre para Envío)
DCD	(Detección de Portadora)

Las señales TXD, DTR y RTS son de salida, mientras que RXD, DSR, CTS y DCD son de entrada. La masa de referencia para todas las señales es SG (Tierra de Señal).

Numero	de Pin	Señal	Descripción	E/S
En DB-25	En DB-9			
1	1	-	Masa chasis	-
2	3	TxD	Transmit Data	S
3	2	RxD	Receive Data	E
4	7	RTS	Request To Send	S
5	8	CTS	Clear To Send	E
6	6	DSR	Data Set Ready	E
7	5	SG	Signal Ground	-
8	1	CD/DCD	(Data) Carrier Detect	E
15	-	TxC(*)	Transmit Clock	S
17	-	RxC(*)	Receive Clock	E
20	4	DTR	Data Terminal Ready	S
22	9	RI	Ring Indicator	E
24	-	RTxC(*)	Transmit/Receive Clock	S

(*) = Normalmente no conectados en el DB-25

Conector DB 25	Conector DB 9



5.2 EL PUERTO SERIE EN LA PC.

La computadora controla el puerto serie mediante un circuito integrado especifico, llamado UART (Transmisor-Receptor-Asíncrono Universal). Normalmente se utilizan los siguientes modelos de este chip: 8250 (bastante antiguo, con fallos, solo llega a 9600 baudios), 16450 (versión corregida del 8250, llega hasta 115.200 baudios) y 16550A (con buffers de E/S). A partir de la gama Pentium, la circuiteria UART de la placa base son todas de alta velocidad, es decir UART 16550A. De hecho, la mayoría de los módems conectables a puerto serie necesitan dicho tipo de UART, incluso algunos juegos para jugar en red a través del puerto serie necesitan de este tipo de puerto serie. Por eso hay veces que un 486 no se comunica con la suficiente velocidad con un PC Pentium... Los portátiles suelen llevar otros chips: 82510 (con buffer especial, emula al 16450) o el 8251 (no es compatible).
Para controlar al puerto serie, la CPU emplea direcciones de puertos de E/S y líneas de interrupción (IRQ). En el AT-286 se eligieron las direcciones 3F8h (o 0x3f8) e IRQ 4 para el COM1, y 2F8h e IRQ 3 para el COM2. El estándar del PC llega hasta aquí, por lo que al añadir posteriormente otros puertos serie, se eligieron las direcciones 3E8 y 2E8 para COM3-COM4, pero las IRQ no están especificadas. Cada usuario debe elegirlas de acuerdo a las que tenga libres o el uso que vaya a hacer de los puertos serie (por ejemplo, no importa compartir una misma IRQ en dos puertos siempre que no se usen conjuntamente, ya que en caso contrario puede haber problemas). Es por ello que últimamente, con el auge de las comunicaciones, los fabricantes de PCs incluyan un puerto especial PS/2 para el ratón, dejando así libre un puerto serie.

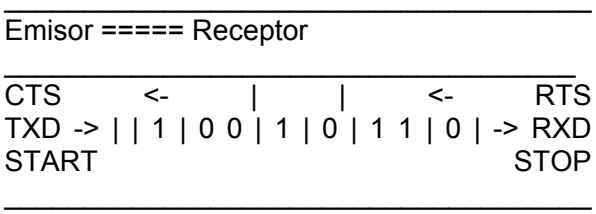
Mediante los puertos de E/S se pueden intercambiar datos, mientras que las IRQ producen una interrupción para indicar a la CPU que ha ocurrido un evento (por ejemplo, que ha llegado un dato, o que ha cambiado el estado de algunas señales de entrada). La CPU debe responder a estas interrupciones lo mas rápido posible, para que de tiempo a recoger el dato antes de que el siguiente lo sobrescriba. Sin embargo, las UART 16550A incluyen unos buffers de tipo FIFO, dos de 16 bytes (para recepción y transmisión), donde se pueden guardar varios datos antes de que la CPU los recoja. Esto también disminuye el numero de interrupciones por segundo generadas por el puerto serie.
El RS-232 puede transmitir los datos en grupos de 5, 6, 7 u 8 bits, a unas velocidades determinadas (normalmente, 9600 bits por segundo o más). Después de la transmisión de los datos, le sigue un bit opcional de paridad (indica sí él numero de bits transmitidos es par o impar, para detectar fallos), y después 1 o 2 bits de Stop. Normalmente, el protocolo utilizado serial 8N1 (que significa, 8 bits de datos, sin paridad y con un bit de stop).

Una vez que ha comenzado la transmisión de un dato, los bits tienen que llegar uno detrás de otro a una velocidad constante y en determinados instantes de tiempo. Por eso se dice que el RS-232 es asíncrono por carácter y sincrono por bit. Los pines que portan los datos son RXD y TXD. Las demás se encargan de otros trabajos: DTR indica que la computadora esta encendido, DSR que el aparato conectado a dicho puerto esta encendido, RTS que la computadora puede recibir datos (porque no esta ocupado), CTS que el aparato conectado puede recibir datos, y DCD detecta que existe una comunicación presencia de datos.

Tanto el aparato a conectar como el ordenador (o el programa terminal) tienen que usar el mismo protocolo serie para comunicarse entre si. Puesto que el estándar RS-232 no permite indicar en que modo se esta trabajando, es el usuario quien tiene que decidirlo y configurar ambas partes. Como ya se ha visto, los parámetros que hay que configurar son: protocolo serie (8N1), velocidad del puerto serie, y protocolo de control de flujo. Este ultimo puede ser por hardware (el que ya hemos visto, el handshaking RTS/CTS) o bien por software (XON/XOFF, el cual no es muy recomendable ya que no se pueden realizar transferencias binarias). La velocidad del puerto serie

no tiene por que ser la misma que la de transmisión de los datos, de hecho debe ser superior. Por ejemplo, para transmisiones de 1200 baudios es recomendable usar 9600, y para 9600 baudios se pueden usar 38400 (o 19200).

Este es el diagrama de transmisión de un dato con formato 8N1. El receptor indica al emisor que puede enviarle datos activando la salida RTS. El emisor envía un bit de START (nivel alto) antes de los datos, y un bit de STOP (nivel bajo) al final de estos.



5.3 PUERTO PARALELO:

PRECAUCION: Conectar dispositivos al puerto paralelo implica el riesgo de daños permanentes a la tarjeta madre de la PC, tenga siempre presente que aún los profesionales cometen errores, por lo tanto no está de más recomendarle extremo cuidado al trabajar en el puerto paralelo. Se recomiendan conocimientos sólidos en electrónica y programación para manipular el puerto paralelo.

Los puertos de comunicación de la PC son de particular interés para el estudioso de la electrónica ya que le permiten utilizar una computadora personal para controlar todo tipo circuitos electrónicos utilizados, principalmente, en actividades de automatización de procesos, adquisición de datos, tareas repetitivas y otras actividades que demandan precisión. Éste artículo es el primero de una serie que analizará diversos usos para el puerto paralelo de la PC.

Existen dos métodos básicos para transmisión de datos en las computadoras modernas. En un esquema de transmisión de datos en serie un dispositivo envía datos a otro a razón de un bit a la vez a través de un cable. Por otro lado, en un esquema de transmisión de datos en paralelo un dispositivo envía datos a otro a una tasa de *n* número de bits a través de *n* número de cables a un tiempo. Sería fácil pensar que un sistema en *paralelo* es *n* veces más rápido que un sistema en *serie*, sin embargo esto no se cumple, básicamente el impedimento principal es el tipo de cable que se utiliza para interconectar los equipos. Si bien un sistema de comunicación en paralelo puede utilizar cualquier número de cables para transmitir datos, la mayoría de los sistemas paralelos utilizan ocho líneas de datos para transmitir un byte a la vez, como en todo, existen excepciones, por ejemplo el estándar SCSI permite transferencia de datos en esquemas que van desde los ocho bits y hasta los treinta y dos bits en paralelo. En esta unidad nos concentraremos en transferencias de ocho bits ya que ésta es la configuración del puerto paralelo de una PC.

Un típico sistema de comunicación en paralelo puede ser de una dirección (*unidireccional*) o de dos direcciones (*bidireccional*). El más simple mecanismo utilizado en un puerto paralelo de una PC es de tipo unidireccional y es el que analizaremos en primer lugar. Distinguimos dos elementos: la *parte transmisora* y la *parte receptora*. La parte transmisora coloca la información en las *líneas de datos* e informa a la parte receptora que la información (los datos) están disponibles; entonces la parte receptora lee la información en las líneas de datos e informa a la parte transmisora que ha tomado la información (los datos). Observe que ambas partes sincronizan su respectivo acceso a las líneas de datos, la parte receptora no leerá las

líneas de datos hasta que la parte transmisora se lo indique en tanto que la parte transmisora no colocará nueva información en las líneas de datos hasta que la parte receptora remueva la información y le indique a la parte transmisora que ya ha tomado los datos, a ésta coordinación de operaciones se le llama *acuerdo* ó *entendimiento*. Bien, en éstos ámbitos tecnológicos es recomendable utilizar ciertas palabras en inglés que nos permiten irónicamente un mejor entendimiento de los conceptos tratados. Repito: a la coordinación de operaciones entre la parte transmisora y la parte receptora se le llama handshaking, que en español es el acto con el cual dos partes manifiestan estar de acuerdo, es decir, se dan un apretón de manos.

Para implementar el *handshaking* se requieren dos líneas adicionales. La línea de estroboscopio (en inglés *strobe*) es la que utiliza la parte transmisora para indicarle a la parte receptora la disponibilidad de información. La línea de admisión (*acknowledge*) es la que utiliza la parte receptora para indicarle a la parte transmisora que ha tomado la información (los datos) y que está lista para recibir más datos. El puerto paralelo provee de una tercera línea de handshaking llamada en inglés busy (*ocupado*), ésta la puede utilizar la parte receptora para indicarle a la parte transmisora que está ocupada y por lo tanto la parte transmisora no debe intentar colocar nueva información en las líneas de datos. Una típica sesión de transmisión de datos se parece a lo siguiente:

Parte transmisora:
- La parte transmisora checa la línea *busy* para ver si la parte receptora está ocupada. Si la línea *busy* está activa, la parte transmisora espera en un bucle hasta que la línea *busy* esté inactiva.
- La parte transmisora coloca la información en las líneas de datos.
- La parte transmisora activa la línea de *strobe*.
- La parte transmisora espera en un bucle hasta que la línea *acknowledge* está activa.
- La parte transmisora inactiva la línea de *strobe*.
- La parte transmisora espera en un bucle hasta que la línea *acknowledge* esté inactiva.
- La parte transmisora repite los pasos anteriores por cada byte a ser transmitido.

Parte receptora:
- La parte receptora inactiva la línea *busy* (asumiendo que está lista para recibir información).
- La parte receptora espera en un bucle hasta que la línea *strobe* esté activa.
- La parte receptora lee la información de las líneas de datos (y si es necesario, procesa los datos).
- La parte receptora activa la línea *acknowledge*.
- La parte receptora espera en un bucle hasta que esté inactiva la línea de *strobe*.
- La parte receptora inactiva la línea *acknowledge*.
- La parte receptora repite los pasos anteriores por cada byte que debe recibir.

Se debe ser muy cuidadoso al seguir éstos pasos, tanto la parte transmisora como la receptora coordinan sus acciones de tal manera que la parte transmisora no intentará colocar varios bytes en las líneas de datos, en tanto que la parte receptora no debe leer más datos que los que le envíe la parte transmisora, un byte a la vez.

5.4 EL HARDWARE DEL PUERTO PARALELO

El puerto paralelo de una típica PC utiliza un conector hembra de tipo D de 25 patitas (DB-25 S), éste es el caso más común, sin embargo es conveniente mencionar los tres tipos de conectores definidos por el estándar IEEE 1284, el primero, llamado 1284 tipo A es un conector hembra de 25 patitas de tipo D, es decir, el que mencionamos al principio. El orden de las patitas del conector es éste:

El segundo conector se llama **1284 tipo B** que es un conector de 36 patitas de tipo *centronics* y lo encontramos en la mayoría de las impresoras; el tercero se denomina **1284 tipo C**, se trata de un conector similar al 1284 tipo B pero más pequeño, además se dice que tiene mejores propiedades eléctricas y mecánicas, éste conector es el recomendado para nuevos diseños. La siguiente tabla describe la función de cada patita del conector 1284 tipo A:

Patita	**E/S**	**Polaridad activa**	**Descripción**
1	Salida	0	Strobe
2 ~ 9	Salida	-	Líneas de datos (bit 0/patita 2, bit 7/patita 9)
10	Entrada	0	Línea acknowledge (activa cuando el sistema remoto toma datos)
11	Entrada	0	Línea busy (si está activa, el sistema remoto no acepta datos)
12	Entrada	1	Línea Falta de papel (si está activa, falta papel en la impresora)
13	Entrada	1	Línea Select (si está activa, la impresora se ha seleccionado)
14	Salida	0	Línea Autofeed (si está activa, la impresora inserta una nueva línea por cada retorno de carro)
15	Entrada	0	Línea Error (si está activa, hay un error en la impresora)
16	Salida	0	Línea Init (Si se mantiene activa por al menos 50 micro-segundos, ésta señal autoinicializa la impresora)
17	Salida	0	Línea Select input (Cuando está inactiva, obliga a la impresora a salir de línea)
18 ~ 25	-	-	Tierra eléctrica

Tabla 1: Configuración del puerto paralelo estándar

Observe que el puerto paralelo tiene 12 líneas de salida (8 líneas de datos, strobe, autofeed, init, y select input) y 5 de entrada (acknowledge, busy, falta de papel, select y error). El estándar IEEE 1284 define cinco modos de operación:

Modo compatible
Modo nibble
Modo byte
Modo EPP, puerto paralelo ampliado
Modo ECP, puerto de capacidad extendida

El objetivo del estándar es diseñar nuevos dispositivos que sean totalmente compatibles con el puerto paralelo estándar (SPP) definido originalmente por la IBM. Hay tres direcciones de E/S asociadas con un puerto paralelo de la PC, éstas direcciones pertenecen al **registro de datos**, el **registro de estado** y el **registro de control**. El *registro de datos* es un puerto de lectura-escritura de ocho bits. Leer el registro de datos (en la modalidad unidireccional) retorna el último valor escrito en el registro de datos. Los registros de control y estado proveen la interfase a las otras líneas de E/S. La distribución de las diferentes señales para cada uno de los tres registros de un puerto paralelo esta dada en las siguientes tablas:

Dirección	**Nombre**	**Lectura/Escritura**	**Bit #**	**Propiedades**
Base + 0	Puerto de datos	Escritura	Bit 7	Dato 7
			Bit 6	Dato 6
			Bit 5	Dato 5
			Bit 4	Dato 4
			Bit 3	Dato 3
			Bit 2	Dato 2
			Bit 1	Dato 1
			Bit 0	Dato 0

Tabla 2: Registro de datos

Dirección	Nombre	Lectura/Escritura	Bit #	Propiedades
Base + 1	Puerto de estado	Sólo Lectura	Bit 7	Busy
			Bit 6	Acknowledge
			Bit 5	Falta de papel
			Bit 4	Select In
			Bit 3	Error
			Bit 2	IRQ (Not)
			Bit 1	Reservado
			Bit 0	Reservado

Tabla 3: Registro de estado

Dirección	Nombre	Lectura/Escritura	Bit #	Propiedades
Base + 2	Puerto de control	Lectura/Escritura	Bit 7	No usado
			Bit 6	No usado
			Bit 5	Permite puerto bidireccional
			Bit 4	Permite IRQ a través de la línea acknowledge
			Bit 3	Selecciona impresora
			Bit 2	Inicializa impresora
			Bit 1	Nueva línea automática
			Bit 0	Strobe

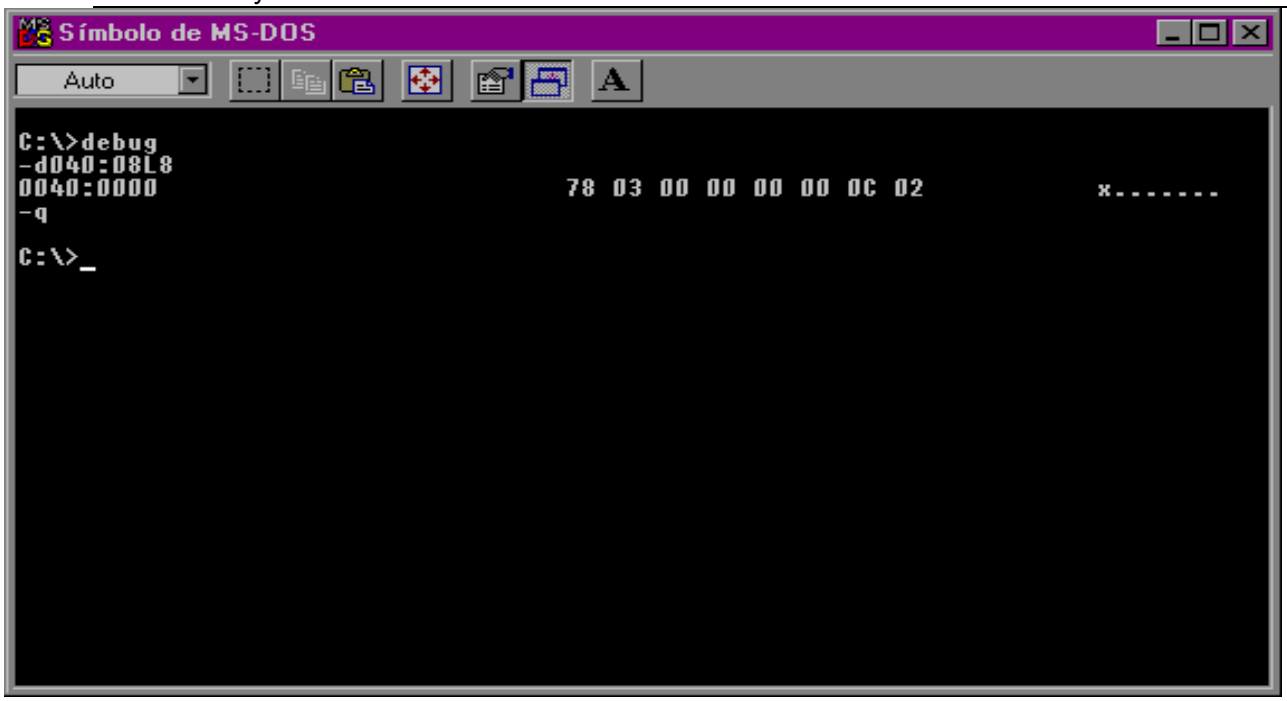
Tabla 4: Registro de control

Una PC soporta hasta tres puertos paralelo separados, por tanto puede haber hasta tres juegos de registros en un sistema en un momento dado. Existen tres **direcciones base** para el puerto paralelo asociadas con tres posibles puertos paralelo: 0x3BCh, 0x378h y 0x278h, nos referimos a éstas como las direcciones base para el puerto **LPT1**, **LPT2** y **LPT3**, respectivamente. El registro de datos se localiza siempre en la dirección base de un puerto paralelo, el registro de estado aparece en la dirección base + 1, y el registro de control aparece en la dirección base + 2. Por ejemplo, para un puerto LPT2 localizado en 0x378h, ésta es la dirección del registro de datos, al registro de estado le corresponde la dirección 0x379h y su respectivo registro de control está en la dirección 0x37Ah. Cuando la PC se enciende el BIOS ejecuta una rutina para determinar el número de puertos presentes en el sistema asignando la etiqueta LPT1 al primer puerto localizado, si existen más puertos entonces se asignarán consecutivamente las etiquetas LPT2 y LPT3 de acuerdo a la siguiente tabla:

Dirección inicial	Función
0000:0408	Dirección base para LPT1
0000:040A	Dirección base para LPT2
0000:040C	Dirección base para LPT3
0000:040E	Dirección base para LPT4

Tabla 5: Direcciones base en el BIOS

Para trabajar con el puerto paralelo necesitamos en primer lugar conocer la dirección base asignada por el BIOS (estamos hablando de una PC compatible con IBM), podemos utilizar un programa llamado **Debug.exe** que nos indique la(s) dirección(es) asignada(s): en la plataforma Windows vamos al menú *inicio*, seleccionamos *programas* y luego *MS-DOS* para abrir una ventana de *Símbolo de MS-DOS* y aquí podemos introducir los comandos indicados más abajo. Si se trabaja en ambiente DOS basta con teclear en la línea de comandos la palabra **debug**, el programa responde colocando un signo de menos **-** en donde tecleamos sin dejar espacios en blanco **d040:08L8** y presionamos la tecla *entrar*, entonces el programa *debug.exe* nos indica en una serie de números la(s) dirección(es) para el (los) puerto(s) paralelo(s) disponibles en nuestro sistema, la siguiente imagen muestra el resultado obtenido en mi máquina:



Se puede observar una serie de números de dos dígitos (ocho en total), se trata del volcado de memoria que empieza en la dirección 40:0008h. Los primeros seis pares de números representan las direcciones base para los puertos paralelo instalados, en la imagen de arriba se aprecia que el único puerto paralelo de la máquina está en la dirección 0x378h (78 03). Los números están invertidos porque Intel almacena tal información en un formato de "byte de bajo orden - byte de alto orden". Una vez que obtenemos la información deseada cerramos el programa *Debug.exe* simplemente tecleando la letra **q** y presionando la tecla *entrar*. Para cerrar la ventana de *Símbolo de MS-DOS* tecleamos la palabra **exit** y presionamos la tecla *entrar*.

Por supuesto, también podemos conocer la dirección asignada al puerto paralelo utilizando un programa, el código es el siguiente:

```
/*************************************************************
*  puerto1.c                                    *

*  Determina la direccion del puerto paralelo          *
*************************************************************/

#include <stdio.h>
#include <dos.h>

int main()
{
   unsigned int __far *puntero_a_direccion;
   int i;

   puntero_a_direccion=(unsigned int __far *)0x00000408;

   for (i=0; i<3; i++)
   {
```

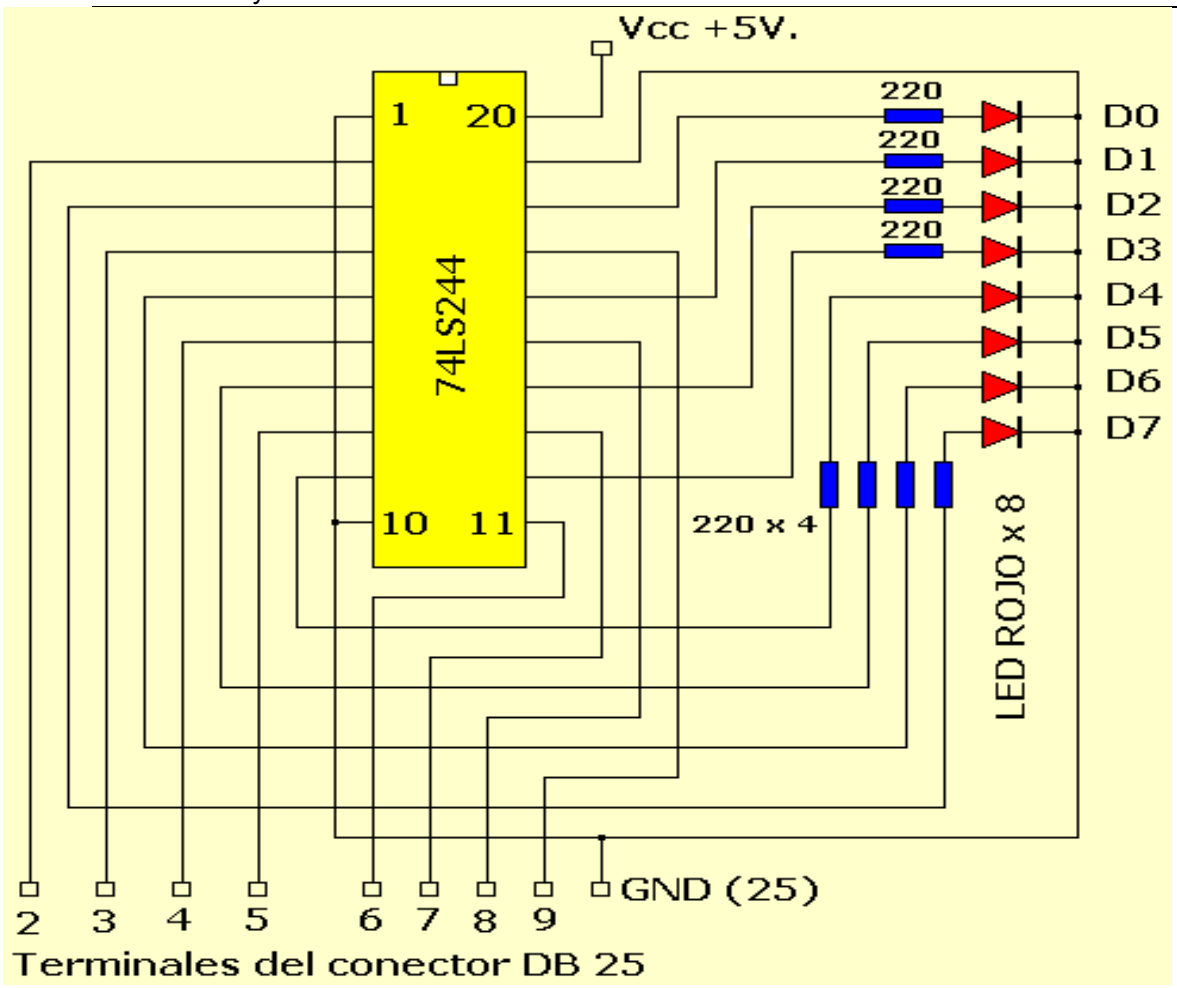
```
      if (*puntero_a_direccion == 0)
      cout<<"No se encontro puerto asignado a LPT%d \n", i+1);
      else
      cout<<"La direccion asignada a LPT%d es 0x%Xh\n",
        i+1, *puntero_a_direccion);
      puntero_a_direccion++;
   }

   return 0;
}
```

El programa es muy sencillo, necesitamos un puntero que señale a la ubicación de memoria que almacena la información para las diferentes direcciones asignadas al puerto paralelo, lo llamamos **puntero_a_direccion**, éste puntero debe ser de al menos dos bytes, por ésta razón se declara como de tipo **unsigned int __far**. Es importante que se verifique en la documentación de su compilador la manera precisa para utilizar la palabra clave *far*, ésta puede ser simplemente la palabra como tal ó bien puede llevar uno (*_far*) o dos guiones bajos (*__far*), asegúrese de utilizar la forma correcta ó el programa no funcionará. Creo que con lo explicado arriba el funcionamiento del programa *puerto1.c* no presenta problema alguno.

5.5 INTERFAZ PARA EL PUERTO PARALELO

Actualmente, la mayoría de los puertos instalados en las computadoras son de tipo multimodal configurables a través del BIOS de la máquina, en éste artículo me refiero expresamente al modo **Normal (SPP)**, además de éste están las opciones **Bidireccional**, **EPP versión 1.7**, **EPP versión 1.9** y **ECP** principalmente. El modo de operación *Normal* es el más elemental y solamente permite la escritura en las líneas de datos, patitas 2 a la 9 del conector DB-25 del puerto paralelo de la PC.

Eléctricamente, el puerto paralelo entrega señales TTL y como tal, teóricamente, se le puede conectar cualquier dispositivo que cumpla con los niveles de voltaje específicos de la lógica TTL, sin embargo el hardware del puerto paralelo está muy limitado en cuanto a su capacidad de manejo de corriente, por ésta razón se debe ser muy cuidadoso con el manejo de las señales del puerto, un corto circuito puede dañar permanentemente la tarjeta madre de la PC. Para disminuir lo más posible el riesgo de daños al puerto utilizamos un circuito integrado **74LS244** como etapa separadora y al mismo tiempo mejoramos la capacidad de manejo de corriente, de esta forma podemos conectar una serie de diodos emisores de luz (LED) que nos indiquen la actividad en las líneas de datos del puerto paralelo. El circuito se detalla en el siguiente diagrama:



Por cada línea de entrada que tomamos directamente del puerto paralelo existe una etapa amplificadora-separadora dentro del circuito integrado 74LS244 que nos permite trabajar con una tasa de entrega de corriente suficiente para desplegar en los diodos emisores de luz la información escrita en las líneas de datos del puerto. Además es posible habilitar ó deshabilitar el despliegue del nibble de orden inferior ó superior del byte escrito en el puerto. Colocando en un nivel lógico alto la patita 1 del CI 74LS244 inhabilitamos el despliegue del nibble de orden bajo, un nivel lógico alto en la patita 19 evita el despliegue del nibble de orden alto. Para comodidad, conecto las patitas 1 y 19 permanentemente a tierra de tal manera que sea posible visualizar la actividad en los diodos emisores de luz (LED). En el diagrama se especifican con números las correspondientes patitas del conector DB-25. Obviamente se requiere de una fuente regulada de 5 Voltios para operar éste circuito, además los siguientes materiales:

- 1. Circuito Integrado TTL 74LS244.
- 8. Diodos Emisores de Luz.
- 8. Resistencias de 220 Ohms, 1/2 Watt.
- 1. Cable y conector para el puerto paralelo.

Naturalmente lo más recomendable es probar el correcto funcionamiento del circuito **antes** de conectarlo al puerto paralelo de la PC. Ensamble el circuito, preferentemente en un circuito impreso, y conéctelo a una fuente regulada de 5 Voltios, conecte temporalmente un extremo de una resistencia de 10,000 Ohms a una línea de entrada, el resto de las líneas de entrada conéctelas a tierra. El otro extremo de la resistencia conéctelo directamente al borne positivo de la fuente de alimentación para inducir una señal TTL alta, el respectivo LED debe encender. Con un trozo de alambre conectado a Tierra, toque temporal mente el extremo de la resistencia que está conectado a la línea de entrada para inducir una señal TTL de lógica baja, el LED se debe apagar. Repita ésta operación para cada una de las ocho líneas de entrada. Una vez que ha verificado el correcto funcionamiento del circuito está listo para conectarlo al puerto paralelo de la PC.

En primer lugar apague la computadora y el circuito. Conecte el cable al puerto paralelo asegurándose que el conector esté firme en su sitio. Encienda el circuito y por último encienda la computadora, por regla general, el circuito de restablecimiento de la computadora coloca inicialmente en las líneas de datos del puerto paralelo un valor igual a 0x0h, por lo tanto todos los diodos deben estar apagados una vez que la computadora ha terminado su proceso de arranque, sin embargo, si algún diodo permanece encendido esto no indica una condición de falla, es responsabilidad del software que usted escriba para controlar el puerto inicializarlo con un valor adecuado antes de realizar cualquier otra operación.

5.6 ESCRIBIENDO DATOS AL PUERTO PARALELO.

Con ocho bits podemos escribir en el puerto un total de 256 valores diferentes, cada uno de éstos representa un byte de información y cada byte puede representar una acción concreta que nosotros podemos definir de acuerdo a nuestras necesidades. En éste unidad el objetivo es entender cómo trabajar con el puerto paralelo, por lo tanto hagamos un programa que nos permita escribir un número cualquiera entre 0 y 255 de tal manera que sea posible visualizar el valor en formato binario. En primer lugar consulte la documentación de su compilador para verificar la correcta sintaxis de la función que nos sirve para escribir en el puerto, en el caso específico de Symantec C++, dicha función es **outp( )**. Ésta función requiere dos parámetros, el primero de tipo *unsigned int* que especifica la dirección del puerto paralelo, y el segundo de tipo *char* que especifica el valor a escribir en las líneas de datos de puerto. Una típica llamada a la función *outp( )* se parece a ésto:

```
outp(0x378, 65);
```

Se aprecia la facilidad de manejo de la función, aunque diferentes compiladores dan a sus respectivas funciones nombres diferentes, la mecánica es la misma, se requieren dos parámetros, la dirección del puerto y el valor a escribir en el puerto. Estudie el siguiente código:

```
/*********************************************************
*  puerto2.c                                            *
*  Escribe datos al puerto paralelo de la PC            *
*                                                       *
*********************************************************/

#include <stdio.h>
#include <dos.h>

int puerto(int direcc);
int seleccion;

int main()
{
   unsigned int __far *puntero_a_direccion;
   int i, direccion[3]={0,0,0}, disponible[3]={0,0,0};

   puntero_a_direccion = (unsigned int __far *)0x00000408;

   cout<<"Seleccione el puerto:\n");
   /* ¿Cuantos puertos existen? */
   for (i=0; i<3; i++)
   {
      if (*puntero_a_direccion == 0)
      cout<<"Puerto LPT%d.............no disponible\n", i+1);
      else
      {
         disponible[i] = 1;
         direccion[i] = *puntero_a_direccion;
         cout<<"Puerto LPT%d.............%d\n", i+1, i+1);
      }
      puntero_a_direccion++;
   }
   cout<<("Salir del programa.....0\n");
   scanf("%d", &seleccion);

   do
   {
      switch(seleccion)
      {
         case 0:/* Salir del programa */
         cout<<("Adios!!!\n");
         return 0;
         break;

         case 1:/* Puerto LPT1 */
         if(disponible[0]==1)
         puerto(direccion[0]);
         else
         {
            cout<<("ERROR: PUERTO NO DISPONIBLE\n");
            return 0;
         }
```

```
            break;

            case 2:/* Puerto LPT2 */
            if(disponible[1]==1)
            puerto(direccion[1]);
            else
            {
               cout<<"ERROR: PUERTO NO DISPONIBLE\n");
               return 0;
            }
            break;

            case 3:/* Puerto LPT3 */
            if(disponible[2]==1)
            puerto(direccion[2]);
            else
            {
               cout<<"ERROR: PUERTO NO DISPONIBLE\n");
               return 0;
            }
            break;

            default:
            cout<<("ERROR: OPCION INCORRECTA!\n");
            break;
         }
         if(seleccion!=0)
         {
            cout<<"Seleccione otra opcion\n");
            scanf("%d", &seleccion);
         }
      }
      while(seleccion!=0);

      return 0;
}

int puerto(int direcc)
{
   unsigned char valor; /* Solo valores entre 0 y 255 */

   cout<<"Ahora puede escribir cualquier valor\n");
   cout<<"entre 1 y 255, 0 para terminar el programa\n");
   do
   {
      cout<<("Valor al puerto?\n");
      scanf("%d", &valor);
      outp(direcc, valor);
      cout<<"Se ha escrito %d al puerto\n", valor);
   }
   while(valor!=0);

   return seleccion=0;
}
```

El programa determina el número de puertos paralelos instalados en la computadora y almacena sus respectivas direcciones en el arreglo llamado **dirección[i]**, en caso de encontrarse una dirección válida además se establece en 1 el valor del arreglo llamado **disponible[i]** simplemente como una medida de seguridad adicional. Como resultado de éstas acciones el programa despliega un menú basado en los puertos encontrados, en este punto se puede seleccionar, en caso de que disponga de más de un puerto, el puerto en donde está conectado el circuito mostrado en el diagrama de arriba.

Una vez seleccionado el puerto se puede escribir un valor cualquiera comprendido entre 0 y 255. Independientemente del puerto seleccionado toda la funcionalidad del programa está encapsulada en la función de tipo *int* llamada **puerto ( )** a la cual se le pasa un único parámetro que es la dirección del puerto seleccionado. La función está codificada de tal forma que al escribir un 0 el programa termina, de ésta manera al cerrar el programa las líneas de datos del puerto paralelo están todas en un nivel lógico bajo. Éste programa y el circuito asociado son útiles para entender de forma visual la forma de representar valores en formato binario, además establece las bases de trabajo para otros proyectos de control basados en computadora. Disfrute la experiencia.

5.7 ENVIANDO DATOS POR EL PUERTO PARALELO.

Para comenzar construiremos un circuito que nos permita enviar señales por el puerto paralelo y lo haremos utilizando diodos LEDs, como el consumo de los LEDs es superior al que nos envía el puerto utilizaremos un buffer como el 74HC244 o el 74HCT245, prefiero este último ya que la construcción de la placa será mas sencilla, aquí tienes los datos del integrado.

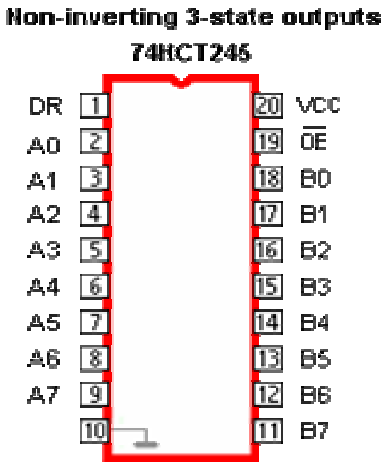


Tabla de Verdad

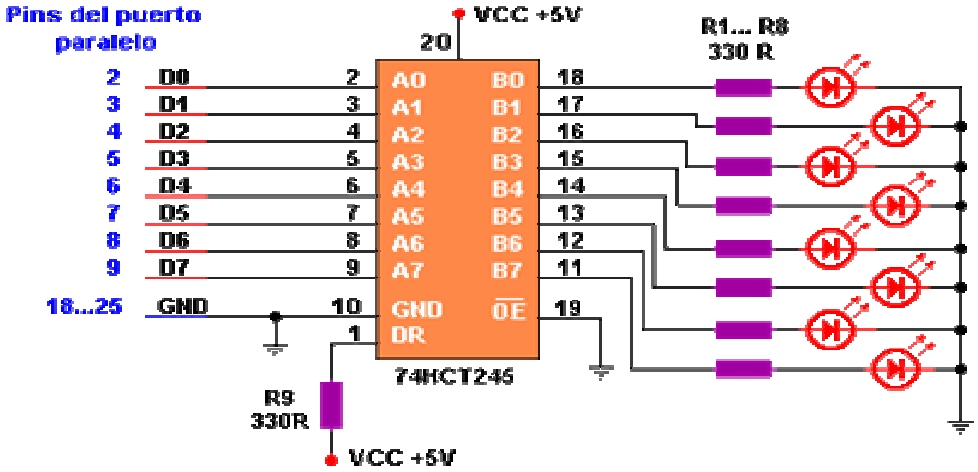
INPUTS		INPUTS/OUTPUTS	
$\overline{OE}$	DIR	A_n	B_n
L	L	A = B	inputs
L	H	inputs	B = A
H	X	Z	Z

Nota:
X = Sin importancia
Z = Alta impedancia

Este integrado tiene la ventaja de ser bidireccional, es decir todos los pins A pueden ser entradas y los B salidas si DIR (pin1 DR) se encuentra a nivel bajo (L), el modo invertido se obtiene con DR al nivel alto. Si el terminal OE (pin 19) se encuentra a nivel alto (H) el integrado pone los pins A y B con alta impedancia, es decir ni entra ni sale señal alguna, en ese caso el estado de DR no tiene importancia. En la siguiente tabla tienes los nombres de los pines y sus funciones correspondientes.

No. PIN	Nombre	Función
1	DIR	Control de dirección
2, 3, 4, 5, 6, 7, 8, 9	A0-A7	Entrada/Salida de datos
10	GND	Fuente (0V)
18, 17, 16, 15, 14, 13, 12, 11	B0-B7	Entrada/Salida de datos
19	$\overline{OE}$	Habilitación (Activo (L))
20	Vcc	Fuente (+5V)

En el siguiente esquema no se representaron todos los pines del puerto, sino los correspondientes al puerto de datos y los de masa, que son los que utilizaremos en esta primera práctica.

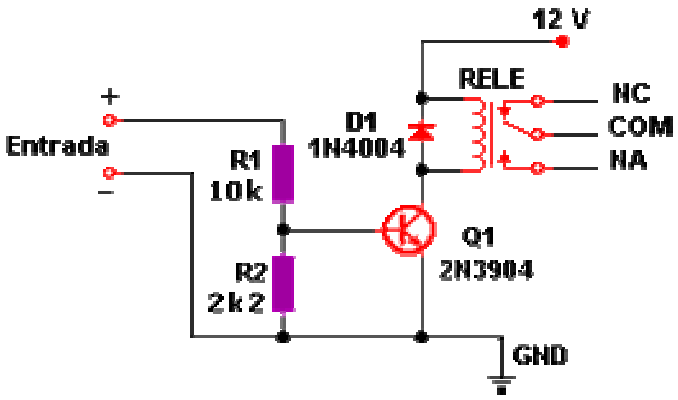


El código del programa sería el siguiente:

```
/*Programa que envía señales al puerto de datos ingresando un número decimal*/

#include<dos.h>
#include<stdio.h>
#include<conio.h>

int a;          /*variable, valor que enviare al puerto*/
void main()
{
a=0;            /*asignando 0 a la variable a*/
outp(888,a);    /*apago todos los LED's*/
clrscr();               /*limpio la pantalla*/

printf("Ingrese el número en decimal para enviar al puerto.\n");
printf("El número máximo permitido es 255:\n");
```

```
scanf("%d",&a);         /*tomo el valor y se lo paso a "a"*/
outp(888,a);    /*lo envío al puerto de datos*/

printf(" Dato enviado...!!!\n\n");
printf("Se acabó, presione una tecla para salir");

getch();                /*...y hasta luego*/
outp(888,0);    /*apago todos los LED's*/
}
```

Deseas encender algún aparato de casa? No hay problema, sólo le agregas una etapa de potencia a cada salida, algo como esto.



5.8 ENVIANDO DATOS POR EL PUERTO DE CONTROL.

Este puerto es de lectura-escritura. En este caso lo utilizaremos para enviar señales desde la PC hacia el exterior, y el circuito que se utilizara será similar al empleado en el puerto de datos, de ese modo es posible tener 12 salidas, lo que si se debe tener en cuenta es que el primero el segundo y el cuarto bit de este registro están invertidos, pero esta dificultad es superada con la codificación del programa que controla este puerto.

El programa de prueba lo tienes aquí.

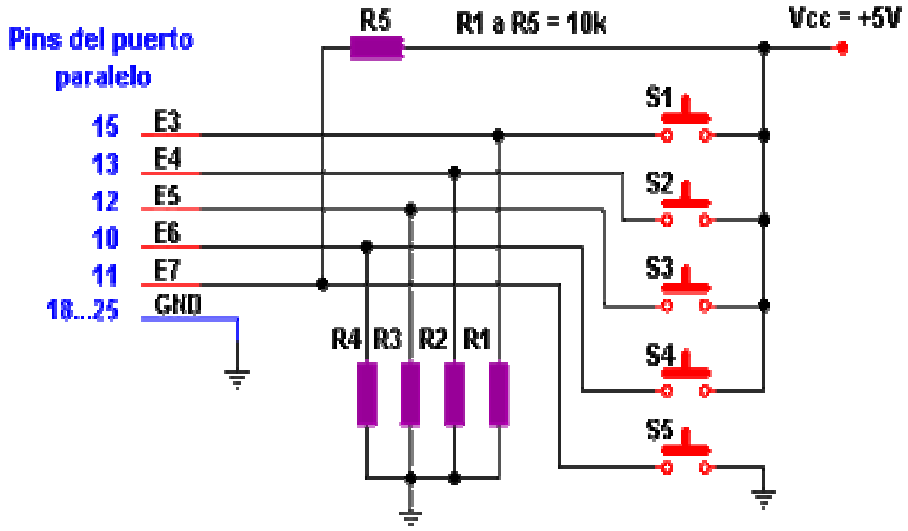
```
/*Programa que envía señales al puerto de control
ingresando un número decimal*/

#include<dos.h>
#include<stdio.h>
#include<conio.h>

int a;          /*variable, valor que enviare al puerto*/
void main()
{
a=11;           /*11 en binario es 1011 los valores 1
corresponden*/
                /*a los pines invertidos*/
outp(890,a);    /*apago todos los LED's*/
clrscr();               /*limpio la pantalla*/

printf("Ingrese el número en decimal para enviar al
puerto.\n");
printf("El número máximo permitido es 15:\n");

scanf("%d",&a);         /*tomo el valor y se lo paso a "a"*/
outp(890,a);    /*lo envío al puerto de estado*/

printf("tachannnn... Dato enviado...!!!\n\n");
printf("Se acabó, presione una tecla para salir");

getch();                /*...y hasta luego*/
outp(890,11);  /*apago todos los LED's*/
}
```

5.9 RECIBIENDO DATOS POR EL PUERTO PARALELO.

Para recibir datos en nuestro computadora por el puerto paralelo utilizaremos su registro de estado, recuerda que solo dispones de 5 bits (7, 6, 5, 4 y 3) y que el bit 7 funciona en modo invertido.

El circuito que construiremos será uno de los mas sencillos, solo habrá que tener en cuenta que un bit puede tener dos estados posibles, ya sea 0 (0 voltios) o 1 (5 voltios) no podemos dejarlo librado al azar, razón por la cual, si no se envía señal alguna, éste deberá estar unido a la masa (tierra).

El esquema eléctrico es el siguiente:



Manual de Prácticas

Práctica No. 1

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	I Sistemas Numéricos		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	1h	Vo. Bo. Profesor	

OBJETIVO: Al termino de la practica el alumno aprenderá a realizar conversiones entre bases (binario, octal, decimal y hexadecimal).

MARCO TEORICO: El sistema de números decimales se dice se dice que es de base o raíz, debido a que usa diez dígitos y los coeficientes se multiplican por potencias de 10. El sistema binario es un sistema diferente de números. Los coeficientes de los números del sistema binario tienen dos valores posibles: 0 y 1. Cada coeficiente a_j se multiplica por 2^j. Y en el caso de las otras bases se sigue un procedimiento similar, es costumbre tomar los r dígitos necesarios para los coeficientes del sistema decimal cuando la base del número es menor que 10. Se usan las letras del alfabeto para completar los diez dígitos decimales cuando la base del número es mayor que 10. Las letras A, B, C, D, E, y F, se utilizan para los dígitos 10, 11, 12, 13, 14,15, respectivamente.

DESARROLLO:

Convierta los siguientes números decimales en binarios:

a) 12 b) 673. 23 c) 1998

Convierta los siguientes números binarios en decimales:

a)10.0001111 b) 1111.000111 c)1110.00111.

Convierta los siguientes números de la base dada en las bases indicadas:

a) decimal 225.225 en binario, octal y hexadecimal.

b) Binario 1101111.001 en decimal, octal y hexadecimal.

c) Octal 623.77 en decimal, binario y hexadecimal.

CONCLUSIONES: Los números binarios son difíciles de trabajar ya que requieren tres o cuatro veces más dígitos que su equivalente decimal. No obstante las computadoras digitales utilizan números binarios y algunas veces es necesario que el operador humano o usuario se comunique en forma directa con la máquina mediante números binarios.

Práctica No. 2

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	I Sistemas Numéricos		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	2h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno será capaz de realizar operaciones aritméticas entre números de la misma base.

MARCO TEORICO: Los complementos se usan en las computadoras digitales para simplificar la operación de sustracción y para manipulaciones lógicas. Hay dos tipos de complementos para cada sistema base r. Complemento a uno y complemento a dos.

DESARROLLO: Haga la resta de los siguientes números binarios usando complemento a (1), y complemento a (2).

a) 11010 – 1101 b) 11010 – 10000

c) 10010 – 10011 d) 100000 – 00111

CONCLUSIONES: Una comparación entre los complementos de 1 y 2 revela las ventajas y desventajas de cada uno. El complemento de 1 tiene la ventaja de ser mas fácil de implementar por componentes digitales, ya que lo único que debe hacerse es cambiar los 0 en números 1 y los 1 en números 0.

Práctica No. 3

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	II Algoritmos		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	2h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno será capaz de plantear un problema y aplicar un algoritmo como el primer paso para la solución del problema.

MARCO TEORICO: Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado grafica o numéricamente.

DESARROLLO:

a) Escribir un algoritmo que calcule y escriba el cuadrado de 243.

b) Calcular el perímetro y la superficie de un cuadrado dada la longitud de su lado.

c) Calcular el perímetro y la superficie de un rectángulo dada la base y la altura del mismo.

d) Realizar el algoritmo que sume dos números.

CONCLUSIONES: Un algoritmo es una herramienta que nos permitirá resolver problemas en C++.

Práctica No. 4

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	II Algoritmos		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	3h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno será capaz de utilizar la representación grafica en la solución de problemas (diagrama de flujo).

MARCO TEORICO: Un diagrama de flujo es una de las técnica de representación de algoritmos más antigua y a la vez mas utilizada, aunque su empleo ha disminuido considerablemente, sobre todo desde la aparición de lenguajes de programación estructurados.

DESARROLLO: Representar el diagrama de flujo que nos permita resolver los siguientes problemas:

a) Determinar el área y volumen de un cilindro cuyas dimensiones radio y altura se leen desde el teclado.
b) Calcular la superficie de un círculo.
c) Determinar la hipotenusa de un triangulo rectángulo conocidas las longitudes de los catetos.

CONCLUSIONES: Los diagramas de flujo son herramientas que le permiten al programador, ir visualizando el desarrollo de la solución, por medio de diagramas.

Práctica No. 5

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	II Algoritmos		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	3h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno será capaz de aplicar el pseudo código para plantear la solución de problemas.

MARCO TEORICO: El pseudo código es un lenguaje de especificación (descripción) de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, traducción a un lenguaje de programación) relativamente fácil.

DESARROLLO: Realizar el pseudo código que resuelva los siguientes problemas:

a) Calcular y visualizar la suma y el producto de los números pares comprendidos entre 20 y 400, ambos inclusive, es decir sumar, 2 + 4 + 6+...400, así como la multiplicación.
b) Leer desde teclado 500 números enteros y obtener cuantos son positivos.
c) Calcular la suma de los cuadrados de los cien primeros números enteros.
d) Sumar diez números introducidos desde el teclado.

CONCLUSIONES: El pseudo código es una manera de resolver un problema, es como si la solución se fuera platicando de tal manera que los alumnos que incursionan por primera vez en la programación en cualquier lenguaje, puedan entender lo que se va realizando paso a paso.

Práctica No. 6

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	III Metodología de la programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	2h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno aplicara un proceso de desarrollo de un programa, para realizar el programa en la cantidad mínima de tiempo y con los menores errores.

MARCO TEORICO: Los lenguajes de programación, tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan sintaxis del lenguaje. Además de estos elementos básicos existen otros elementos que forman parte de los programas, cuya compresión y funcionamiento será vital para el correcto diseño de un algoritmo y naturalmente la codificación del programa, estos elementos son: (bucles, contadores, acumuladores, interruptores y estructuras secuenciales, selectivas y repetitivas)

DESARROLLO:

a) Diseñar un diagrama de flujo, algoritmo o pseudo código que nos permita calcular la media de cincuenta números e imprimir su resultado.
b) Diseñar un diagrama, algoritmo o pseudo código que nos permita calcular independientemente la suma de los números pares e impares comprendidos entre 1 y 200.

Nota: Utilizar contadores, interruptores, acumuladores, lo que sea mas conveniente.

CONCLUSIONES: Los elementos básicos y los elementos que forman parte del lenguaje, le permitirán al programador novato ir aprendiendo como se resuelven problemas, utilizando C++.

Práctica No. 7

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	4h	Vo. Bo. Profesor	

OBJETIVO: El alumno aprenderá a programar en C++, utilizando la sentencia de control IF.
MARCO TEORICO: La sentencia IF funciona de la siguiente manera. Cuando se alcanza la sentencia IF dentro de un programa se evalúa la expresión si es verdadera se ejecuta la acción 1 y en caso contrario se ejecuta la acción 2.

DESARROLLO: El alumno con la asesoría del profesor resolverá el siguiente problema en código C++. Se desea calcular el salario neto semanal de los trabajadores de una empresa de acuerdo a las siguientes normas:

Horas semanales trabajadas < 40 se pagan a una tarifa normal, 50 pesos la hora.
Horas extras más de 40 a una tarifa del doble de las horas normales.
Impuesto será de 10% para las horas normales.
Impuesto será del 15% para las horas extras.

CONCLUSIONES: El alumno aprendió a resolver problemas utilizando la sentencia de control IF, que nos sirven para controlar el flujo de ejecución de un programa.

Práctica No. 8

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	4h	Vo. Bo. Profesor	

OBJETIVO: El alumno aprenderá a programar en C++, utilizando la sentencia de control SWITCH.

MARCO TEORICO: La estructura de selección múltiple evaluara una expresión que podrá tener n valores distintos, casos distintos, 1, 2, 3, …,n. Según se elija uno de estos valores en la condición, se realizara una de la n acciones, o lo que es igual, el flujo de algoritmos seguirá un determinado camino entre los n posibles.

DESARROLLO: Se desea diseñar un código en C++, que convierta las calificaciones alfabéticas A, B, C, D y E, en calificaciones numéricas 10,9, 8, 7, 6, 5. Es decir A=10, B=9,…, E=5.

CONCLUSIONES: Cuando se utiliza una sentencia SWITCH, asegúrese que el selector de switch y las etiquetas CASE son del mismo tipo (int, char o bool paro no float). Si el selector se evalúa a un valor no listado en ninguna de las etiquetas case, la sentencia switch no gestionará ninguna acción; por esta causa se suele poner una etiqueta (default) para resolver este problema.

Práctica No. 9

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	4h	Vo. Bo. Profesor	

OBJETIVO: Al término de la practica el alumno aprenderá a programar en C++, utilizando la sentencia de control WHILE (mientras).

MARCO TEORICO: Un bucle WHILE (mientras) tiene una condición (expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle, de modo que se evalúa la condición antes de que el cuerpo del bucle sea ejecutado.

DESARROLLO: Realizar un programa en C++, que nos calcule la media de un conjunto de calificaciones de alumnos. Pondremos un valor de centinela, es decir una condición de parada igual a (-99) que detecte el fin del bucle, es decir cuando se introduzca el valor de (-99) el bucle se dejara de ejecutar.

CONCLUSIONES: Los valores de centinela se utilizan para controlar la repetición cuando el número de repeticiones (iteraciones) no se conocen por adelantado y el bucle contiene sentencias que obtienen datos cada vez que se ejecuta. Un valor centinela se introduce después de que se han proporcionado todos los datos válidos al programa y debe ser distinto de los datos válidos.

Práctica No. 10

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	4h	Vo. Bo. Profesor	

OBJETIVO: Al término de la practica el alumno aprenderá a programar en C++, utilizando la sentencia de control FOR (desde).

MARCO TEORICO: La sentencia FOR (desde) es un método para ejecutar un bucle de sentencias un numero fijo de veces. El bucle for se diferencia del bucle while en que las operaciones de control se sitúan en un solo sitio, la cabecera de la sentencia.

DESARROLLO: Resolver el siguiente problema en C++, sumar los primeros cien numero enteros, es decir sumar 1 + 2 +3 +…100.

CONCLUSIONES: La sentencia FOR inicializa un valor y a continuación comprueba una expresión; si la expresión es verdadera se ejecutan las sentencias del cuerpo del bucle y se comprueba la expresión cada vez que termina la iteración. Cuando la expresión es falsa se termina el bucle y se sale del mismo.

Práctica No. 11

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	5h	Vo. Bo. Profesor	

OBJETIVO: Al término de la practica el alumno aprenderá a programar en C++, utilizando la sentencia de control DO – WHILE. (hacer mientras).

MARCO TEORICO: La sentencia DO – WHILE se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener la seguridad de que una determinada acción se ejecutará una o varias veces, pero al menos una vez.

DESARROLLO: Realizar un programa en C++, que seleccione una opción de saludo al usuario dentro de un programa y lo visualice en pantalla.

CONCLUSIONES: Los bucles do-while ejecutan el cuerpo del bucle y a continuación hacen una comprobación de la condición.

Práctica No. 12

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	5h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno aprenderá a resolver problemas, programando en C++, utilizando funciones para resolver estos.

MARCO TEORICO: Cada vez que se llama a una función desde el algoritmo principal se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función y se presupone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

DESARROLLO: Resolver el siguiente problema, programando en C++. Diseñar un algoritmo que contenga un subprograma que calcule el factorial de un número y una llamada al mismo.

CONCLUSIONES: Las funciones nos permiten dividir los programas en subprogramas, y que por medio de una llamada al subprograma nos devuelva el resultado de la función.

Práctica No. 13

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	IV Lenguajes de programación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	5h	Vo. Bo. Profesor	

OBJETIVO: Al término de la práctica el alumno aprenderá a programar en C++, problemas de arreglos de una dimensión y de dos dimensiones.

MARCO TEORICO: Un arreglo es una secuencia de objetos del mismo tipo. Los objetos se llaman elementos del arreglo y se numeran consecutivamente 0, 1, 2, 3… El tipo de elementos en el arreglo puede ser cualquier tipo de dato de C++, incluyendo estructuras definidas por el usuario. Normalmente el arreglo se utiliza para almacenar tipos de datos tales como (int, char o float).

DESARROLLO: Resolver el siguiente problema, programar en C++ y determinar el número mayor de una lista definida.

CONCLUSIONES: Un arreglo es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado.

Práctica No. 14

Carrera	Electricidad y Electrónica Industrial		
Cuatrimestre	Segundo	Grupo	
Tipo Práctica	□ Laboratorio □ Simulación	Fecha	
Asignatura	Informática II		
Unidad Temática	V Canales de comunicación		
No de alumnos por práctica	3		
Nombre Del Profesor			
Nombre (S) De Alumno (S)			
Tiempo Estimado	10 h	Vo. Bo. Profesor	

OBJETIVO: Al termino de la practica el alumno será capaz de controlar un motor de corriente continua, por medio de transistores y programando en C++.

MARCO TEORICO:

Una de las ventajas de esta interfaz frente a las precedentes (con relés y opto acopladores) radica en que si las señales de control conmutan a frecuencia elevada, los transistores son capaces de seguirlas, mientras que los relés ya no lo hacen cuando el período de la conmutación se halla en torno a los milisegundos.

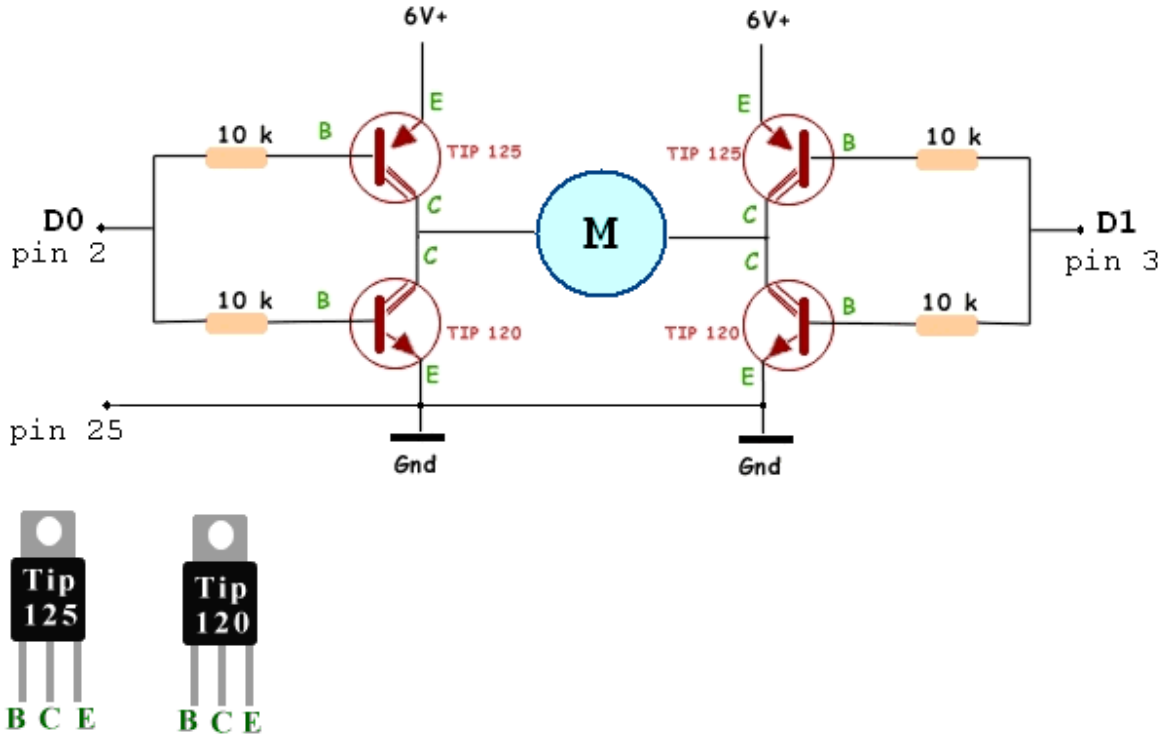
DESARROLLO:

Control de un motor de corriente continua:

Para disminuir el riesgo de daños al puerto de control y mejorar la capacidad de manejo de corriente que necesita el motor se puede utilizar una etapa de desacoplamiento.

Mediante transistores:

Una alternativa simple consiste en utilizar un circuito como el de la figura siguiente.



En éste se utilizan transistores para la etapa de potencia del motor. Cuando el bit D0 se pone a 0 y el D1 a 1, el borne izquierdo del motor se encuentra a +6V y el derecho a tierra, con lo que el motor gira en un sentido. Cuando el bit D0 vale 1 y el D1 0, es el borne derecho el que está alimentado mientras que el izquierdo se halla conectado a tierra, de modo que el motor gira en sentido opuesto.

CONCLUSIONES: Aprender a programar en C++, nos permitirá controlar un dispositivo por medio del puerto serie o por el puerto paralelo.

BIBLIOGRAFIA:

Diseño Digital
M. Morris mano
Prentice Hall

Fundamentos de programación
Luis Joyanes Aguilar
Mc. Graw Hill

Programación en C++
Luis Joyanes Aguilar
Mc. Graw Hill

Como programar en C++
Deitel y Deitel
Prentice Hall