



**Universidad Nacional Autónoma de México**  
**Facultad de Ingeniería**  
**Algoritmos y estructuras de datos**  
**Tema 1:**  
**ELEMENTOS PARA EL ESTUDIO DE**  
**ESTRUCTURAS DE DATOS**

# 1 Elementos para el estudio de estructuras de datos

**Objetivo:** Comprender los aspectos básicos de la estructura de una computadora digital, que permita obtener un marco de referencia para iniciar el estudio de las estructuras de datos.

# 1 Elementos para el estudio de estructuras de datos

- 1.1 Componentes físicos de una computadora.
- 1.2 Elementos internos de la computadora.
- 1.3 Conceptos básicos de programación de bajo nivel.
- 1.4 Conceptos de programación de alto nivel (estructurada).
  - 1.4.1 Representación de tipos de datos.
  - 1.4.2 Ciclos de control.
- 1.5 Manejo de memoria, acceso, asignación dinámica, apuntadores, arreglos.

## Bibliografía:

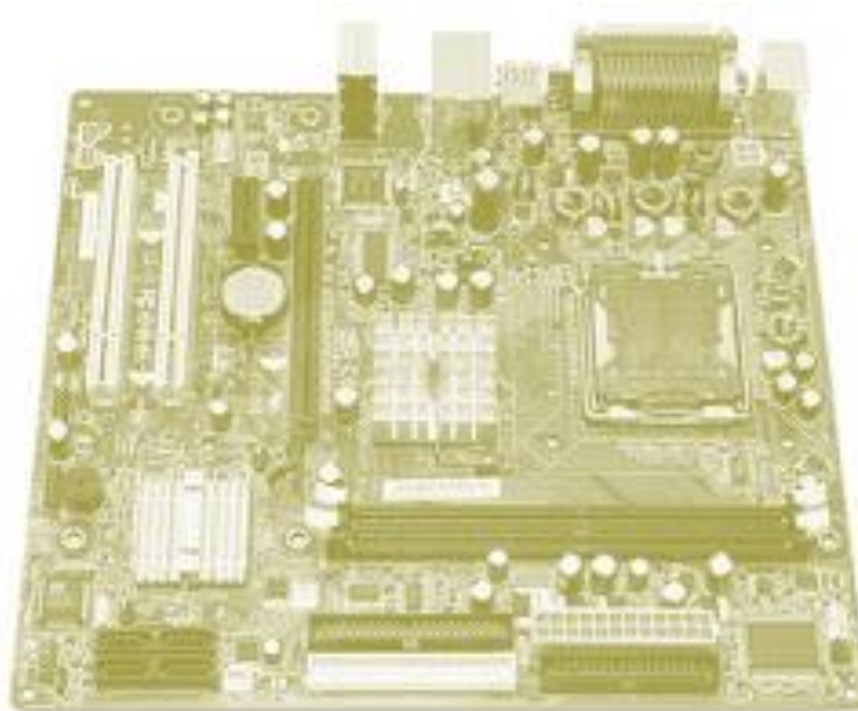


*El lenguaje de programación C.*

Brian W. Kernighan, Dennis M. Ritchie,  
Pearson Educación.

# Licencia GPL de GNU

```
/*  
*  
* This program is free software: you can redistribute it and/or modify  
* it under the terms of the GNU General Public License as published by  
* the Free Software Foundation, either version 3 of the License, or  
* (at your option) any later version.  
*  
* This program is distributed in the hope that it will be useful,  
* but WITHOUT ANY WARRANTY; without even the implied warranty of  
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
* GNU General Public License for more details.  
*  
* You should have received a copy of the GNU General Public License  
* along with this program. If not, see <http://www.gnu.org/licenses/>.  
*  
* Author: Jorge A. Solano  
* E-mail: jorge.a.solano@hotmail.com  
*  
*/
```

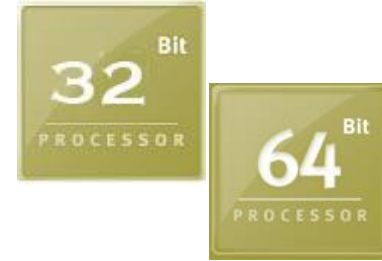


## 1.1 Componentes físicos de una computadora.

## 1.1 Componentes físicos de una computadora.

**El hardware o estructura física de una computadora es el conjunto de elementos que componen una computadora.**

**Físicamente, las computadoras están compuestas por la unidad central de proceso y los dispositivos de entrada y salida de datos.**



## Unidad Central de Proceso

Es la unidad principal, permite controlar dispositivos periféricos, ejecutar instrucciones así como el manejo interno de datos. Está conformada por dos unidades (la unidad de control y la unidad aritmético-lógica) y por un conjunto de registros.



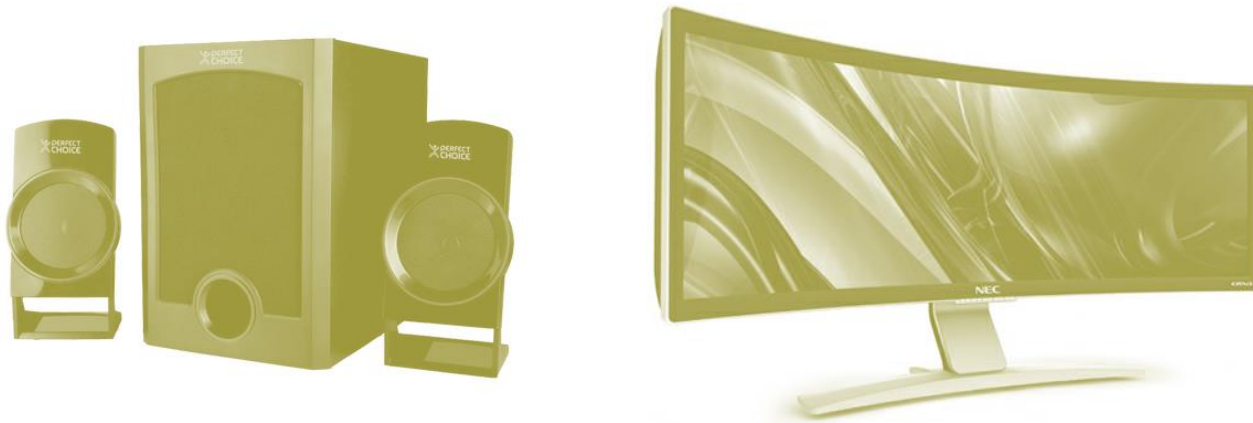
## Unidad de entrada

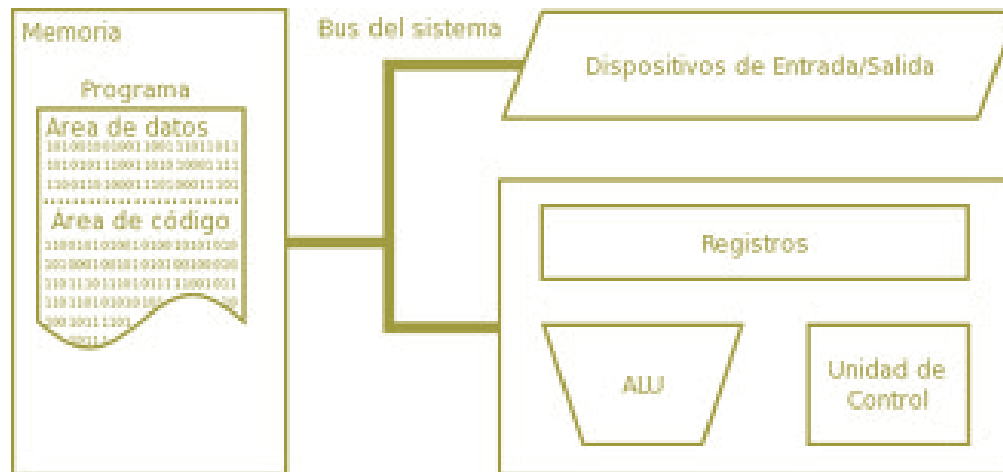
**Medio que permite establecer comunicación entre el usuario y la máquina.**



## Unidad de salida

Medio que muestra los resultados de un proceso de una manera entendible para el usuario final.

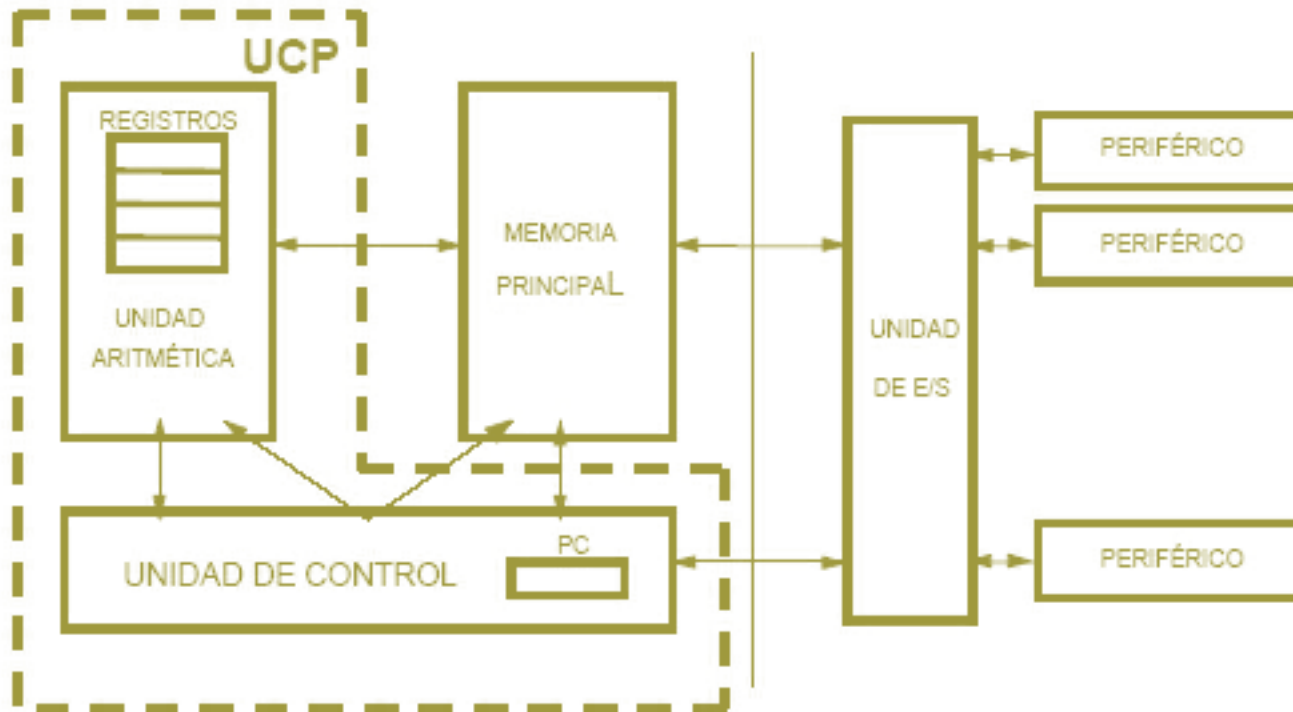




## 1.2 Elementos internos de la computadora.

## 1.2 Elementos internos de la computadora.

De manera interna, una computadora está compuesta por cuatro unidades básicas: unidad aritmético-lógica, unidad de control, unidad de memoria y las unidades de entrada y salida.



Arquitectura Von Neumann

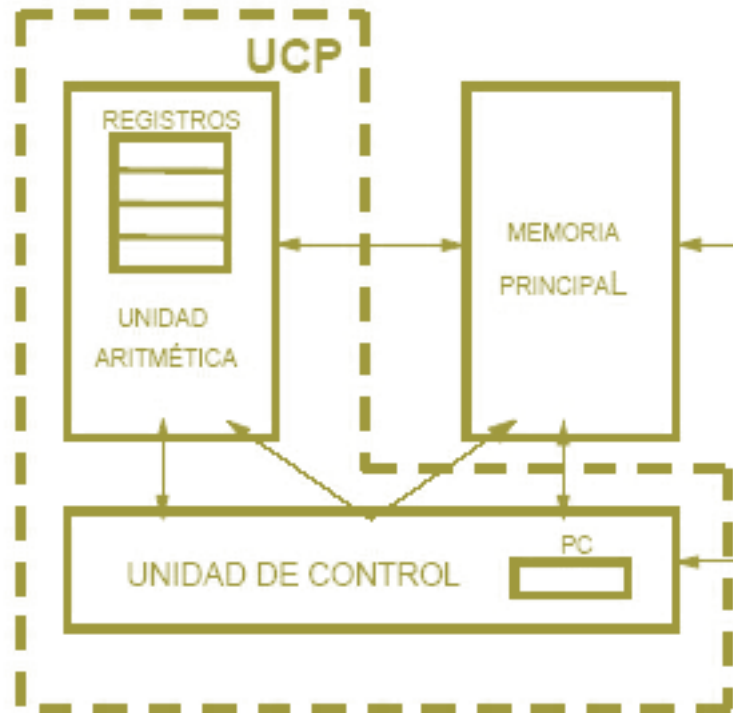
## Unidad Central de Proceso

La computadora trabaja en lenguaje binario, es decir, la información que se ingresa es transformada a lenguaje máquina (cero y uno) y, antes de regresarla al usuario, se vuelve a transformar en un lenguaje entendible para éste (decimal o ascii).

Las transformaciones mencionadas son llevadas a cabo por medio de la unidad central de proceso, para posteriormente ejecutar las instrucciones del programa, efectuar movimientos aritméticos y lógicos con los datos y comunicarse con todos los elementos del sistema.

La UCP es un conjunto de circuitos electrónicos. Cuando se unen estos conjuntos electrónicos se obtiene un chip al que se le llama microprocesador.

La unidad central de proceso se divide en dos partes fundamentales: la unidad de control y la unidad aritmético-lógica.





## Unidad de Control

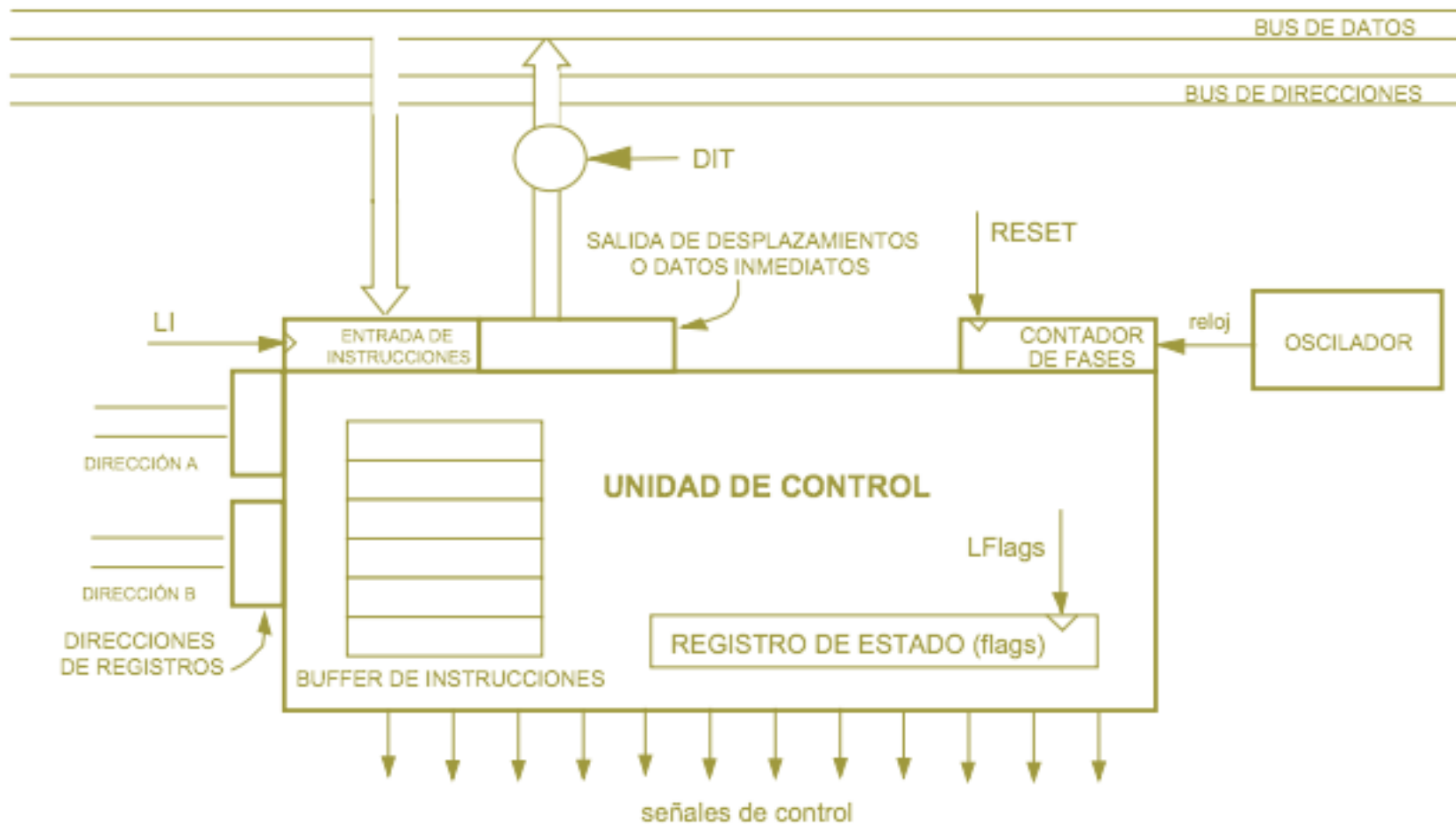
La unidad de control se encarga de administrar la secuencia de operaciones, por ejemplo, no realizar un cálculo hasta recibir el resultado de una operación previa o no enviar información a la unidad de salida mientras se estén realizando los cálculos.

La Unidad de Control es el núcleo del procesador y tiene 3 funciones principales:

- Leer e interpretar las instrucciones del programa.
- Dirigir el accionar de los componentes internos.
- Administra el flujo de datos desde y hacia la memoria RAM.

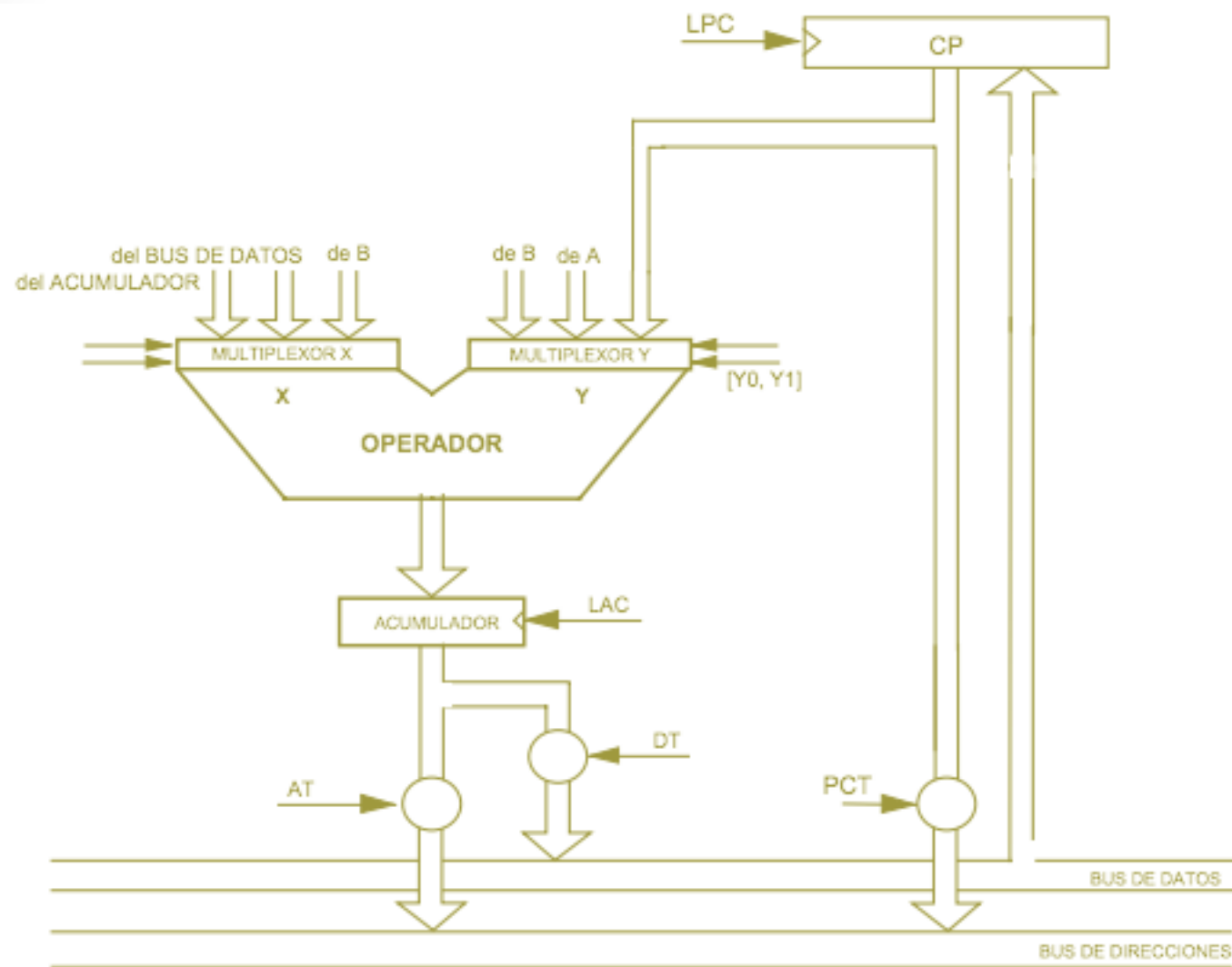
Los pasos para ejecutar una instrucción cualquiera son los siguientes:

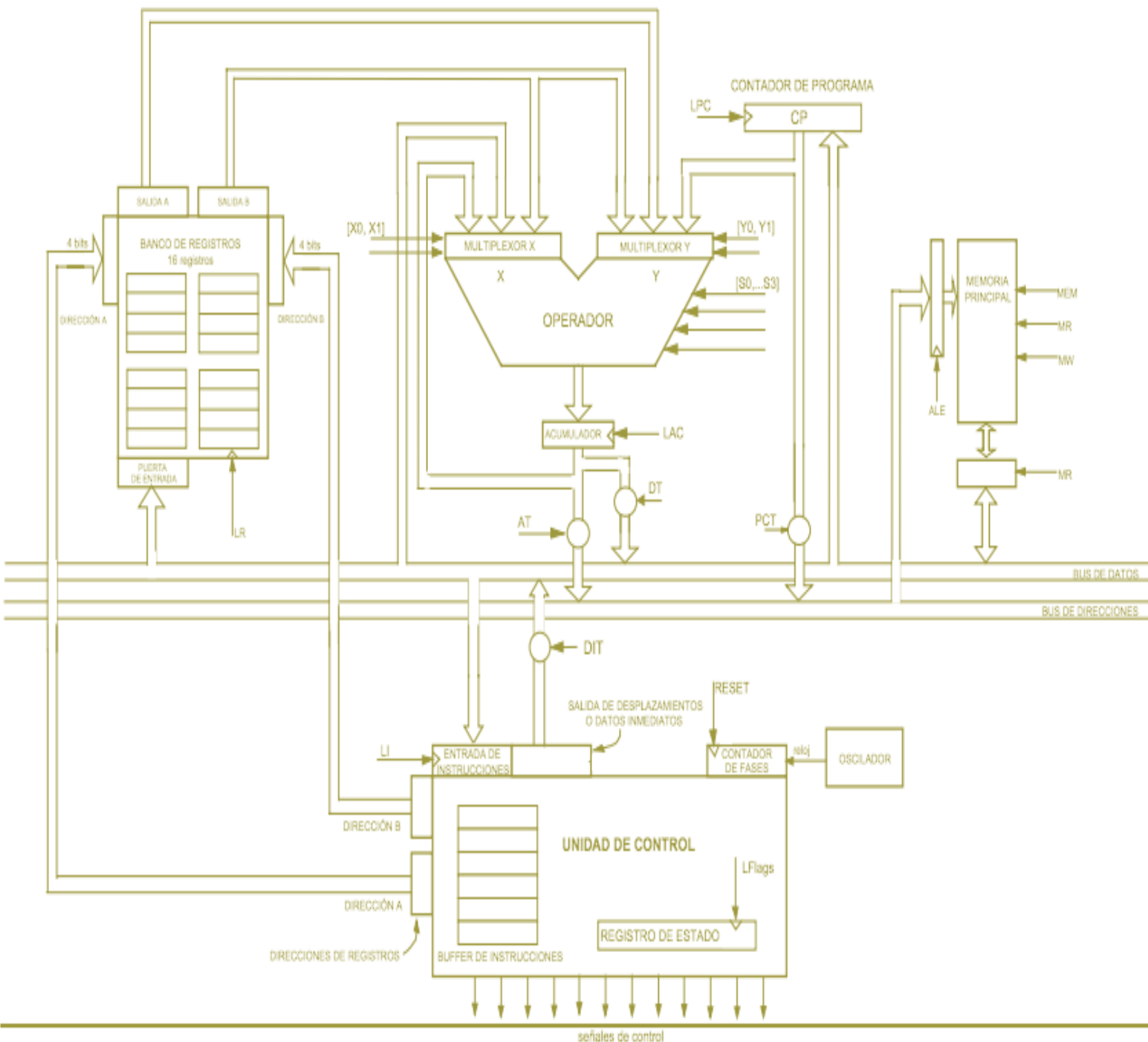
1. Ir a la memoria y extraer el código de la siguiente instrucción a ejecutar.
2. Decodificar la instrucción.
3. Ejecutar la instrucción.
4. Prepararse para leer la siguiente instrucción y volver al primer paso.



## Unidad Aritmético-Lógica (ALU)

La unidad aritmético-lógica (ALU) procesa la información matemática y lógicamente. Se encarga de realizar todos los cálculos aritméticos básicos (suma, resta, multiplicación y división) y operaciones lógicas (comparaciones).





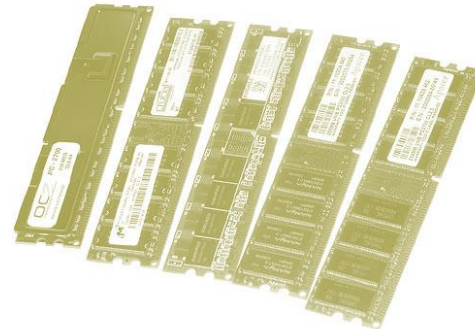
## Programa en mnemónicos

ORG 2000h	; espacio de memoria donde se va a trabajar
XOR AX, AX	; se limpia el registro AX
XOR BX, BX	; se limpia el registro BX
MOV AX, 1h	; se asigna el valor 1 al registro AX
MOV BX, 2h	; se asigna el valor 2 al registro BX
ADD BX,AX	; realiza $BX = BX + AX$
END	

## Unidad de memoria

Es la parte de la computadora donde se almacena información, ya sea de entrada o de salida.

Dentro de la memoria reside el programa a ser ejecutado, los datos necesarios para el proceso y los resultados que emita el programa.





**La memoria es un elemento electrónico que posee un conjunto de registros direccionables donde residen tanto las instrucciones de un programa como los datos (información).**

**Dentro de una computadora, existen dos tipos de memoria:**

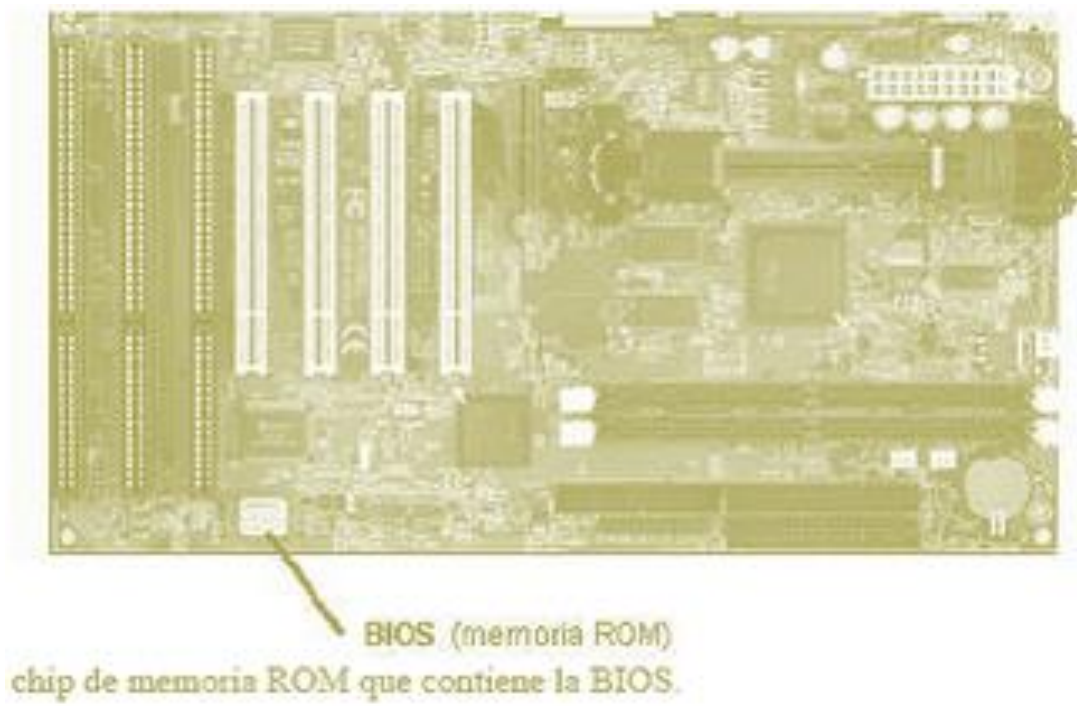
- **Memoria ROM**
- **Memoria RAM**

## Memoria ROM (Read Only Memory)

Es un tipo de memoria no volátil, es decir, la información escrita en este tipo de memoria no se puede modificar.

Se trata de un circuito integrado que se encuentra instalado en la tarjeta principal (Motherboard).

Las computadoras ocupan las memorias ROM en los sistemas de arranque y/o con alguna otra información crítica del fabricante.



- **Memoria PROM (Programmable Read Only Memory)**

Es una variación de la memoria ROM, ya que en las memorias PROM la información sí puede ser modificada aunque solo una vez.

En este tipo de memorias cada uno de los bits depende de un fusible puede ser quemado una única vez.

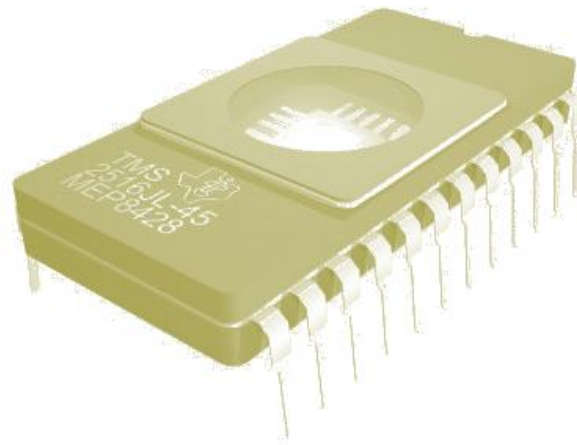
Una memoria PROM sin programar se encuentra con todos los fusibles sin ser quemados, o sea, con valor 1. Cada fusible quemado corresponde a un 0 produciendo una discontinuidad en el circuito. Estas memorias se van programando aplicando pulsos eléctricos.



- **Memoria EPROM (Erasable Programmable Read Only Memory)**

Este tipo de memoria ROM es un chip no volátil y está conformada por transistores de puertas flotantes o celdas FAMOS que salen de fábrica sin carga alguna.

Esta memoria puede ser borrada sólo si se la expone a luces ultravioletas.



Una vez que la memoria EPROM es programada, se vuelve no volátil, es decir, los datos almacenados permanecerán ahí de forma indefinida.

El proceso de borrado del chip es siempre total. El tiempo para borrar su contenido es de por lo menos veinte minutos.

- **Memoria EEPROM (Electrically Erasable Programmable Read Only Memory)**

Esta memoria puede ser programada y borrada eléctricamente resultando también no volátiles. Además de tener puertas flotantes, cuenta con una capa de óxido ubicado en el drenaje de la celda MOSFET, lo que permite que la memoria pueda borrarse eléctricamente.





## Memoria RAM (Random Access Memory)

Es una memoria de almacenamiento primario. Permite almacenar, temporalmente, instrucciones de programa y datos.

Una memoria RAM se divide en varias localidades de igual tamaño. Dichas localidades de memoria tienen una dirección única para que la computadora pueda distinguirlas y acceder a ellas.

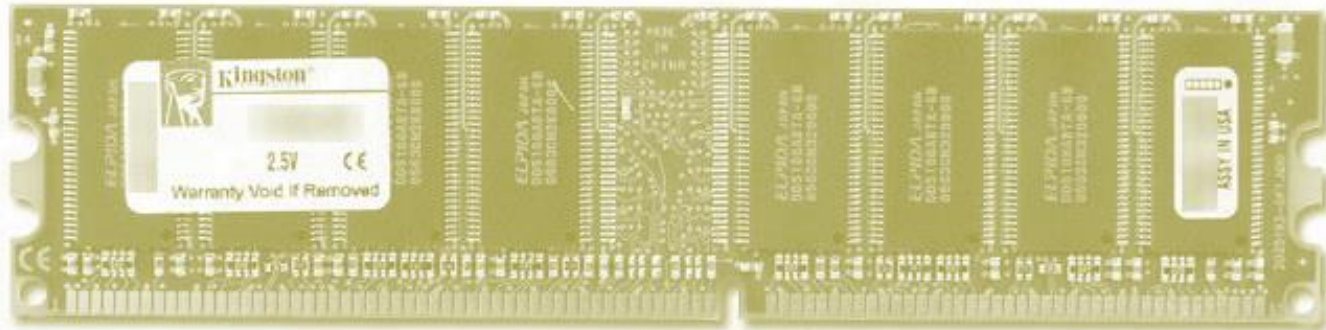
La información que contiene una memoria RAM está dada en forma de corriente eléctrica que fluye a través de circuitos microscópicos en chips de silicio.

Es una memoria volátil ya que la información que contiene no se conserva de manera permanente (en cuanto pierde energía se eliminan los datos almacenados).

Las memorias RAM no poseen partes móviles, por lo tanto, el acceso a estos dispositivos es veloz.

Los programas y datos se transmiten a la memoria RAM antes de ser ejecutados o utilizados.

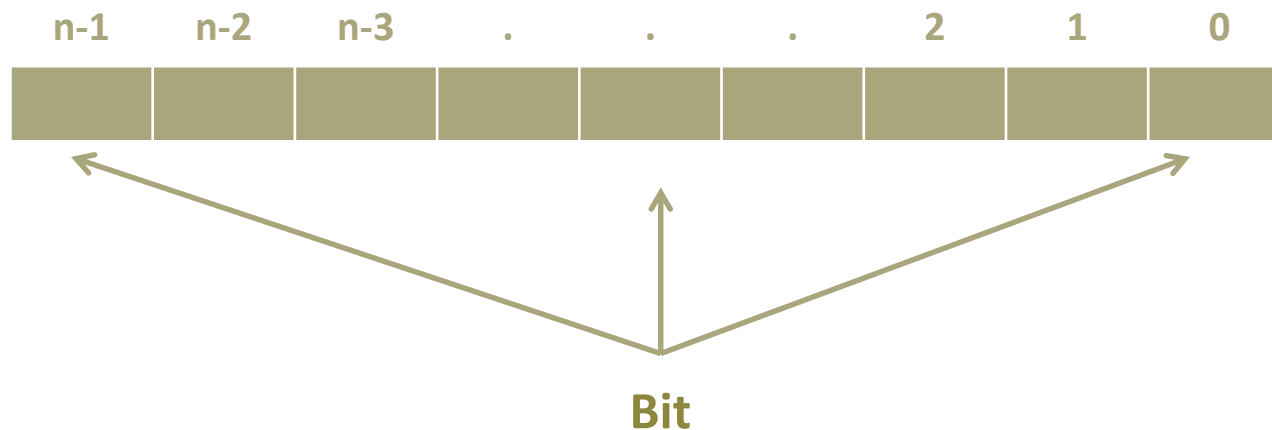
Como el espacio de este tipo de memorias es escaso, una vez que se ejecuta el programa, se libera la memoria.



La unidad central de proceso toma instrucciones y datos de la memoria primaria en grupos de  $n$  bits llamados palabra de computadora.

Un bit es el nombre que recibe un dígito en binario, el cual solo puede tomar dos posibles estados: cero y uno. Un byte es un conjunto de 8 bits.

La longitud de una palabra de computadora corresponde al número de bits que la componen. En la mayoría de los equipos de cómputo esta longitud oscila entre 8 y 64 bits.



Un bit puede representar dos estados distintos ( $2^1$ ). Una palabra de computadora de longitud  $n$  puede representar  $2^n-1$  estados distintos.

	n-1	n-2	n-3	.	.	.	2	1	0
0									
1									
2									
3									
...									
m-3									
m-2									
m-1									

La memoria, físicamente, está constituida por un conjunto de  $m$  palabras de longitud  $n$ . A cada palabra de computadora se le asocia una dirección de memoria.

La representación de información en la memoria se realiza mediante un cierto número de bits, dependiendo del tipo de dato así como de la longitud de la palabra.

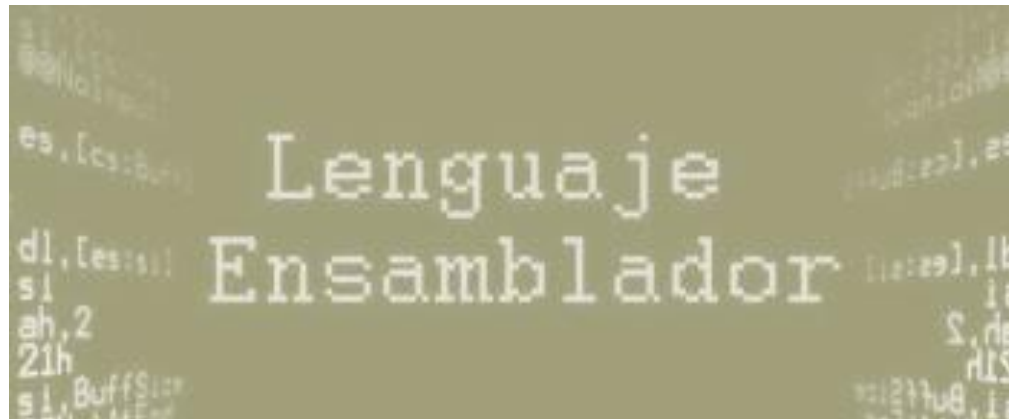
Para escribir (almacenar) o leer (recuperar) de una palabra de computadora, existe el registro de dirección, el registro de datos y líneas de control de flujo de datos.





Unidad	2 <sup>n</sup>	Número de bytes	Equivalencia
Kilobyte (KB)	2 <sup>10</sup>	1 024	1024 bytes
Megabyte (MB)	2 <sup>20</sup>	1 048 576	1024 KB
Gigabyte (GB)	2 <sup>30</sup>	1 073 741 824	1024 MB
Terabyte (TB)	2 <sup>40</sup>	1 099 511 627 776	1024 GB
Petabyte (PB)	2 <sup>50</sup>	1 125 899 906 842 624	1024 TB
Exabyte (EB)	2 <sup>60</sup>	1 152 921 504 606 846 976	1024 PB
Zettabyte (ZB)	2 <sup>70</sup>	1 180 591 620 717 411 303 424	1024 EB
Yottabyte (YB)	2 <sup>80</sup>	1 208 925 819 614 629 174 706 176	1024 ZB

### Unidades de almacenamiento



## 1.3 Conceptos básicos de programación de bajo nivel.

### 1.3 Conceptos básicos de programación de bajo nivel.

Los lenguajes más próximos a la arquitectura de la computadora (hardware) se denominan lenguajes de bajo nivel.

Los lenguajes de bajo nivel son códigos dependientes del procesador, es decir, el programa que se realiza con este tipo de lenguajes no se pueden migrar o utilizar en otras arquitecturas.

Debido a que están prácticamente diseñados a medida del hardware, aprovechan al máximo las características del mismo.

Para la codificación de instrucciones es preciso tener en cuenta tanto el número de bits de la instrucción así como la correspondencia entre instrucciones y su codificación binaria.

El lenguaje más representativo de este nivel es el lenguaje ensamblador, que está formado por abreviaturas de letras y números llamadas mnemónicos.

Una instrucción de procesador está formada por la operación a realizar y el o los registros a utilizar (de donde se obtiene o almacena la información). La estructura de una instrucción es la siguiente:

Operación	Número (8 bits)	Número (8 bits)
add	0x23	0x34
sub	0x4F	0xBA
mul	0x12	0xFF
div	0xFF	0x08

Las instrucciones están representadas por un número dentro del procesador dependiendo del fabricante. Por lo tanto, las instrucciones definidas así como su correspondiente codificación a lenguaje binario dependen del procesador que se esté utilizando.

Por ejemplo, el código de instrucción 00 que hace referencia a la instrucción add. Si se quiere sumar dos cantidades, la instrucción y codificación sería la siguiente:

Instrucción:	ADD	0x27	0xB2
Codificación:	00	00100111	10110010
	0000 1001 1110 1100	1000 0000 0000 0000	
Representación:	0x09EC8000		



```
00100000000010100011011000000100101100011
110001011101000100011111111110100000100
00101001011000011010111011010110110010001
01101100000101011001000100001110001001111
10100110010110100110110100111101111011110
00011010001000100010001000100010001000100
0100100110 #include <stdio.h> 00101000011010
10001001int main() 0000101111
010101001{ 000011000
111001100 printf("Hello World"); 0001100
00100000111 return 42; 010101110110
0001101000100011101001110001101000011010
01001001101111010111011110000001010001110
11000100100010101100100111011101000101111
01010100111001101010111000101010100011000
1110011000001101111110101001111110001100
00100000111111101010010010011010101110110
```

## 1.4 Conceptos de programación de alto nivel (estructurada).

## 1.4 Conceptos de programación de alto nivel (estructurada).

**El paradigma estructurado sugiere que el análisis y diseño que se realicen sean un conjunto de procedimientos descendentes, esto quiere decir que se realiza una descomposición funcional de procesos en otros procesos de menor nivel (divide y vencerás).**

**El paradigma estructurado divide el código en bloques, estructuras que pueden o no comunicarse entre sí.**

**El teorema del programa estructurado, demostrado por Böhm-Jacopini, dicta que todo programa puede desarrollarse utilizando únicamente 3 instrucciones de control: secuencia, selección e iteración.**

- **Secuencial:** Una instrucción no se ejecuta hasta que termina la anterior.
- **Selección:** Permite que el programa se bifurque en una u otra instrucción según se cumpla o no una condición lógica.
- **Iteración:** Bucles o ciclos de control que ejecutan una secuencia de instrucciones mientras se cumple la condición establecida.

### 1.4.1 Representación de tipos de datos

Un lenguaje de programación de alto nivel maneja representaciones, por lo menos, tres tipos de datos: caracteres, números enteros y números reales.

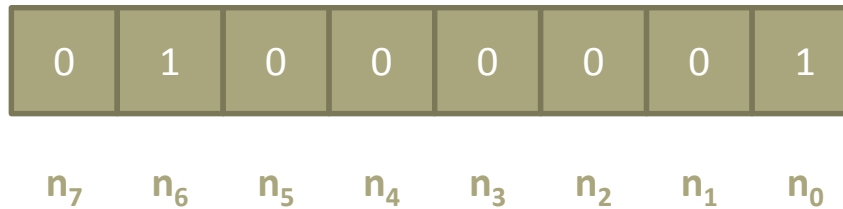
## Representación de un tipo de dato caracter

En memoria, un carácter ocupa 8 bits y se representa en sistema binario dependiendo del tipo de codificación que esté ocupando la máquina. Las representaciones de datos más utilizadas son código ASCII, código EBCDIC y UNICODE.

## 1. Elementos para el estudio de estructuras de datos.

## Ejemplo 1

En código ASCII, el carácter 'A' se representa por el código  $41_{16}$ . Este número en binario se representa por los dígitos 0100 0001. Su representación en memoria es la siguiente:





## Representación de un tipo de dato entero

Dependiendo de la arquitectura de la computadora, los números enteros pueden ocupar 16 ó 32 bits en memoria, donde el primer bit está reservado para el signo y los restantes definen la capacidad de almacenamiento del número.

Un número entero que ocupa 32 bits se distribuye en la memoria de la siguiente manera:


1 bit para el signo del número  
31 bits para el número

$\pm$	$n_{30}$	$n_{29}$	$n_{28}$	$n_{27}$	$n_{26}$	$n_{25}$	$n_{24}$	$n_{23}$	$n_{22}$	$n_{21}$	$n_{20}$	$n_{19}$	$n_{18}$	$n_{17}$	$n_{16}$
$n_{15}$	$n_{14}$	$n_{13}$	$n_{12}$	$n_{11}$	$n_{10}$	$n_9$	$n_8$	$n_7$	$n_6$	$n_5$	$n_4$	$n_3$	$n_2$	$n_1$	$n_0$

## Ejemplo 2

Convertir el número  $28,345_{10}$  a binario y mostrar su representación en memoria (32 bits).

$$28,345_{10} = 110111010111001_2$$

28 345	/ 2	
14 172	1	
7 086	0	
3 543	0	
1 771	1	
885	1	
442	1	
221	0	
110	1	
55	1	
27	1	
13	1	
6	1	
3	0	
1	1	
0	1	

## Ejemplo 2

La representación en memoria del número  $28,345_{10}$  es:

$$28,345_{10} = 110111010111001_2$$

+

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	1	1	0	1	0	1	1	1	0	0	1

Es importante tener en cuenta que un número entero puede superar la capacidad de almacenamiento determinada y ello provoca pérdida de información (parcial o total).

## Representación de un tipo de dato entero en complemento ar

Dependiendo de la arquitectura de la computadora, un dato puede ser almacenado en memoria con su complemento (cuando el número es negativo).

## Complemento aritmético

El **completo aritmético (ar)** de un número real se refiere a la cantidad que le falta a dicho número para ser igual a una unidad del orden inmediato superior.

El complemento aritmético ar (o complemento a la base) de un número real se obtiene a partir de la siguiente fórmula:

$$ar = r^n - |N|$$

donde:

ar: complemento aritmético de un número real base r.

r: es la base del número.

n: número de dígitos de la parte entera del número.

N: el número dado.



### Ejemplo 3

Dado el número  $789_{10}$ , obtener su complemento aritmético a la base ar (Complemento a10).

$$\begin{aligned}ar &= r^n - |N| \\c &= 10^3 - |789| \\c &= 1000 - 789 = 211\end{aligned}$$

$$ar = 211_{10}$$

#### Ejemplo 4

Obtener el resultado de restar  $1001_2$  con  $1111_2$ .

$$\begin{array}{r} \phantom{100}1001 \\ - \phantom{100}1111 \\ \hline \phantom{100}111010 \end{array}$$

El resultado de la resta está dado en complemento a la base (complemento a2), por lo tanto, se debe aplicar la fórmula de complemento aritmético para obtenerlo en magnitud y signo, ya que:

$$n = (ar)^r$$

### Ejemplo 5

Dado complemento aritmético  $111010_2$ , obtener el número en magnitud y signo.

$$\begin{aligned}n &= (ar)r \\ ar &= r^n - |N| \\ n &= 2^6 - |111010|\end{aligned}$$

$$\begin{array}{r} 1000000 \\ - \quad 111110 \\ \hline 0000110 \end{array}$$

## Ejemplo 5

Para un número en complemento el bit de signo es parte del número.

En el ejemplo anterior, el bit más significativo del número está prendido (1), por lo tanto, se puede afirmar que el número es negativo.

Entonces, la magnitud y signo del número  $111010_2$  es  $-110_2$ .

## Ejercicio 1

**Obtener el complemento aritmético de los siguientes números:**

Linea	Base 2	Base 10
1	11100101	83456
2	11110000	71912
3	11000011	70007
4	11100001	49567
5	10000111	49234
6	11111000	61816
7	10001111	57362
8	11101111	56253

El complemento aritmético menos uno (ar-1 o complemento a la base disminuida) de un número real se calcula con base en la siguiente fórmula:

$$ar-1 = r^n - r^{-m} - |N|$$

donde:

ar-1: complemento aritmético de un número real base r.

r: es la base del número.

n: número de dígitos de la parte entera del número.

m: número de dígitos de la parte fraccionaria del núm.

N: el número dado.

## Ejemplo 6

Dado el número  $789_{10}$ , obtener su complemento aritmético a la base  $ar-1$  (Complemento a  $10-1$ ).

$$\begin{aligned}ar-1 &= r^n - r^m - |n| \\c &= 10^3 - 10^0 - |789| \\c &= 1000 - 1 - 789 = 210\end{aligned}$$

$$ar-1 = 210_{10}$$

## Ejemplo 7

Dado el número  $1001.01_2$ , obtener su complemento aritmético a la base ar-1(Complemento a2-1).

$$\begin{aligned}\text{ar-1} &= r^n - r^{-m} - |n| \\ c &= 2^4 - 2^{-2} - |1001.01| \\ c &= 10000 - 0.01 - 1001.01 \\ c &= 10000.0 - 1001.1 \\ c &= 00110.1\end{aligned}$$

$$\text{ar-1} = 110.1_2$$



## Ejercicio 2

**Obtener el complemento a la base disminuida de los siguientes números:**

Línea	Base 2	Base 10
1	10000111	49234
2	11111000	61816
3	10001111	57362
4	11101111	56253
5	11100101	83456
6	11110000	71912
7	11000011	70007
8	11100001	49567

Realizar la siguiente operación: 123 - 98.

Magnitud y signo

$$\begin{array}{r} 123 \\ 11 \\ - 98 \\ \hline 025 \end{array}$$

Complemento a2

$$\begin{array}{r} 123 \\ + 902 \\ \hline 1025 \end{array}$$

Complemento a2-1

$$\begin{array}{r} 123 \\ + 901 \\ \hline 1024 \\ 1 \\ \hline 025 \end{array}$$

Realizar la siguiente operación: 123 - 125.

Magnitud y signo

$$\begin{array}{r} 123 \\ - 125 \\ \hline - 002 \end{array}$$

Complemento a2

$$\begin{array}{r} 123 \\ + 875 \\ \hline 998 \\ - 002 \end{array}$$

Complemento a2-1

$$\begin{array}{r} 123 \\ + 874 \\ \hline 997 \\ + 1 \\ - 002 \end{array}$$

Se tiene un tipo de dato que ocupa 3 bits, solo es posible almacenar  $2^2$  datos diferentes en memoria.

Magnitud y signo

0 1 1  
0 1 0  
0 0 1  
0 0 0

---

1 0 0  
1 0 1  
1 1 0  
1 1 1

Complemento a2

0 1 1  
0 1 0  
0 0 1  
0 0 0

---

1 1 1  
1 1 0  
1 0 1  
1 0 0

Complemento a2-1

0 1 1  
0 1 0  
0 0 1  
0 0 0

---

1 1 1  
1 1 0  
1 0 1  
1 0 0

### Ejemplo 8

Se desea representar en memoria el número entero -11011010111001<sub>2</sub> en complemento ar.

Para representar el complemento del número es necesario expresarlo con el número máximo de elementos a almacenar en memoria, es decir, para un tipo de datos de que ocupa 32 bits en memoria, el número quedaría expresado como sigue:

**-00000000000000000000110111010111001<sub>2</sub>**

## Ejemplo 8

Una vez que se posee el número completo se obtiene el complemento ar:

$$\begin{aligned} \text{(a) } -00000000000000000110111010111001_2 &= \\ &= 1111111111111111001000101000111_2 \text{ (ar)} \end{aligned}$$

## Ejemplo 8

La representación en memoria del número entero -  
 $0000000000000000110111010111001_2$  en complemento ar es:

$11111111111111111001000101000111_2$

-

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	0	0	0	1	0	1	0	0	0	1	1	1

## Representación de un tipo de dato real

En la práctica se está constantemente trabajando con números reales, sin embargo, estos números no siempre se pueden representar de manera exacta debido a que el número de dígitos está limitado por el tamaño de palabra de cada máquina.

Por lo tanto, los números necesariamente son redondeados o truncados, y, con ello, se provoca un error.



Para almacenar en memoria un número real se puede utilizar la notación de punto flotante normalizado o la notación de punto flotante no normalizada.

## Notación científica no normalizada

Un número  $A$  puede ser expresado en notación científica no normalizada, es decir, expresar el número como una potencia de 10:

$$A = C \times 10^n$$

Donde  $C \geq 1$  y  $n$  es un entero positivo, negativo o cero ( $n \in \mathbb{Z}$ ).

Por ejemplo, si se desea expresar el número 836.238 en la forma de punto flotante no normalizada el número quedaría:  $836238 \times 10^{-3}$ .

## Notación científica normalizada

Un número  $A$  puede ser expresado en notación científica normalizada, es decir, expresar el número como una potencia de 10:

$$A = C \times 10^n$$

Donde  $C < 1$  y  $n$  es un entero positivo, negativo o cero ( $n \in \mathbb{Z}$ ).

Por ejemplo, si se desea expresar el número 836.238 en la forma de punto flotante normalizada el número quedaría:  $0.836238 \times 10^3$ .

Los número binarios también pueden ser representados en notación científica normalizada (notación de punto flotante normalizada) de la siguiente manera:

$$A = M \times 2^n$$

donde  $n$  es un entero positivo, negativo o cero (expresado en binario) y  $M$  es la mantisa (dígitos significativos del número), que debe ser menor a 1.

### Ejemplo 9

Expresar los siguientes números en su forma científica normalizada.

$$A = M \times 2^n$$

$$\begin{array}{c} 11111.01_2 \\ 11111.01_2 = 0.1111101_2 \times 2^{101} \end{array}$$

$$\begin{array}{c} -0.00000011101101_2 \\ -0.00000011101101_2 = -0.11101101_2 \times 2^{-110} \end{array}$$

Un número flotante ocupa 32 bits (4 bytes) en la memoria y se distribuyen de la siguiente manera:

1 bit para el signo de la mantisa

1 bit para el signo del exponente

7 bits para el exponente entero (en binario)

23 bits para la mantisa (en binario)

$\pm$	$\pm$	$e_6$	$e_5$	$e_4$	$e_3$	$e_2$	$e_1$	$e_0$	$m_0$	$m_1$	$m_2$	$m_3$	$m_4$	$m_5$	$m_6$
$m_7$	$m_8$	$m_9$	$m_{10}$	$m_{11}$	$m_{12}$	$m_{13}$	$m_{14}$	$m_{15}$	$m_{16}$	$m_{17}$	$m_{18}$	$m_{19}$	$m_{20}$	$m_{21}$	$m_{22}$

**NOTA:** Como la mantisa siempre empieza con uno, no hay necesidad de almacenar este dígito.

## Ejemplo 10

Convertir el número  $31.25_{10}$  a sistema binario y dibujar su representación en memoria.

$$31.25_{10} = 0.3125_{10} \times 10^2$$

$$0.3125_{10} \times 10^2 = 0.3125_{10} \times 100_{10}$$

## Ejemplo 10

0.3125	* 2
0.6250	0
0.2500	1
0.5000	0
0.0000	1



100	% 2
50	0
25	0
12	1
6	0
3	0
1	1
0	1



$$31.25_{10} = 0.0101_2 \times 1100100_2 = 11111.01_2$$

$$11111.01_2 = 0.1111101_2 \times 2^{101}$$

<u>0</u>	<u>0</u>	0	0	0	0	1	0	1	1	1	1	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



## Ejemplo 11

Realizar la representación del número -0.000721619 en memoria en una palabra de 32 bits.

Al convertir el número a binario se obtiene:

$$-0.000721619_{10} = -0.000000000010101101001010110000100000000101_2$$

Se convierte el número a su forma científica normalizada:

$$-0.10101101001010110000100000000101_2 \times 2^{-1010}$$

## Ejemplo 11

Como el exponente es negativo, se pasa a complemento ar (a2) a 8 cifras ya que el signo es parte del número (Se puede aplicar complemento ar-1 y, al resultado, sumarle 1):

$$-00001010_2 = 11110101_2 + 1_2 = 11110110_2$$

Por lo tanto, el número a almacenar en memoria es:

$$-0.10101101001010110000100000000101_2 \times 2^{11110110}$$

## Ejemplo 11

Por lo tanto, la representación en memoria del número real -  $0.10101101001010110000100000000101_2 \times 2^{11110110}$  es:

<u>1</u>	<u>1</u>	1	1	1	0	1	1	0	0	1	0	1	1	0	1
0	0	1	0	1	0	1	1	0	0	0	0	1	0	0	0

### Ejercicio 3

Se quiere almacenar el resultado de multiplicar  $35.34 \times 7.23$  en una computadora que maneja palabras de 16 bits.

Dibujar su representación en memoria. Considerar 2 bits para los signos, 5 bits para el exponente y 9 bits para la mantisa.

El lenguaje de programación C maneja diferentes tipos de datos enteros y de punto flotante, además de caracteres. Los tipos de datos básicos son:

- Caracteres: codificación definida por la máquina.
- Enteros: números sin punto decimal.
- Reales: números fraccionarios de precisión normal y de doble precisión.

## Variables de tipo carácter en lenguaje C

Tipo	Bits	Valor Mínimo	Valor Máximo
signed char	8	-128	127
unsigned char	8	0	255

Si se omite el clasificador, por defecto se considera “signed”.

## Variables de tipo entero en lenguaje C

Tipo	Bits	Valor Mínimo	Valor Máximo
signed short	16	-32 768	32 767
unsigned short	16	0	65 535
signed int	32	-2,147,483,648	2 147 483 647
unsigned int	32	0	4 294 967 295
signed long	32	-2 147 483 648	2 147 483 647
unsigned long	32	0	4 294 967 295
enum	16	-32 768	32 767

Si se omite el clasificador, por defecto se considera “signed”.

## Variables de tipo punto flotante en lenguaje C

Tipo	Bits	Valor Mínimo	Valor Máximo
float	32	$3.4E^{-38}$	$3.4E^{38}$
double	64	$1.7E^{-308}$	$1.7E^{308}$
long double	80	$3.4E^{-4932}$	$3.4E^{4932}$

**Las variables de tipo flotante siempre poseen signo.**



**Cada tipo de dato posee un especificador para el lenguaje:**

Tipo de dato	Especificador de formato
Entero	%d, %i, %o, %x
Flotante	%f, %lf, %e, %g
Carácter	%c, %d, %i, %o, %x
Cadena de caracteres	%s

## Secuencias de escape

Las secuencias de escape están constituidas por dos caracteres aunque representan uno solo. A continuación se listan:

<code>\a</code>	carácter de alarma
<code>\b</code>	retroceso
<code>\f</code>	avance de hoja
<code>\n</code>	salto de línea
<code>\r</code>	regreso de carro
<code>\t</code>	tabulador horizontal
<code>\v</code>	tabulador vertical
<code>'\0'</code>	carácter nulo

### 1.4.2 Ciclos de control

Las estructuras de control de flujo especifican el orden en que se realiza el procesamiento de datos. Existen dos tipos básicos de estructuras de control: las estructuras de selección y las estructuras de repetición.

## Estructuras de selección

Las estructuras de selección permiten realizar una u otra acción con base en una condición lógica. Las opciones a realizar son mutuamente excluyentes.

## Estructura de selección if

La estructura de selección más conocida es la estructura if. Su sintaxis es la siguiente:

```
if (condición){  
    /* Código a ejecutar si la condición  
    valuada es verdadera */  
}
```

## Ejemplo 12

```
#include <stdio.h>

int main () {
    int a;
    printf ("Ingresar número\n");
    scanf ("%d", &a);
    if (a < 0 ) {
        a = a*-1;
    }
    printf ("El valor absoluto de a es: %d\n", a);
    return 13;
}
```

## Estructura de selección if-else

La estructura de selección if completa involucra una segunda parte conocida como else. Su sintaxis es la siguiente:

```
if (condición){  
    /* Código a ejecutar si la condición  
    valuada es verdadera */  
} else {  
    /* Código a ejecutar si la condición  
    valuada es falsa */  
}
```

### Ejemplo 13

```
#include <stdio.h>

int main () {
    int a;
    printf ("Ingresar número\n");
    scanf ("%d", &a);
    if (a%2 == 0 ) {
        printf ("El número %d es par.\n", a);
    } else {
        printf ("El número %d es impar.\n", a);
    }
    return 48;
}
```



Pueden existir tantas estructuras if-else anidadas como se requieran:

```
if (condición) {  
    if (condición)  
        if (condición)  
            // Instrucciones  
        else  
            // Instrucciones  
    else  
        // Instrucciones  
} else {  
    if (condición)  
        // Instrucciones  
    else  
        if (condición)  
            // Instrucciones  
        else  
            // Instrucciones  
}
```

## Estructura de selección switch-case

La estructura switch-case permite valuar una variable entre varias opciones. Su sintaxis es la siguiente:

```
switch (opcion){  
    case valor1:  
        /* Código a ejecutar*/  
        break;  
    case valor2:  
        /* Código a ejecutar*/  
        break;  
    ...  
    case valorN:  
        /* Código a ejecutar*/  
        break;  
    default:  
        /* Código a ejecutar*/  
}
```

## Ejemplo 14

```
#include <stdio.h>

main () {
    int mes;
    printf ("Proporcione el mes de su cumpleaños: \n");
    scanf ("%d", &mes);
    switch (mes) {
        case 1:
            printf("Usted nació en Enero. \n");
            break;
        case 2:
            printf("Usted nació en Febrero. \n");
            break;
        case 3:
            printf("Usted nació en Marzo. \n");
            break;
```

## Ejemplo 14

```
case 4:
    printf("Usted nació en Abril. \n");
    break;
case 5:
    printf("Usted nació en Mayo. \n");
    break;
case 6:
    printf("Usted nació en Junio. \n");
    break;
case 7:
    printf("Usted nació en Julio. \n");
    break;
case 8:
    printf("Usted nació en Agosto. \n");
    break;
case 9:
    printf("Usted nació en Septiembre. \n");
    break;
```

## Ejemplo 14

```
        case 10:
            printf("Usted nació en Octubre. \n");
            break;
        case 11:
            printf("Usted nació en Noviembre. \n");
            break;
        case 12:
            printf("Usted nació en Diciembre. \n");
            break;
        default:
            printf ("OpciOn invAlida. \n");
    }
}
```

## Estructura de selección condicional

La expresión condicional (también llamado operador ternario) permite realizar una comparación rápida. Su sintaxis es la siguiente:

Condición ? SiSeCumple : SiNoSeCumple

Se valúa una condición, si la condición se cumple (verdadera) se ejecuta la instrucción después del símbolo ?; si la condición no se cumple (falso), se ejecuta la instrucción después de :.

## Ejemplo 15

```
#include <stdio.h>

int main () {
    int a, b;
    printf ("\nPrograma que calcula el error matemático.");
    printf ("\nentre dos números | a - b |.");
    printf ("\nProporcione a:\n");
    scanf ("%d", &a);
    printf ("\nProporcione b:\n");
    scanf ("%d", &b);
    int c = a < b ? b - a : a - b;
    printf ("\nE = %d\n", c);
    return 88;
}
```

## Ejercicio 4

Realizar un programa que permita convertir un número entero en base 10 a base 8. Si el usuario no ingresa un número el programa debe marcar un error.

La firma (sintaxis) de la función scanf es la siguiente:

```
int scanf(const char *formato, ...);
```



## Estructuras de repetición

Las estructuras de repetición son las llamadas estructuras cíclicas o de bucles. Permiten ejecutar un conjunto de instrucciones mientras se cumpla una condición. Existen tres estructuras de repetición: while, do-while y for.

## Estructura de repetición while

La estructura del ciclo while valida la condición dada y si es correcta ejecuta el código que se encuentra dentro de las llaves, si no es correcta sigue el flujo normal del programa. Su sintaxis es la siguiente:

```
while (condición){           // Inicio del ciclo while
    /* Código a ejecutar */
}
```

// fin del ciclo while

## Ejemplo 16

```
#include <stdio.h>

int main () {
    int a;
    a = 0;
    while (a < 10 ) {
        printf ("\v\t%d\n",a++);
        sleep(1);
    }
    return 58;
}
```

## Estructura de repetición do-while

La estructura cíclica do-while ejecuta, por lo menos, el bloque de código que se encuentra dentro de las llaves y después valida la condición, si ésta se cumple vuelve a ejecutar el bloque, de lo contrario sigue con el flujo normal del programa. Su sintaxis es la siguiente:

```
do {  
    /*  
        Código a ejecutar  
    */  
} while (condición);
```

## Ejemplo 17

```
#include <stdio.h>
#define p printf
#define s scanf

main () {
    float a, b;
    a=5.5;
    b=8.7;
    char salir = 'n';
    do {
        system ("clear");
        p ("Ingrese a y b separados por espacios\n");
        s ("%f %f",&a,&b);
        float c = a<b ? b-a : a-b;
        p ("E = %f",c);
        p ("\nDesea calcular otro error (s/n)\n");
        setbuf(stdin,NULL);
        s ("%c",&salir);
        getchar();
    } while (salir == 's');
}
```

## Estructura de repetición for

La estructura for es una estructura repetitiva que consta de tres partes: un valor inicial, una condición y una expresión final. Su sintaxis es la siguiente:

```
for (valorInicial; condición; (in/de)-cremento) {  
    /*Código a ejecutar*/  
}
```

## Ejemplo 18

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

main () {
    int grado, x, fx, coef, cont;
    printf ("\nProporcione el grado del polinomio: ");
    scanf ("%d", &grado);
    printf ("\nProporcione la x a evaluar: ");
    scanf ("%d", &x);
    fx = 0;
    for (cont = 0 ; cont <= grado ; cont++){
        printf ("\nCoeficiente de x%d: ", cont);
        scanf ("%d", &coef);
        fx = fx + coef*pow(x,cont);
    }
    printf ("\nf(x) = %d\n",fx);
}
```

## Ejercicio 5

A partir de un texto ingresado desde la entrada estándar, obtener el número total de vocales, consonantes y palabras.

La sintaxis de la función strtok es:

```
char *strtok(char *s1, const char *s2);
```

Si se utilizar la función pasando como argumento el parámetro NULL lo que se hace es recorrer el apuntador a la siguiente coincidencia: strtok(NULL, s2).



## Enumeración

Una enumeración es un tipo de dato constante. Su sintaxis se muestra a continuación:

```
enum nombre {elem1, elem2, ... elemn};
```

A cada elemento de la enumeración se le asigna un valor constante.

## Ejemplo 19

```
#include<stdio.h>

main() {
    enum dias {lunes, martes, miercoles, jueves, viernes, sabado, domingo};
    enum dias varEnum;
    varEnum = viernes;
    switch(varEnum) {
        case lunes:
        case martes:
            printf("¡Inicio de semana!\n");
            break;
        case miercoles:
            printf("Ombligo de semana.\n");
            break;
        case jueves:
        case viernes:
        case sabado:
            printf("¡Inicia el fin de semana!\n");
            break;
        case domingo:
            printf("Se acabó la semana.");
            break;
    }
}
```

## Ejercicio 6

**Crear un programa que permita manejar tipos de datos booleanos personalizados, es decir, verdadero o falso.**

## Funciones

En C es posible tener dentro de un archivo fuente varias funciones, esto con el fin de dividir las tareas y que sea más fácil la depuración o mejora del código.

En lenguaje C la función principal se llama main. El compilador solo ejecuta las instrucciones que se encuentran dentro de la función main, empero, ésta puede a su vez llamar o ejecutar otras funciones.

## Ejemplo 20

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

void imp_rev(char s[]) {
    int t;
    for( t=strlen(s)-1; t>=0; t--)
        printf("%c",s[t]);
    printf("\n");
}

int main() {
    char nombre[]="Facultad";
    imp_rev(nombre);
    return 54;
}
```

## Ejercicio 7

**Realizar una función que permita convertir un número de base 10 a base 2. La función debe ser externa a la función principal y debe ser recursiva (en caso de ocupar ciclos de repetición).**



## 1.5 Manejo de memoria, acceso, asignación dinámica, apuntadores, arreglos.

## 1.5 Manejo de memoria, acceso, asignación dinámica, apuntadores, arreglos.

**C es un lenguaje de alto nivel porque permite programar a bajo nivel. Lenguaje C permite trabajar directamente con la memoria principal.**

**También es posible manejar la memoria de manera estática o de forma dinámica.**



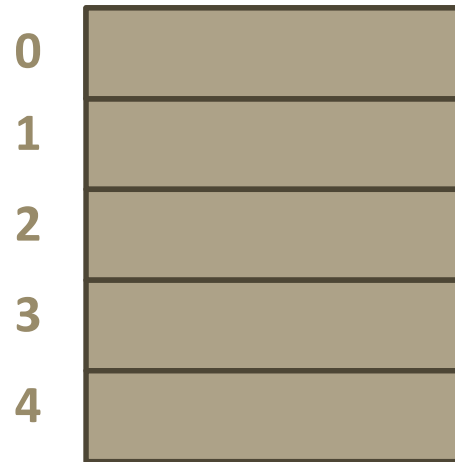
## Arreglos

Se denomina arreglo a un conjunto de datos del mismo tipo que son contiguos y ordenados. Un arreglo posee un tamaño fijo definido al momento de crearse.

A cada localidad del arreglo (elemento) se le asocia un número. Es posible crear arreglos unidimensionales y multidimensionales.

## Arreglos unidimensionales

La primera localidad de un arreglo corresponde al índice 0 y la última corresponde al índice  $n-1$ , donde  $n$  es el tamaño con el que se creó el arreglo.



La sintaxis para definir un arreglo en lenguaje C es la siguiente:

`tipoDeDato nombre [ tamaño ]`

Donde nombre se refiere al identificador del arreglo, tamaño es un número entero y define el número máximo de elementos que puede contener el arreglo. Un arreglo puede ser de cualquier tipo de dato: entero, real, carácter o estructura.

## Ejemplo 21

```
#include <stdio.h>

#define TAMANO 5

int main () {
    int lista[TAMANO] = {10, 8, 5, 8, 7};
    int c = 0;
    while (c < 5 ) {
        printf ("\nCalif alumno %d = %d",c+1,lista[c]);
        c += 1;
    }
    printf ("\n");
    return 0;
}
```

## Ejemplo 22

```
#include<stdio.h>

main(){
    char palabra[20];
    int i=0;
    printf("Introduce una palabra: ");
    scanf("%s", palabra);
    printf("La palabra ingresada es: %s\n", palabra);
    for (i = 0 ; i < palabra[i] ; i++){
        printf("%c\n", palabra[i]);
    }
}
```

## Arreglos multidimensionales

En C es posible crear arreglos de varias dimensiones. La sintaxis para declarar arreglos multidimensionales es:

```
tipoDato nombre [ tamaño ][ tamaño ]...[tamaño];
```

## Ejemplo 23

```
#include<stdio.h>

int main(){
    int matriz[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
    int i, j;
    printf("Matriz\n");
    for (i=0 ; i<3 ; i++){
        for (j=0 ; j<3 ; j++){
            if ((j%3)==0)
                printf("\n");
            printf("\t%d ",matriz[i][j]);
        }
        printf("\n");
    }
    return 42;
}
```

## Ejercicio 8

Crear un programa que permita el inicio de sesión con nombre de usuario y contraseña. La validación debe regresar un valor booleano, es decir, hay que utilizar el programa enumerador de booleanos.



## Apuntador

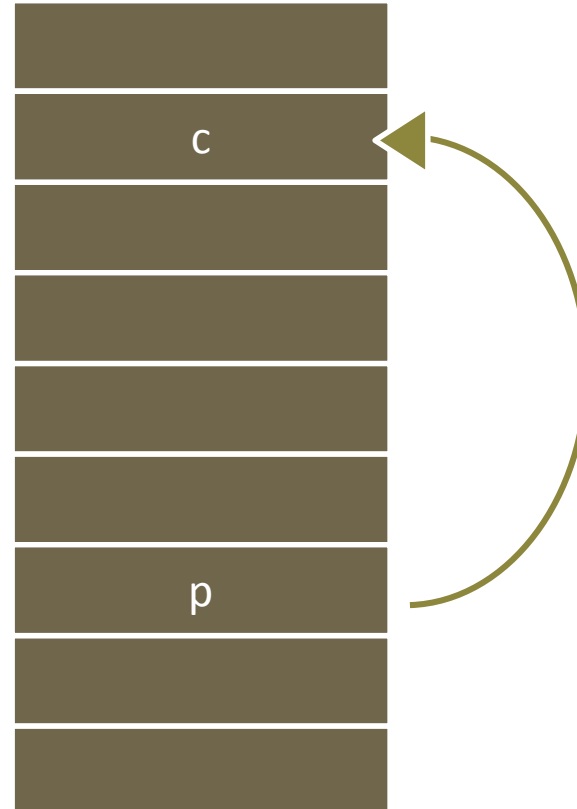
Un apuntador es una variable que contiene la dirección de una variable.

Los apuntadores se utilizan debido a que algunas veces son la única forma de expresar una operación, y debido a que, generalmente, llevan un código compacto y eficiente.

Los apuntadores también pueden emplearse para obtener claridad y simplicidad.

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

```
int main () {  
    char c;  
    char *ap;  
    ap = &c;  
    return 5;  
}
```



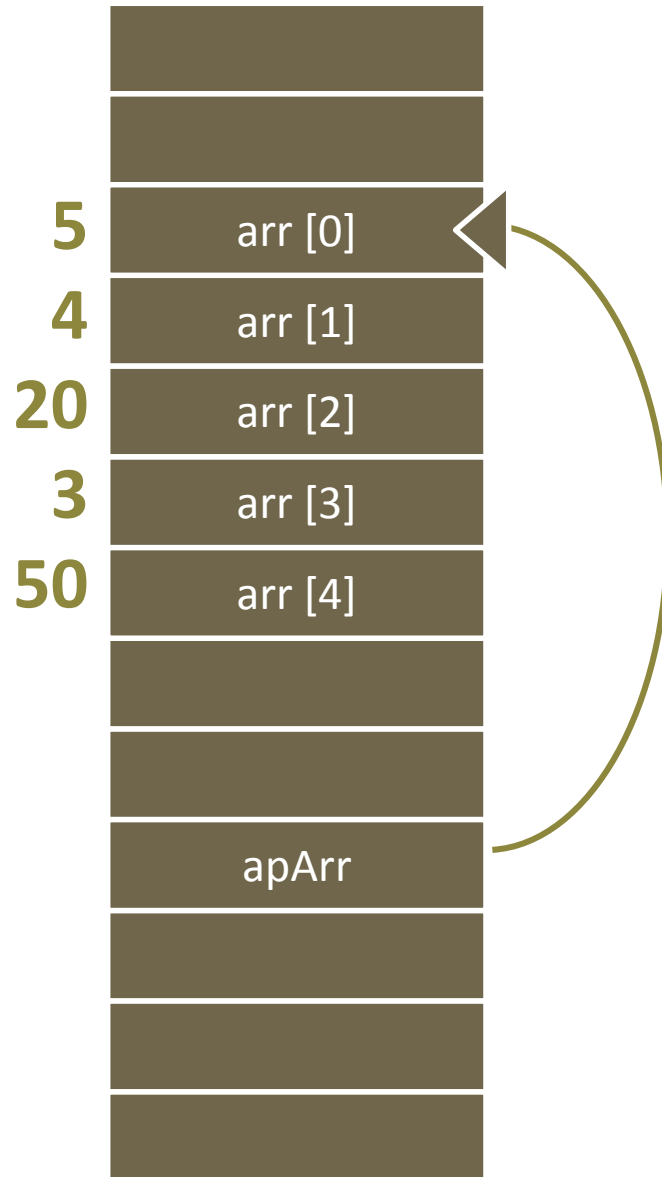
## Ejemplo 24

```
#include<stdio.h>

int main () {
    int a = 5, b = 10, c[5]={15,14,13,12,11};
    int *apEnt;                // apuntador a entero
    apEnt = &a;                // apEnt -> a
    b = *apEnt;                // b = 5
    b = *apEnt +1;             // b = 6
    *apEnt = 0;                // a = 0
    apEnt = &c[4];             // apEnt -> c[4]
    return 34;
}
```

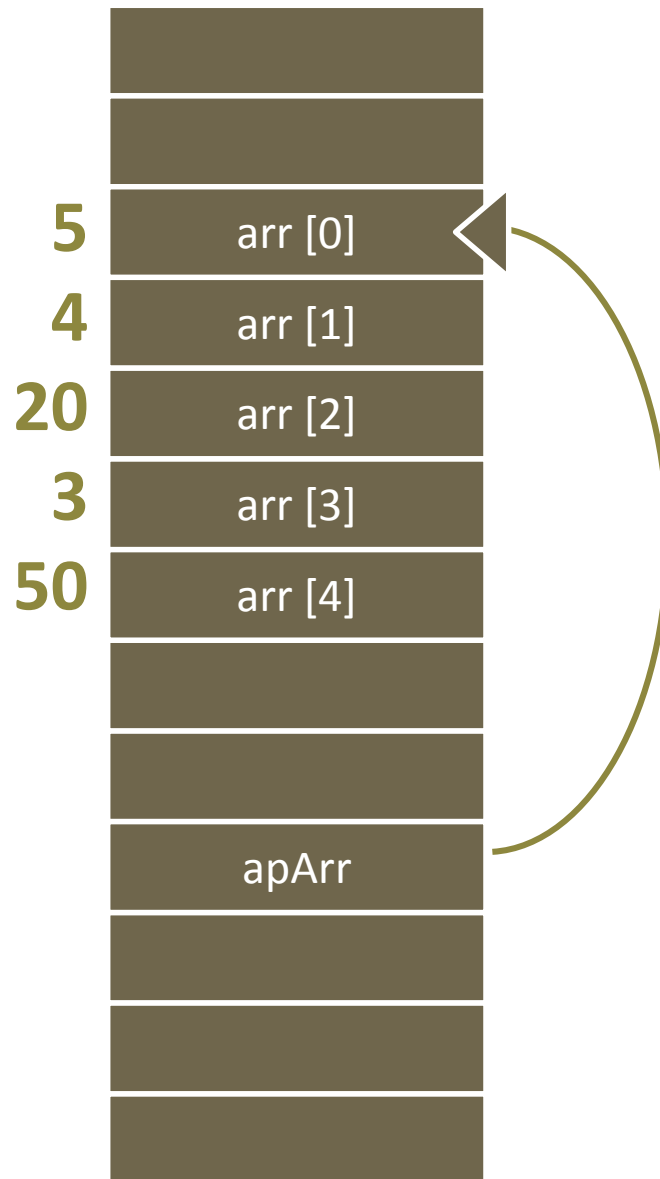
```
#include <stdio.h>
```

```
int main () {  
    int arr [5];  
    int *apArr;  
    apArr = &arr[0];  
    return 84;  
}
```



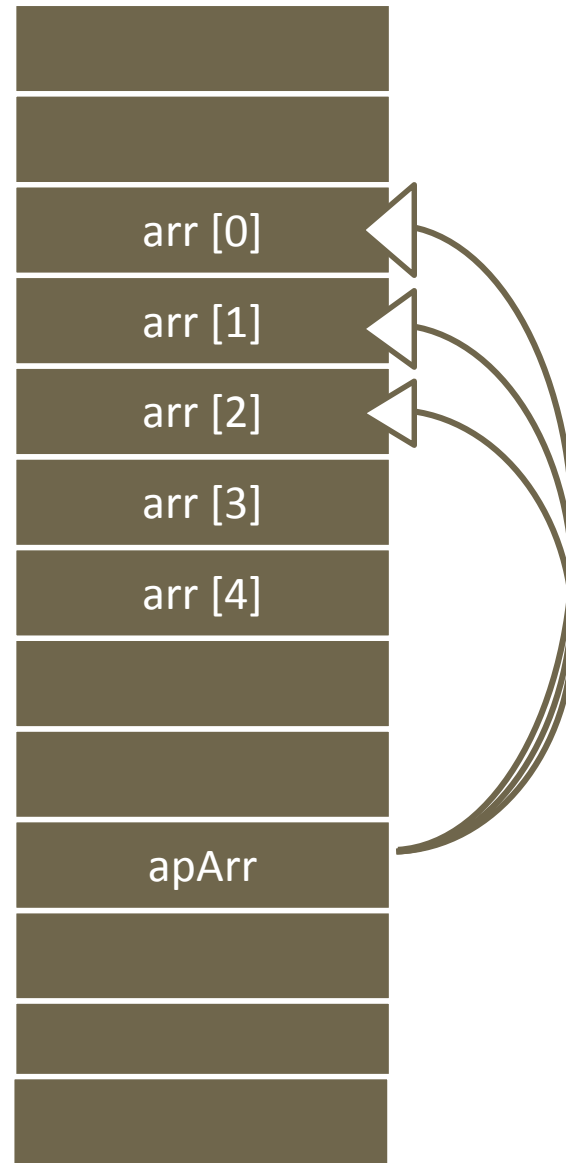
```
#include <stdio.h>
```

```
int main () {  
    int arr [5];  
    int *apArr;  
    apArr = arr;  
    int x = *apArr;  
    return 1;  
}
```



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

```
int main () {  
    int arr [5];  
    int *apArr;  
    apArr = arr;  
    int x = *apArr;  
    *(apArr+1);  
    *(apArr+2);  
    return 8;  
}
```



## Aritmética de direcciones:

Si `apArr` es un apuntador a algún elemento de un arreglo, entonces:

`apArr++`: Incrementa `apArr` para apuntar al siguiente elemento.

`apArr+=i`: Incrementa `apArr` para apuntar al `i` elementos adelante.

## Ejemplo 25

```
#include <stdio.h>

int main(){
    int matriz[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
    int i, *ap;
    ap=matriz;
    printf("Matriz\n");
    for (i=0 ; i<9 ; i++){
        printf("\t%d",*(ap+i));
    }
    printf("\n");
}
```



## Ejemplo 26

```
#include <stdio.h>

void ver(void *, int);

int main() {
    char a='b';
    int x=3;
    double y=4.5;
    char *cad="hola";
    ver(&a, 0);
    ver(&x, 2);
    ver(&y, 1);
    ver(cad, 3);
    getchar();
    return 0;
}
```

## Ejemplo 26

```
void ver( void *p, int d) {  
    switch(d) {  
        case 0:  
            printf("%c\n",*(char *)p);  
            break;  
        case 1:  
            printf("%d\n",*(double *)p);  
            break;  
        case 2:  
            printf("%ld\n",*(int *)p);  
            break;  
        case 3:  
            printf("%s\n",(char *)p);  
            break;  
        default:  
            printf("Error ");  
    }  
}
```

## Ejemplo 27

```
#include <stdio.h>

char *nombre_mes(int);

int main(void){
    char *mes = NULL;
    int n;
    puts("Introduzca el numero del mes deseado");
    if (scanf("%d", &n) != 1){
        printf("Caracter invAlido\n");
        return;
    }
    mes = nombre_mes(n);
    printf("El mes seleccionado es:\n%s\n", mes);
    return 4;
}
```

## Ejemplo 27

**/\* Método que genera un arreglo de caracteres (apuntador de  
apuntadores \*/**

```
char *nombre_mes(int numero){  
    static char *mes[] = {  
        "Mes invAlido",  
        "Enero", "Febrero", "Marzo",  
        "Abril", "Mayo", "Junio",  
        "Julio", "Agosto", "Septiembre",  
        "Octubre", "Noviembre", "Diciembre"  
    };  
  
    return (numero < 1 || numero > 12) ? mes[0] : mes[numero];  
}
```

## Apuntador a función

Un apuntador a función es una función que apunta a otra función. La firma para declarar un apuntador a función es la siguiente:

```
tipoDatoRetorno (*apFunc)([parámetros]);
```

Para que una función pueda apuntar a la dirección de memoria de otra función es necesario tener la función estática declarada, es decir:

```
int funcion(int);
```

La asignación de una función apuntador se realiza de la siguiente manera:

```
apFunc = &funcion;
```

## Ejemplo 28

```
#include <stdio.h>

int suma(int,int);

int main(){
    int res;
    int (*apSuma)(int,int)=&suma;
    res = apSuma(6,4);
    printf("%d\n",res);
}

int suma(int a,int b) {
    return a+b;
}
```

## Asignación dinámica de memoria

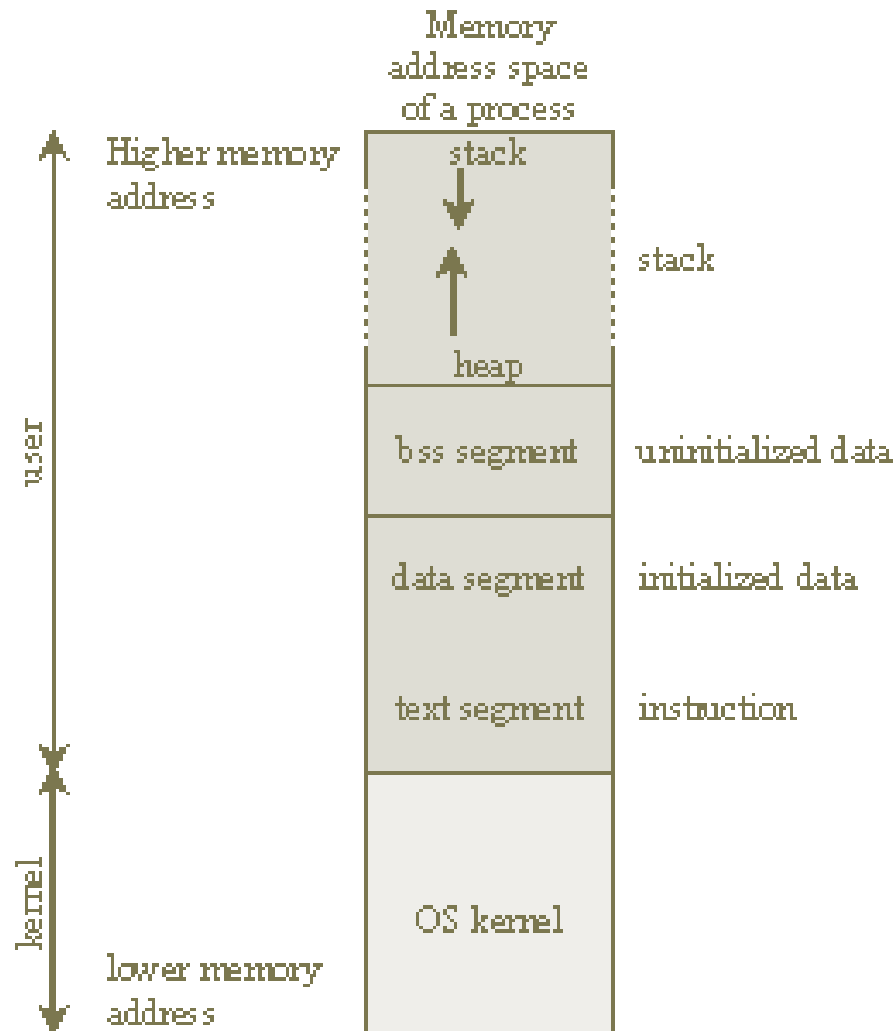
La memoria dinámica es aquella que se reserva en tiempo de ejecución. Su tamaño puede variar durante la ejecución del programa.

El uso de memoria dinámica es necesario cuando a priori no se conoce el número de datos y/o elementos a manejar.



La memoria reservada de forma dinámica suele estar alojada en el heap o almacenamiento libre, y la memoria estática en el stack o pila.

La pila generalmente es una zona muy limitada. El heap, en cambio, en principio podría estar limitado por la cantidad de memoria disponible durante la ejecución del programa y el máximo de memoria que el sistema operativo permita direccionar a un proceso.



**El segmento de texto (también llamado segmento de código) es donde el código del programa compilado reside, es decir, el código ejecutable o código binario del programa.**

**El segmento de datos contiene las variables globales, estáticas y constantes.**

**El segmento bss contiene todos los datos (o variables) sin inicializar**

El heap contiene todos los datos reservados en tiempo de ejecución mediante las funciones calloc o malloc.

El stack es utilizado para almacenar variables globales y para pasar argumentos hacia función, así como para devolver la dirección de la siguiente instrucción a ejecutar después de que la función haya terminado de ejecutarse.

La memoria estática tiene una duración fija, que se reserva y libera de forma automática. La memoria dinámica se reserva de forma explícita y continúa existiendo hasta que se libera de forma explícita.

El almacenamiento dinámico puede afectar el rendimiento debido a que se llevan a cabo arduas tareas en tiempo de ejecución: buscar un bloque de memoria libre y almacenar la posición y tamaño de la memoria asignada.

La biblioteca estándar de C proporciona las funciones malloc, calloc, realloc y free para el manejo de memoria dinámica. Estas funciones están definidas en librería stdlib.h.

## Función malloc

La función malloc reserva un bloque de memoria de manera dinámica y devuelve un apuntador a void. Su firma se especifica a continuación:

```
void *malloc(size_t size);
```

El parámetro size especifica el número de bytes a reservar. En caso de que no sea posible realizar la asignación de memoria, devuelve el valor nulo (NULL).

## Ejemplo 29

```
#include <stdio.h>
#include <stdlib.h>
int main (){
    int *arreglo, num, cont;
    printf("¿Cuantos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)malloc (num * sizeof(int));
    if (arreglo!=NULL) {
        for (cont=0 ; cont<num ; cont++){
            printf("Elemento %d del arreglo.\n",cont+1);
            scanf("%d",arreglo+cont);
        }
        printf("Vector leído:\n\v\t");
        for (cont=0 ; cont<num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
    }
}
```



## Función calloc

La función calloc funciona de modo similar a la función malloc, pero además de reservar memoria, inicializa con 0 la memoria reservada, por tanto, es utilizada comúnmente para arreglos unidimensionales y multidimensionales. Su sintaxis es la siguiente:

```
void *calloc (size_t nelem, size_t size);
```

El parámetro nelem indica el número de elementos a reservar, y size el tamaño de cada elemento.

### Ejemplo 30

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

int main (){
    int *arreglo, num, cont;
    printf("¿Cuantos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)calloc (num, sizeof(int));
    if (arreglo!=NULL) {
        printf("Espacio reservado:\n\v\t[");
        for (cont=0 ; cont<num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
    }
}
```

## Ejemplo 30-2

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

int main (){
    int *arreglo, num, cont;
    printf("¿Cuantos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)malloc (num * sizeof(int));
    if (arreglo!=NULL) {
        printf("Espacio reservado:\n\v\t[");
        for (cont=0 ; cont<num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
    }
}
```

## Función realloc

La función realloc permite redimensionar el espacio asignado previamente y de forma dinámica. Su sintaxis es la siguiente:

```
void *realloc (void *ptr, size_t size);
```

Donde ptr es el apuntador a redimensionar y size el nuevo tamaño, en bytes, que tendrá.

Si el apuntador tiene el valor nulo, la función actúa como malloc. Si la reasignación no se pudo realizar, devuelve un apuntador a nulo, dejando intacto el apuntador que se pasa como parámetro.

### Ejemplo 31

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

int main (){
    int *arreglo, num, cont;
    printf("¿Cuantos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)malloc (num * sizeof(int));
    if (arreglo!=NULL) {
        for (cont=0 ; cont < num ; cont++){
            printf("Inserte el elemento %d del arreglo.\n",cont+1);
            scanf("%d",arreglo+cont);
        }
        printf("Vector insertado:\n\v\t[");
        for (cont=0 ; cont < num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
    }
```

### Ejemplo 31

```
printf("Aumentando el tamaño del buffer al doble.\n");
num *= 2;
int *arreglo2 = (int *)realloc (arreglo,num*sizeof(int));
if (arreglo2 != NULL) {
    for (; cont < num ; cont++){
        printf("Inserte el elemento %d del arreglo.\n",cont+1);
        scanf("%d",arreglo+cont);
    }
    printf("Vector insertado:\n\v\t[");
    for (cont=0 ; cont < num ; cont++){
        printf("\t%d",*(arreglo+cont));
    }
    printf("\t]\n");
}
}
```

## Función free

La función `free` sirve para liberar memoria que se asignó dinámicamente. Si el apuntador es nulo, `free` no hace nada. Su sintaxis es la siguiente:

```
void free(void *ptr);
```

El parámetro `ptr` es el apuntador a la memoria que se desea liberar.

### Ejemplo 32

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

int main (){
    int *arreglo, num, cont;
    printf("¿Cuantos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)malloc (num * sizeof(int));
    if (arreglo!=NULL) {
        for (cont=0 ; cont<num ; cont++){
            printf("Inserte el elemento %d del arreglo.\n",cont+1);
            scanf("%d",arreglo+cont);
        }
        printf("Vector insertado:\n\v\t[");
        for (cont=0 ; cont<num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
        printf("Liberando espacio reservado dinamicamente.\n");
        free(arreglo);
    }
}
```



## Ejemplo 32-2

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

int main (){
    int *arreglo, num, cont;
    printf("¿Cuantos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)malloc (num * sizeof(int));
    if (arreglo!=NULL) {
        for (cont=0 ; cont < num ; cont++){
            printf("Inserte el elemento %d del arreglo.\n",cont+1);
            scanf("%d",arreglo+cont);
        }
        printf("Vector insertado:\n\v\t[");
        for (cont=0 ; cont < num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
    }
```

## Ejemplo 32-2

```
printf("Aumentando el tamaño del buffer al doble.\n");
num *= 2;
int *arreglo2 = (int *)realloc (arreglo,num*sizeof(int));
if (arreglo2 != NULL) {
    for (; cont < num ; cont++){
        printf("Inserte el elemento %d del arreglo.\n",cont+1);
        scanf("%d",arreglo+cont);
    }
    printf("Vector insertado:\n\v\t[");
    for (cont=0 ; cont < num ; cont++){
        printf("\t%d",*(arreglo+cont));
    }
    printf("\t]\n");
    free (arreglo2);
}
free (arreglo);
}
```

Tratar de utilizar un apuntador cuyo bloque de memoria ha sido liberado con free puede ser sumamente peligroso. El comportamiento del programa queda indefinido: puede terminar de forma inesperada, sobrescribir otros datos y provocar problemas de seguridad.

Para evitar problemas con apuntadores a memoria dinámica ya liberados, se recomienda que se establezca su valor a NULL.

### Ejemplo 33

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

int main (){
    int *arreglo, num, cont;
    printf("¿Cuántos elementos tiene el arreglo?\n");
    scanf("%d",&num);
    arreglo = (int *)malloc (num * sizeof(int));
    if (arreglo!=NULL) {
        for (cont=0 ; cont < num ; cont++){
            printf("Inserte el elemento %d del arreglo.\n",cont+1);
            scanf("%d",arreglo+cont);
        }
        printf("Vector insertado:\n\v\t");
        for (cont=0 ; cont < num ; cont++){
            printf("\t%d",*(arreglo+cont));
        }
        printf("\t]\n");
        free(arreglo);
        arreglo=NULL;
    }
```

### Ejemplo 33

```
printf("Aumentando el tamaño del buffer al doble.\n");
num *= 2;
int *arreglo2 = (int *)realloc (arreglo,num*sizeof(int));
if (arreglo2 != NULL) {
    for (; cont < num ; cont++){
        printf("Inserte el elemento %d del arreglo.\n",cont+1);
        scanf("%d",arreglo+cont);
    }
    printf("Vector insertado:\n\v\t[");
    for (cont=0 ; cont < num ; cont++){
        printf("\t%d",*(arreglo+cont));
    }
    printf("\t]\n");
    free (arreglo2);
}
free (arreglo);
}
```

# 1 Elementos para el estudio de estructuras de datos

**Objetivo:** Comprender los aspectos básicos de la estructura de una computadora digital, que permita obtener un marco de referencia para iniciar el estudio de las estructuras de datos.

- 1.1 Componentes físicos de una computadora.
- 1.2 Elementos internos de la computadora.
- 1.3 Conceptos básicos de programación de bajo nivel.
- 1.4 Conceptos de programación de alto nivel (estructurada).
  - 1.4.1 Representación de tipos de datos.
  - 1.4.2 Ciclos de control.
- 1.5 Manejo de memoria, acceso, asignación dinámica, apuntadores, arreglos.

## Análisis (Kernel de linux)

```
#include <linux/module.h>          /* Needed by all modules */
#include <linux/kernel.h>          /* Needed for KERN_INFO */
#include <linux/jiffies.h>

int init_module(void) {
    int n = 5;

    printk(KERN_INFO "Hello world 1.\n");
    unsigned long j, stamp_1, stamp_half, stamp_n;

    j = jiffies;
    stamp_1  = j + HZ;
    stamp_half = j + HZ/2;
    stamp_n   = j + n * HZ / 1000;

    printk(KERN_INFO "Current time in jiffies = %lu \n",j);
    printk(KERN_INFO "Current time in ms = %d \n",jiffies_to_msecs(j));
    printk(KERN_INFO "1 second in the future = %d \n",jiffies_to_msecs(stamp_1));
    printk(KERN_INFO "half a second= %d \n", jiffies_to_msecs(stamp_half));
    printk(KERN_INFO "n milliseconds = %d \n",jiffies_to_msecs(stamp_n));

    return 0;
}
```

## Pebble

