TEMA 2: Soporte lógico de los ordenadores

Índice:

1.Motivación

Se entiende por soporte lógico el conjunto de programas que dirigen el funcionamiento del ordenador. Es decir, se trata de todo lo que se engloba dentro del término **software**.

Como ya se comentó en el tema anterior, los programas dirigen las operaciones de transformación de datos. Recordemos que el ordenador se comporta como una máquina de propósito general, especialmente destinada a la manipulación de datos, y las órdenes específicas de cada transformación se integran en los programas. Un programa destinado a la manipulación de imágenes, por ejemplo, se compone de una serie de instrucciones para su manipulación.

Sin embargo, antes de entrar a considerar en mayor detalle el soporte lógico, hemos de conocer la forma en que se representan los datos dentro del ordenador. Está claro que el ordenador no transforma elementos de la realidad, sino información sobre dichos elementos reales. Es decir, la información sobre los objetos reales ha de introducirse en el ordenador. La primera parte del tema consiste en ver cómo se realiza este proceso, cómo los datos que manejamos

a nivel conceptual se pueden introducir en el ordenador. Así, veremos la forma en que la información se representa en el interior del ordenador. Obviamente, en este apartado se considerará tanto la forma de introducir datos como la de introducir instrucciones.

2. Representación interna de la información

En este apartado se van a considerar las distintas formas de representar datos e instrucciones en el interior de nuestros ordenadores. Empezaremos considerando la representación de datos, desde los más simples a los más complejos, para terminar estudiando el modo de especificar al ordenador las órdenes relativas a la forma en que se ha de manipular la información, para poder obtener los resultados que nosotros deseamos.

Antes de nada hemos de recordar que el ordenador sólo puede manejar 0's y 1's. ¿Por qué? El ordenador no tiene consciencia y no puede discernir más que entre estados claramente diferenciados: hay corriente, no hay corriente. Esto contrasta con la elevada capacidad humana para la manipulación de símbolos y conceptos. Desde pequeños somos capaces de distinguir entre caracteres de diversa índole (letras, números, iconos, símbolos especiales,…), pudiéndolos combinar para expresarnos y comunicarnos con los demás. Esta capacidad de comprender y distinguir entre símbolos no se da en los ordenadores. De ahí la necesidad de transformar la información al formato que precisa el conjunto de circuitos electrónicos que componen el ordenador.

Si el ordenador no puede distinguir más que entre dos estados, al final, sin más remedio, habrá que expresar todos los datos de modo que puedan representarse mediante dos situaciones distintas (dos símbolos distintos): hay corriente – no hay corriente, hay magnetización – no hay magnetización, etc. Estos dos estados vienen representados por los símbolos 0 y 1.

Cada una de estas unidades elementales de información se denomina ***bit***. Obviamente, un bit ofrece poca capacidad de representación. Por ello, se utilizan grupos de bits. En definitiva, mediante grupos de bits hemos de poder aludir a los distintos datos que queremos manipular. De esta forma, a cada dato o elemento de la realidad le tendré que hacer corresponder una combinación de 0's y 1's. Esto se denomina ***código***.

Los códigos se manejan en muchos ámbitos de la vida real. De hecho, los códigos permiten representar de forma única (y abreviada) a los objetos (e incluso a las personas). Por ejemplo:

Entidades	**Código**
Persona	DNI
Coche	Matrícula
Cuenta corriente	Código de cuenta
2	10
Let it be	1…….0……..01

Por tanto, el uso de 0's y 1's para representar datos no es una elección, sino una necesidad. En cualquier estado en que se encuentre la información (en memoria principal, en disco duro, en disquete, en CD-ROM), se mantiene esta distinción entre dos estados diferentes.

Como sólo podemos utilizar dos símbolos (representado los estados distinguibles), el sistema de codificación así obtenido se denomina ***binario***. Iremos estudiando, poco a poco, cuáles son las normas que rigen este sistema de numeración y de codificación.

Pero debemos comenzar cuestionando la capacidad de representación mediante dígitos binarios. Es decir, supongamos que mi ordenador sólo puede almacenar en memoria agrupamientos de 4 bits. En este caso, ¿cuántos datos diferentes podría manejar? Para poder responder a esta pregunta, conviene hacerse la analogía respecto al sistema decimal de numeración. Supongamos que para codificar matrículas sólo puedo utilizar 4 caracteres (del 0 al 9). Usando estos 4 dígitos, ¿cuántas matrículas puedo generar? O lo que es lo mismo, ¿cuántos coches puedo representar?

En el caso del sistema decimal de numeración la respuesta es clara. Podremos generar matrículas desde la 0000 hasta la 9999. Es decir, podremos obtener 10000 matrículas diferentes. Si observamos, en realidad lo que hemos determinado es el conjunto completo de formas de combinar 4 dígitos decimales. Se ve más fácil si consideramos sólo dos dígitos

0
1
…
8
9
10
11
…
19
20
21
…
28
29
…
…
57
…
98
99

Se observa que con dos dígitos podríamos representar 100 matrículas (coches) diferentes. Esto puede generalizarse mediante la fórmula siguiente:

$$\text{Número de matrículas} = 10^{\text{número dígitos que componen la matrícula}}$$

De esta forma, cuando tenemos dos cifras podemos diferenciar entre 10^2 matrículas. Si tenemos 4 dígitos podremos representar 10^4 matrículas diferentes. Por ello, para conseguir mayor capacidad de representación se utilizan también caracteres en las matrículas. En este caso, el número de símbolos permitidos aumenta, y por tanto, la base de la anterior fórmula.

Igual ocurre con la representación binaria. El número de "objetos" diferentes que se pueden representar con un determinado número de dígitos (***n***) vendrá determinado por 2^n. Es decir, hay que elevar el número de símbolos disponibles en el sistema de numeración (2 en binario (0-1), 10 en decimal (0 … 9)) al número de caracteres que puedo usar. Obviamente, a mayor número de caracteres mayor capacidad de representación.

Veamos un ejemplo con 4 dígitos binarios. Formaremos todas las combinaciones posibles (que serán 2^4, como hemos visto). De esta forma, mediante 4 bits sólo puedo distinguir entre 16 "cosas" diferentes, asignando un código único y específico a cada una de ellas. Obviamente, con 4 bits no puedo ni representar las letras del alfabeto. En el gráfico siguiente se muestran las codificaciones que podría asignar:

0000(a)
0001(b)
0010(c)
0011(d)
0100(e)
0101(f)
0110(g)
0111(h)
1000(i)
1001(j)
1010(k)
1011(l)
1100(m)
1101(n)
1110(ñ)
1111(o)

Si quisiéremos representar todos los caracteres de la a a la z, ¿cuántos caracteres serían necesarios? Hemos de tener en cuenta que se trata de 27 símbolos distintos. Componed un código para cada uno de ellos.

Recordamos aquí las medidas de capacidad de las memorias principales y de almacenamiento masivo:

1 byte: 8 bits
1 Kbyte: 1024 bytes = 2^{10} bytes
1 Mbyte: 1024x1024 bytes = 2^{20} bytes
1 Gbyte: 1024x1024x1024 bytes = 2^{30} bytes
1 Tbyte: 1024x1024x1024x1024 = 2^{40} bytes

y medidas típicas de algunos elementos de almacenamiento son:

RAM: entre 512Mbytes y 2Gbyte
Disquete: 1’44 Mbytes
Disco duro: 80 – 120 -180 Gbyte
CD – ROM: 700 Mbytes
DVD: hasta 8 Gbytes

Como se aprecia, todas las medidas, por conveniencia, vienen expresadas en potencias de , ya que este número es la base del sistema binario de numeración.

2.1 Representación interna de datos

En este apartado se analizan las formas de representar los diferentes tipos de datos en el interior del ordenador.

2.1.1 Codificación de caracteres

Se indica aquí la forma de representar los caracteres de entrada y salida, usados para intercambiar información con el ordenador (serán los caracteres disponibles en cualquier teclado). Estos caracteres se pueden clasificar de la siguiente forma:

a) Caracteres alfabéticos. Son las letras mayúsculas y minúsculas del alfabeto: a, b, c, d, e, f, …, z, A, B, C, D, E, F, …, Z
b) Caracteres numéricos. Las diez cifras decimales: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
c) Caracteres de control. Se trata de caracteres que indican órdenes: salto de línea (ENTER), tecla espaciadora (espacio), escape (ESC), teclas de función, etc.
d) Caracteres especiales: (,), [,], +, /, *, ¿, etc.

Los dos primeros tipos constituyen el conjunto de caracteres ***alfanuméricos***.

Uno de los sistemas de codificación más usados ha sido el ASCII. Para representar todos estos símbolos empleaba 8 bits, aunque en realidad uno se utilizaba para detección de errores. Al emplearse 7 bits para codificar los símbolos, en ASCII se pueden distinguir 2^7=128 caracteres diferentes.

La siguiente tabla muestra la equivalencia entre el valor decimal y el carácter ASCII que representa (el número decimal equivalente a la combinación de 0’s y 1’s que codifica a cada carácter):

Valor decimal	Carácter de control	Significado
0	NULL	Nulo
1	SOH	Comienzo de Cabecera
2	STX	Comienzo de Texto
3	ETX	Fin de texto
4	EOT	Fin de transmisión
5	ENQ	Pregunta
6	ACK	Comfirmación Positiva
7	BEL	Pitido
8	BS	Espacio atrás
9	HT	Tabulador Horizontal
10	LF	Salto de línea
11	VT	Tabulador vertical
12	FF	Salto de página
13	CR	Retorno de carro
14	SO	Shift out
15	SI	Shift in
16	DLE	Escape unión datos
17	DC1	Control disposit. 1
18	DC2	Control disposit. 2
19	DC3	Control disposit. 3
20	DC4	Control disposit. 4
21	NAK	Confirmación negativa
22	SYN	Sincronización
23	ETB	Fin bloque texto
24	CAN	Cancelar
25	EM	Fin del medio
26	SUB	Sustitución
27	ESC	Escape
28	FS	Separad. de archivos
29	GS	Separad. de grupos
30	RS	Separad. de registros
31	US	Separad. de unidades

Valor decimal	Carácter	Valor decima	Carácter
32		81	Q
33	!	82	R
34	"	83	S
35	#	84	T
36	$	85	U
37	%	86	V
38	&	87	W
39	`	88	X
40	(	89	Y
41	)	90	Z
42	*	91	[
43	+	92	\
44	,	93	]
45	-	94	^
46	.	95	_
47	/	96	`
48	0	97	a
49	1	98	b
50	2	99	c
51	3	100	d
52	4	101	e
53	5	102	f
54	6	103	g
55	7	104	h
56	8	105	i
57	9	106	j
58	:	107	k
59	;	108	l
60	<	109	m
61	=	110	n
62	>	111	o
63	?	112	p
64	@	113	q
65	A	114	r
66	B	115	s
67	C	116	t
68	D	117	u
69	E	118	v
70	F	119	w

71	G	120	x
72	H	121	y
73	I	122	z
74	J	123	{
75	K	124	|
76	L	125	}
77	M	126	~
78	N	127	Delete
79	Ñ		
80	O		

El código ASCII estaba pensado para evitar la aparición de errores durante la transmisión y el procesamiento de los datos. Los errores se pueden producir, especialmente durante la transmisión mediante líneas de comunicaciones, por errores en las líneas. Una técnica usada para conseguir evitar errores es el denominado ***código redundante***. Los códigos redundantes más sencillos y comunes requieren la inclusión de un bit de ***paridad***. Este bit de paridad se pone a 1 ó 0 según que el número de 1's en el dato sea par (cuando se usa paridad par la asignación del bit de paridad es la siguiente: si el número de 1's es par, el bit de paridad se pondrá a 1; en caso contrario se pondrá a 0) o impar (si se usa paridad impar el valor del bit de paridad se asigna de la forma siguiente: si el número de 1's es impar el bit de paridad valdrá 1; en caso contrario valdrá 0).

Por ejemplo, supongamos que el carácter representado es 0110101 y que se usa paridad par. En este caso, como el número de 1's es par, entonces el bit redundante será un 1 y se coloca como bit más significativo: **1**0110101 (el bit en negrita, más a la izquierda, es el bit de paridad).

De esta forma es fácil comprobar si se ha producido error en la transmisión de los datos. Imaginemos que por el ruido en la línea de transmisión se recibe realmente 00100101, en lugar de 00110101. Al recibirse el dato se comprueba si el número de 1's concuerda con el valor del bit de paridad. Al ser un 0 se espera que el número de bits sea par, y al no serlo, se pone de manifiesto la existencia de error.

Tipos de codificación de caracteres

Debido a la necesidad de utilizar más caracteres (el código ASCII original sólo soportaba caracteres anglosajones, por ejemplo) y la mejora de la tecnología de las técnicas de detección de errores se resolvió utilizar los 8 bits del código ASCII para caracteres. DE esta forma se definieron varios códigos de caracteres de 8 bits, que se

denominó **ASCII extendido**. En el ASCII extendido se pueden distinguir 2^8=256 caracteres, compuestos por los 128 del ASCII normal más 128 caracteres adicionales que permitieron utilizar otros caracteres. Para lenguas de la familia del latín se utilizó concretamente el estándar ISO-8859-1.

Sin embargo, el problema de estos códigos de 8 bits es que cada uno de ellos se define para un conjunto de lenguas con escrituras semejantes y por tanto no dan una solución unificada a la codificación de todas las lenguas del mundo. Es decir, no son suficientes 8 bits para codificar todos los alfabetos y escrituras del mundo.

Por esto surgió **Unicode**, que es un estándar industrial cuyo objetivo es proporcionar el medio por el cual un texto en cualquier forma e idioma pueda ser codificado para el uso informático, incluyendo hasta lenguas muertas. Ha definido más de cien mil caracteres.

Unicode se suele usar como una manera de asignar un código único a cada carácter utilizado en los lenguajes escritos del mundo pero el almacenamiento de esos números es otro tema y además mucho del software escrito puede manejar solamente codificación de caracteres de 8 bits.

Para representar los caracteres Unicode se utiliza la norma UTF, siendo **UTF-8** la más popular. UTF-8 (8-bit *Unicode Transformation Format*) es un formato de codificación de caracteres Unicode que utiliza símbolos de longitud variable. Es capaz de representar cualquier carácter Unicode. Es de longitud variable (de 1 a 4 bytes por carácter Unicode). Incluye el ASCII de 7 bits, por lo que cualquier mensaje ASCII se representa sin cambios.

2.1.2 Codificación de números enteros

Habiendo visto la sección anterior, podríamos pensar en una forma de representar los números: basta con usar para cada dígito el carácter de entrada/salida correspondiente, es decir, su código ASCII.

¿Es esta una alternativa válida? Pensamos en los problemas de esta forma de representar los números. Para representar el número 234 necesitaríamos tres caracteres ASCII, es decir, 24 bits. Sin embargo, esto supone un desperdicio del espacio de almacenamiento, como ahora veremos.

Además, existe una segunda razón para no utilizar esta forma de representación: los circuitos encargados de sumar, restar, multiplicar y dividir (que componen la Unidad Aritmético Lógica) están pensados para trabajar con la representación binaria de los números.

Antes de seguir adelante necesitamos tener ciertas nociones sobre los sistemas de numeración. Ya hemos visto que el sistema binario puede utilizar únicamente dos símbolos (0 y 1), mientras que el sistema decimal podría utilizar 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

También podríamos considerar cualquier otro sistema de numeración. Por ejemplo, en base 4 podríamos usar 4 símbolos diferentes (0, 1, 2 y 3). De la misma forma, en el sistema hexadecimal son 16 los símbolos a utilizar (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

De cualquier forma, todos los sistemas de numeración se caracterizan por:

a) el número de símbolos que podemos usar (tantos como indica la base). Por ejemplo, en binario la base será 2. En el sistema de numeración decimal la base es 10 y en el sistema hexadecimal la base será 16
b) el valor de cualquier número se determina considerando la posición de los dígitos que lo componen, correspondiendo a cada posición una potencia determinada de la base

Para clarificar conceptos comenzaremos analizando algunas cifras en el sistema decimal. Por ejemplo, el número 255 representa el valor

$$255 = 2*10^2+5*10^1+5*10^0$$

Es decir, la cifra más a la izquierda representa las unidades (de ahí la potencia 10^0, es decir, 1). La cifra siguiente representa decenas (y por eso la potencia $10^1 = 10$) y la cifra más a la izquierda (cifra más significativa) representa centenas (y por tanto, la potencia de esta posición es $10^2 = 100$).

Pues igual ocurre con cualquier otro sistema de numeración. Si el número 255 está expresado en base 8, el valor real representado vendrá determinado por potencias de 8, de la forma siguiente:

Valor base 10 de $(255)_8 = 2*8^2 + 5*8^1 + 5*8^0 = 2*64 + 5*8 + 5*1 = (173)_{10}$

Es decir, 173 es el valor en base 10 representado por el número 255 expresado en base 8. ¿Es 255 un número correcto expresado en base

2? Pues obviamente no, ya que en un sistema binario de numeración las cifras sólo podrán estar compuestas de 0's y 1's. Si consideremos una combinación cualquiera de 0's y 1's podemos calcular el valor que representa en base 10 mediante una operación similar a la indicada previamente, pero ahora usando la base .2. Por ejemplo:

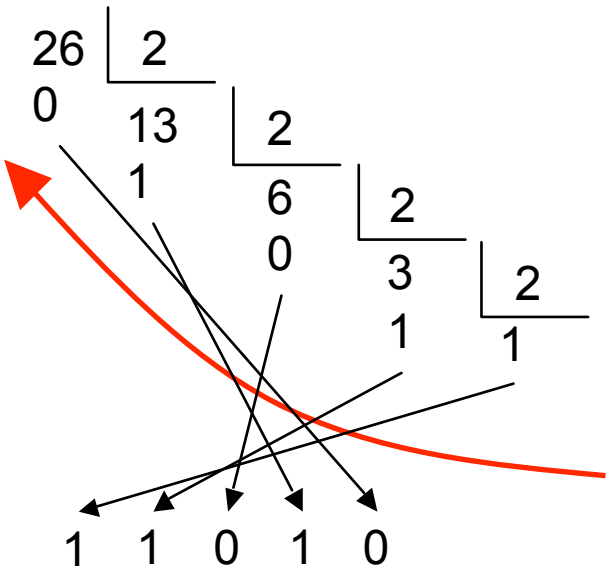
Valor en base 10 de $(11010)_2 = 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 26$.

Si este mismo número se hubiera representado utilizando los caracteres ASCII para 2 y para 6 se hubieran precisado 16 bits y de ahí el comentario sobre la falta de eficacia de dicha forma de representación. En el ejemplo visto anteriormente, si hubiésemos elegido la representación del número 234 mediante los correspondientes caracteres ASCII, entonces se habrían necesitado 24 bits, con los que se pueden representar en realidad 16.777.216 números distintos. Si pensamos que podemos utilizar signo, todos estos números estarían repartidos entre positivos y negativos, con lo que el mayor positivo estaría alrededor de los 8 millones. En definitiva, se aprecia que se desperdiciaría mucha capacidad de representación.

Ya que los números internamente se van a representar como secuencias de 0's y 1's cabe plantearse la forma de pasar de la representación decimal de un número a la secuencia de 0's y 1's que lo codifica. Es decir, la forma de pasar de base 10 a base 2. Para ello basta con dividir de forma sucesiva el número en base 10 por 2. Cuando se haya terminado, el número formado por el último cociente y los restos, desde abajo hasta arriba, formarán la secuencia de 0's y 1's buscada. Por ejemplo, apliquemos este sistema para obtener la representación binaria del número 26, en decimal:

26 : 2 = 13, resto 0
13 : 2 = 6, resto 1
6 : 2 = 3, resto 0
3 : 2 = 1, resto 1

Ya no se puede seguir dividiendo, ya que obtendría un 0 como próximo cociente. La forma de obtener la secuencia de 0's y 1's sería la siguiente:



a) cifra más significativa: el último cociente. Es la más significativa por haberse obtenido tras el mayor número de divisiones. El último cociente es un 1.
b) siguiente cifra, el último resto obtenido: 1
c) siguiente cifra: el resto anterior, 0
d) siguiente cifra: el resto anterior, 1
e) siguiente cifra: el primer resto, 0

Por tanto, la secuencia final será 11010. Como se ve, coincide todo, ya que el valor en base 10 de esta secuencia, calculado previamente, es precisamente 26.

El proceso será exactamente el mismo si se quisiera hacer el paso a cualquier otra base (dividiríamos por la base, siguiendo el procedimiento anteriormente explicado).

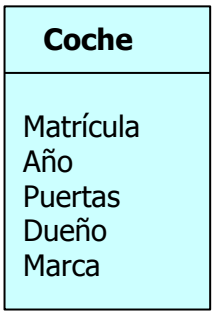
Una vez visto el concepto de sistema de numeración, puede observarse que los números grandes necesitan más 1's y 0's que los pequeños. Sin embargo, en los ordenadores se emplea una cantidad fija de bits para representar números. ¿Qué implica esto? Pues que el número de valores enteros que se pueden representar está limitado por el número de bits empleado para las secuencias de 0's y 1's. Supongamos una máquina en que únicamente se pueden emplear 8 bits para manipular números enteros. Esto significa que el número más grande a representar es 11111111, cuyo valor decimal se puede determinar tal y como hemos visto antes (como truco, el máximo valor corresponde a 2^8-1). Por tanto, con 8 bits podemos llegar a representar valores comprendidos entre 0 y 255. Para comprobar esto determinemos la representación binaria del número 255.

Sin embargo, si todos los bits se emplean para representar cifras, ¿qué ocurre con los números negativos? Para solucionar este problema se destina un bit a signo, asociado a la posición más significativa:

signo	1	0	1	1	1	0	0

Sin embargo, el uso de este bit de signo implica que se pierde un bit para representar cifras. No obstante, se pueden seguir representando el mismo número de valores diferentes, pero ahora repartidos entre positivos y negativos. Esto se va más claro con el siguiente esquema:

a) Sin bit de signo, podemos representar tantos números positivos como los comprendidos entre las secuencias 00000000 y



11111111. Es decir, desde 0 a 255 en decimal.

b) Si se considera el bit de signo, podremos representar tanto números positivos como negativos. Supongamos que los números negativos se representan mediante un 1 en el bit de signo. De esta forma, todos los números negativos a representar oscilarán entre 10000000 y 11111111. Esto significa que los números negativos oscilarán entre –0 y –127. Con respecto a los positivos, irán comprendidos entre 00000000 y 01111111, oscilando entre 0 y 127. Al final, se pueden representar 256 símbolos diferentes, con el único problema de disponer de una doble representación para el valor 0 (con signo negativo y positivo). Pero el número de "cosas" diferentes a representar depende únicamente del número de bits empleados y no del sistema de representación usado.

Como se ha indicado con anterioridad, el tamaño de los números enteros a manejar dependerá del número de bits empleados a tal fin. En la actualidad se suelen emplear 2 ó 4 bytes para la representación de números enteros (siempre con signo).

Es muy importante que tengamos esto en cuenta. Los números enteros están pensados para representar valores que no sean muy grandes. Cuando estemos usando Access, tendremos que asignar

tipos de datos a los valores que queremos almacenar. Imaginemos que queremos guardar datos sobre coches. Diseñamos una tabla que indica que los campos que estamos interesados en almacenar son: matrícula (clave), año de compra, número de puertas, dueño y marca. Esto se representa de la siguiente forma:

Pues, siguiendo los comentarios previos, el ordenador es completamente incapaz de conocer los tipos de datos necesarios para cada uno de los campos de información. Es el usuario de Access el responsable de hacer esta asignación. ¿Cuáles serían los campos para los que podría pensarse en los números enteros como tipo de datos? Pues en el año de compra (tanto si se usan 2 bytes como 4 hay espacio suficiente). Lo mismo ocurre con el número de puertas.

Para la matrícula, ya que contendrá caracteres y letras, no resulta conveniente el tipo entero. Tampoco conviene para cifras grandes o decimales. A la hora de decidir el tipo de datos para cada uno de los datos a manejar conviene también pensar en el uso que se dará a dicha información. Pensemos por ejemplo en un número de teléfono. Está claro que no podría ser representado como un número entero (9 cifras decimales no caben en 2 bytes). Tendríamos entonces que usar alguno de los tipos de datos que permiten representar valores mayores. Pero, ¿necesitamos representar esta información como un número? Es decir, ¿tendré alguna vez que sumar o restar números de teléfonos? Los números de teléfono podrían representarse, perfectamente, mediante caracteres de entrada/salida.

2.1.3 Codificación de números reales

Los números reales son aquellos que pueden representar cifras decimales (0’25, 100’34, etc.). Para representarlos se utiliza la notación científica, que consta de:

a) signo, que permite distinguir entre valores positivos y negativos
b) mantisa, cifras que lo componen
c) exponente, potencia de 10 asociada

Por ejemplo, el número 34’5 podría representarse de la siguiente forma:

a) signo positivo (0)
b) mantisa: 345
c) potencia de 10 por la que hay que multiplicar la mantisa para obtener el valor deseado (34’5). Para obtener ese valor tendríamos que dividir la mantisa por 10, lo que equivale a multiplicar por 10^{-1}. Es decir, el exponente sería -1 en este caso

Para ahorrar en la representación y evitar tener que considerar la coma, se considera la representación normalizada, en que la coma se sitúa siempre en la primera posición, ajustando el exponente de forma adecuada. Veamos algunos ejemplos:

Número	Normalizado	Signo	Mantisa	Exponente
31’18	$0’3118*10^{2}$	+	3118	2
0’17	$0’17*10^{0}$	+	17	0
-0’005	$-0’5*10^{-2}$	-	5	-2

Para cada una de las partes de la notación anterior se usa codificación binaria. En la práctica se emplean dos formas para representar valores de distinto tamaño:

a) simple precisión. Se usa 1 bit de signo, 23 bits para la mantisa y 8 bits para el exponente
b) doble precisión: 1 bit de signo, 52 bits para la mantisa y 11 para el exponente

El uso de una u otra dependerá de lo grande o pequeño que pueda ser el número a manipular. Por ejemplo, para denotar distancias astronómicas se podría usar la doble precisión. Si se quieren manejar temperaturas, bastaría con la precisión simple (de largo.....).

Usualmente cada lenguaje de programación define los tamaños de los tipos de datos usados, lo que determina los máximos valores a almacenar con cada uno de ellos (ocurre igual con otros programas, como Access, por ejemplo). Al manejar dichos programas deberemos de ser conscientes de la capacidad concreta de representación de cada tipo.

En Access, por ejemplo, los tipos de datos numéricos a usar se indican en la tabla siguiente (copiada directamente de la ayuda de este programa):

Setting	Description	Decimal precision	Storage size
Byte	Stores numbers from 0 to 255 (no fractions).	None	1 byte
Decimal	Stores numbers from -10^38 -1 through 10^38 -1 (.adp) Stores numbers from-10^28 -1 through 10^28 -1 (.mdb)	28	12bytes
Integer	Stores numbers from –32,768 to 32,767 (no fractions).	None	2 bytes

Long Integer	(Default) Stores numbers from –2,147,483,648 to 2,147,483,647 (no fractions).	None	4 bytes
Single	Stores numbers from –3.402823E38 to –1.401298E–45 for negative values and from 1.401298E–45 to 3.402823E38 for positive values.	7	4 bytes
Double	Stores numbers from –1.79769313486231E308 to –4.94065645841247E–324 for negative values and from 1.79769313486231E308 to 4.94065645841247E–324 for positive values.	15	8 bytes

Los tipos de datos asociados a números enteros serían ***byte***, ***integer*** y ***long integer***. El resto se corresponde con valores reales (que el ordenador manejaría tal y como hemos visto, con doble o simple precisión, según convenga).

2.1.4 Codificación de datos lógicos

Los datos lógicos sólo permiten representar dos valores: verdadero o falso. ¿Cuántos bits se necesitan para codificar un dato de tipo lógico? Pues basta emplear un bit, que en el caso de ser 0 se interpreta como falso y si es un 1 se interpreta como verdadero. Las operaciones que se pueden realizar con los datos de tipo lógico son:

a) AND (Y). Se trata de una operación que actúa sobre dos operandos. La tabla que indica el resultado de esta operación es:

AND	F	V
F	F	F
V	F	V

Es decir, el resultado de la operación es Verdadero sólo en el caso en que los dos operandos tengan también el valor Verdadero.

b) OR (O). También es una operación sobre dos operandos. Se incluye la tabla indicando el resultado de esta operación en los 4 posibles casos:

OR	F	V
F	F	V

V	V	V

En este caso, el resultado es verdadero siempre que uno de los operandos tenga este valor.

c) NOT (Negación). Este operador actúa sobre un único operando, invirtiendo su valor. Es decir, NOT Verdadero = Falso y NOT Falso = Verdadero.

Aunque estas operaciones pueden parecer triviales, en realidad todo el trabajo de un ordenador se hace en base a ellas: sumas, multiplicaciones, divisiones, etc. Los circuitos lógicos que componen el hardware de la unidad aritmético lógica no pueden realizar más que este tipo de operaciones elementales. Obviamente, en la unidad aritmético lógica se agrupan los componentes básicos para el resultado final de su actividad sea un cálculo aritmético.

2.1.5 Codificación de imágenes y sonido

El ordenador, en principio, no dispone de una forma predefinida para representar ni imágenes ni sonidos, al contrario de lo que ocurre con los tipos de datos vistos hasta ahora. Esto supone que el tratamiento de imágenes y sonido depende de programas específicos para ello.

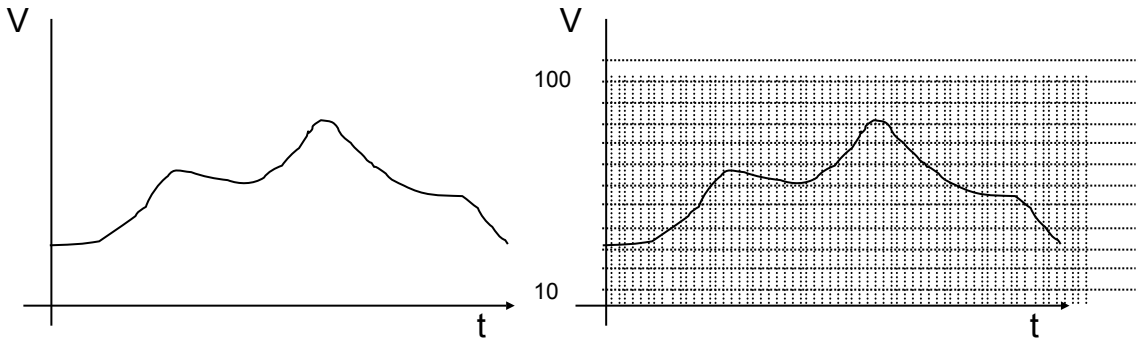
De cualquier forma, siguen siendo válidos los comentarios previos sobre la forma de representar la información dentro del ordenador: sólo mediante 0’s y 1’s. Por tanto, habrá que convertir la imagen en algo que, en última instancia, pueda convertirse en 0’s y 1’s. Esto es lo que hacen los múltiples formatos de codificación de imágenes y sonidos.

a) Codificación de imágenes. Lo que termina almacenándose, sea cual sea el formato (jpeg, mpeg, tiff, gif, png, bmp, etc.), sigue la siguiente estructura:

   - Cabecera indicando características de la imagen, como tamaño en píxeles, si es en color o blanco y negro, etc
   - Datos de la imagen en sí. Para cada píxel se suelen usar tres números, indicando los niveles de color rojo, verde y azul (si la imagen fuera en blanco y negro bastaría un único valor indicando el nivel de gris presente en el píxel). Si deseamos mucha fidelidad en los colores representados será preciso representar los niveles de rojo, verde y azul con un mayor número de bits. Por ejemplo, si se emplea un byte para representar cada nivel de color, un píxel

precisará usar 3 bytes para describir el color final. Con 8 bits se pueden representar 256 niveles diferentes. Si esto no fuera suficiente se deben usar más bytes para almacenar los niveles de color. Obviamente, el coste a pagar será un mayor espacio de almacenamiento usado.

b) Con respecto al sonido, el problema es aún mayor, ya que, en principio, cualquier sonido es una señal continua. Para poderla representar en el ordenador es preciso discretizarla, tomando muestras de ella de forma regular. El valor de estas muestras es lo que se convierte a valor binario, siendo esta combinación de 0's y 1's la que se almacena en el ordenador y a partir de la cual habrá que reconstruir una aproximación de la señal original. Todo este proceso se muestra en la imagen siguiente.



En definitiva, la señal continua queda sustituida por los valores obtenidos en cada una de las muestras: 31-32-33-35-……..-30. Estos valores de muestras se codifican, siguiendo alguno de los esquemas disponibles para sonido (mp3, por ejemplo), convirtiendo en 0's y 1's las medidas obtenidas para las muestras.

El vídeo al ser una conjunción de imágenes consecutivas (fotogramas) y audio sincronizado con dichas imágenes, se almacenaría como imágenes y sonido.

2.1.6 Códigos intermedios

En determinadas circunstancias es preciso analizar los datos codificados en binario (pirateo, criptografía, investigación policial, etc.). En estos casos, considerar la secuencia de 0's y 1's es complicado y difícil de manejar. Para evitar tratar única y exclusivamente con 0's y 1's suelen emplearse algunos sistemas de codificación intermedios, como octal y hexadecimal. Se emplean sistemas de numeración en bases que sean múltiplo de dos, ya que ello facilita la conversión a y desde binario.

Para convertir una cadena binaria de 0's y 1's a formato octal (donde podemos emplear los dígitos entre 0 y 7) basta con agrupar la cadena de bits en grupos de tres ($8 = 2^3$) y hacer la conversión siguiendo la tabla indicada a continuación:

Dígito	**Sec. binaria**
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

Por ejemplo, si tenemos 10110110101 la conversión al sistema octal de representación se haría de la siguiente forma: se agrupan los bits de tres en tres, empezando por la derecha, y posteriormente se realiza la conversión de los grupos utilizando la tabla incluida arriba.

10	110	110	101
2	6	6	5

Para realizar el paso al sistema hexadecimal se sigue una estrategia parecida. La base en este caso es 16 (2^4), por lo que los bits se agruparán de 4 en 4. La tabla de conversión a utilizar es ahora:

Dígito	**Sec. binaria**
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
A	1010

B	1011
C	1100
D	1101
E	1110
F	1111

La anterior cadena de ceros y unos en hexadecimal se traduce de la siguiente forma:

101	1011	101
5	B	5

Por supuesto, también es posible hacer el paso directamente desde decimal, utilizando el proceso de división por la base deseada (8 para octal y 16 para hexadecimal)

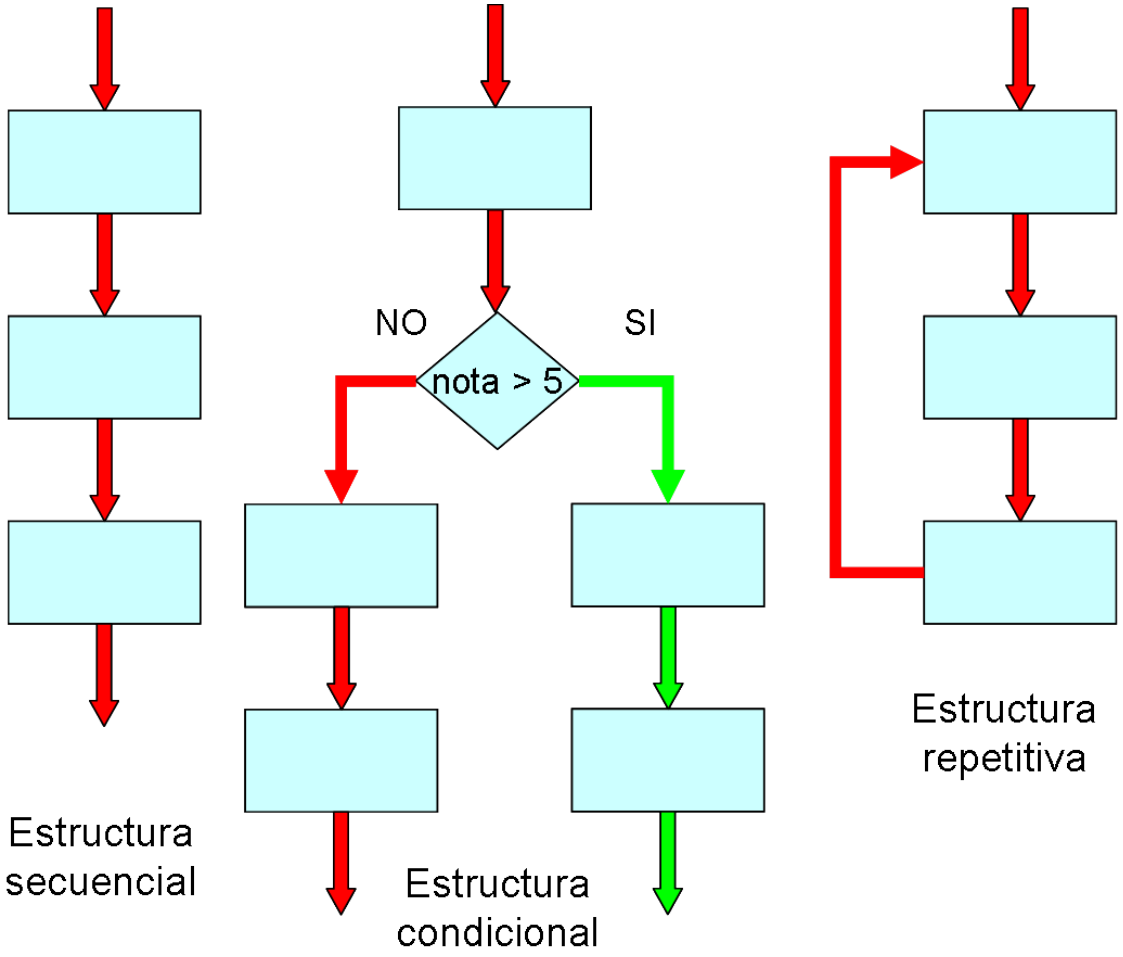
2.2 Representación interna de instrucciones

Vista la forma en que los datos se convierten en 0's y 1's, haciéndolos disponibles para su tratamiento por parte de los circuitos electrónicos del ordenador, pasamos a analizar la forma en que se comunican las instrucciones necesarias para que los datos se transformen de acuerdo a nuestros intereses.

Como siempre, el aspecto central de todo esto viene determinado por la dificultad del ordenador en comprender "conceptos" complejos. Al igual que no puede distinguir más que entre cosas diametralmente opuestas y sencillas (corriente – no corriente), las operaciones que puede realizar son muy elementales. Esto significa que cualquier operación de transformación de datos ha de basarse en dicho conjunto elemental de instrucciones.

Las instrucciones que puede realizar un ordenador se pueden clasificar de la siguiente forma:

a) instrucciones de transferencia, encargadas de mover los datos entre las unidades funcionales del ordenador (por ejemplo, para conseguir que un dato tecleado se almacene en la memoria o para conseguir la suma de dos valores almacenados en diferentes posiciones de memoria)
b) instrucciones de tratamiento, encargadas de realizar operaciones aritméticas y lógicas
c) instrucciones de control. Permiten escapar del orden secuencial de ejecución de instrucciones. El ordenador va ejecutando sentencias, una tras otra, en un orden determinado. Esto es lo que se denomina orden secuencial. Sin embargo, en determinadas situaciones es aconsejable que una operación se haga (o no) en función del valor de una operación previa. Esto supone algo diferente al esquema secuencial habitual: haz lo que indica la próxima instrucción. Estas situaciones se ponen de relieve mediante el esquema siguiente.



Se aprecia que en la estructura secuencial se produce un encadenamiento de las sentencias, de forma que el final de la ejecución de una supone el comienzo del tratamiento de la siguiente. Este será el comportamiento más habitual que se producirá en la ejecución de programas. La estructura condicional, por su parte, permite especificar tratamientos diferentes para casos diferenciados. En el gráfico se muestran dos líneas de ejecución: una para el caso de los aprobados y otra para los suspensos. Pero podrían considerarse muchas más, tantas como sean necesarias. Por último, la estructura repetitiva permite volver atrás, de forma que se permita la repetición controlada (hasta que deje de cumplirse una condición) de un conjunto de sentencias.

Es muy importante poder disponer de estructuras condicionales, que permitan ejecutar sentencias en función de lo que ha ocurrido con la ejecución de otras instrucciones previas. Y quizás, lo que es más importante, la estructura repetitiva, ya que de hecho, el funcionamiento general del ordenador es una repetición continua de la secuencia *busca instrucción – entiende instrucción – ejecuta instrucción*, que se repite de forma indefinida. Si no se dispusiera más que de la estructura sencuencial sería difícil conseguir programas que fuesen realmente útiles.

Sea cual sea el tipo de instrucción de que se trate, al final tendrá que ser convertida en una secuencia de 0's y 1's. Pues bien, habrá un ***código binario*** asignado a cada una de las instrucciones que entiende el procesador. Este conjunto de instrucciones se denomina código máquina y es particular para cada tipo de procesador.

Todas las instrucciones que componen el juego de instrucciones se estructuran de la siguiente forma:

a) una primera parte, denominada código de operación. Se trata de un conjunto de bits que codifican la instrucción de que se trate
b) la segunda parte representa los argumentos (datos) que dicha instrucción necesita.

Por ejemplo, si se trata de una instrucción de suma, la primera parte corresponderá al código correspondiente a dicha instrucción, mientras que la segunda parte indicará los números que deben sumarse (o el lugar en que localizarlos). Según sea el tipo de instrucción, los argumentos indicarán:

a) para la instrucciones de transferencia indicarán la dirección de memoria donde se encuentran los datos a mover y la dirección de destino (es decir, a dónde hay que transferirlos)

b) en las operaciones de tratamiento los argumentos indicarán los datos a manipular o la localización de los mismos. En cuyo caso, será la propia instrucción la que desencadene diversas instrucciones de transferencia para obtener los valores con los que trabajar
c) en las operaciones de control el argumento indicará la dirección de memoria en que se encuentra almacenada la próxima instrucción a ejecutar

Este código máquina es lo que realmente entiende el ordenador. Pero, hay varias cuestiones a tener en cuenta al respecto. Todas ellas tienen que ver con la dificultad inherente a manejar directamente este lenguaje máquina:

a) cada procesador, como se ha indicado, tiene su propio juego de instrucciones. De esta forma, si me dedicara a escribir programas en código máquina para un determinado procesador, sólo servirían para máquinas que tuvieran dicho procesador. Esto no es práctico, como se explicará a continuación
b) Los programas escritos en código máquina son fáciles de entender para el ordenador, pero no para las personas. Sólo tienen 0's y 1's
c) Si quisiera programar tareas complejas, en base a las operaciones tan sencillas que componen el código máquina, la tarea sería realmente ardua y la labor de programación muy compleja

Por tanto, la solución a estos problemas radica en no usar directamente este lenguaje de bajo nivel (bajo alude aquí a cercanía a la máquina, a la cacharrería). Por este motivo se utilizan los lenguajes de programación de alto nivel, donde la especificación de las instrucciones a aplicar sobre los datos puede expresarse con un lenguaje cercano al lenguaje natural. En este lenguaje yo puedo expresar sentencias del estilo de:

```
si (nota > 5)
  entonces
     nota = aprobado
  en caso contrario
     nota = suspenso
escribir nota
```

Obviamente, esta forma de escribir programas resulta mucho más clara y fácil (para los programadores). Pero, ahora, surge la pregunta. ¿Cómo se consigue que esta especificación de órdenes, en un

lenguaje de programación cercano al lenguaje natural, se haga comprensible para el ordenador?

Pues esta traducción entre lenguaje de alto nivel y lenguaje de bajo nivel corre a cargo de unos programas especiales denominados de forma general como traductores y diferenciados en dos tipologías: ***compiladores*** e ***intérpretes***. La diferencia entre unos y otros radica en la forma en que se hace el paso entre lenguajes:

a) en los compiladores el paso se hace de forma conjunta. Es decir, en primer lugar se leen todas las instrucciones de alto nivel y después se traducen de forma conjunta
b) en los intérpretes la conversión se hace sentencia a sentencia. Es decir, se lee una orden de alto nivel y se traduce al conjunto de instrucciones máquina necesarias para ejecutarla. Es más lento que un programa compilado (ya que el proceso de traducción es simultáneo) pero es independiente del procesador (un programa se compila para un procesador en concreto).

Las ventajas de este enfoque son obvias. Por un lado, los programadores pueden expresar las órdenes de manipulación de datos de forma natural, cercana a su modo habitual de expresión. Los programas así escritos son fáciles de entender. Un programador distinto al que hizo el programa podrá comprender la secuencia de operaciones que realiza cualquier programa, examinando su código. Por otra parte, los programas resultan más compatibles. La especificación de órdenes de alto nivel es universal y no depende del ordenador en que vaya a ejecutarse. Para conseguir que funcione en un procesador determinado necesitaremos el compilador específico para dicha máquina y ya. Una vez hecha la traducción se podrá ejecutar sin problema alguno.

Como ejemplo de lenguajes de alto nivel: C (compilado), C++ (compilado), Java (interpretado), Pascal (compilado) , Cobol (compilado), Fortran (compilado), Basic (interpretado). Java se está convirtiendo en uno de los programas más usados, debido a sus características.

Para facilitar la tarea de los programadores existen los llamados entornos de programación. Se trata de programas que incluyen las herramientas necesarias para escribir programas:

- editores de texto donde escribir las órdenes del programa
- compiladores, para comprobar la corrección del programa en todo momento

- depuradores, para poder examinar el funcionamiento del programa en ejecución, analizando paso a paso la forma en que se comporta
- generación de archivo final que puede ejecutarse de forma directa
- ayuda sobre las sentencias disponibles
- relación entre los distintos archivos que componen el programa

3.Almacenamiento de datos. Importancia de las bases de datos

El almacenamiento de los datos es un aspecto que el ordenador (y los sistemas operativos) dejan completamente en manos de los usuarios. Serán ellos quienes decidan la forma de agrupar los datos, en las carpetas que considere oportunas, haciendo que determinados conjuntos de datos aparezcan recogidos en un mismo archivo.

Por tanto, el archivo es la unidad elemental de almacenamiento de información, y contendrán la información que los usuarios deseen. Independientemente de su contenido, todos los archivos se almacenan de la misma manera. Por ejemplo, los archivos de texto almacenarán conjuntos de caracteres, mientras que los archivos ejecutables (con extensión exe) contendrán instrucciones para que el ordenador realice cierta manipulación con los datos.

La información que contiene un archivo puede estructurarse de forma que se reserve una parte del archivo para los datos de un objeto en concreto. Así, si se dispone de un archivo de alumnos, puede hacerse que para cada alumno se disponga de un ***registro*** que contenga sus datos. Gráficamente podría considerarse que el registro es cada una de las filas de la tabla siguiente:

Nombre	Apellidos	Edad	Dirección

Antonio	López Martínez	25	C/ Mesones, 37
….	….	….	….
….	….	….	….
Juan	Oliver Pérez	20	C/ Ronda, 18

Cada registro contiene información de distintos tipos. Cada una de ellas se denomina ***campo***. En este caso, los campos se utilizan para almacenar el nombre, los apellidos, la edad y la dirección (es decir, tenemos 4 campos para cada registro).

Pero el almacenamiento de la información en registros sólo sirve si un programa es capaz de tratar la información de esta forma. Es decir, el sistema operativo no tiene por qué saber la forma en que se organizan los datos. Para él todos los archivos son tratados de idéntica forma, como una secuencia de caracteres almacenados en un determinado espacio de algún dispositivo de almacenamiento.

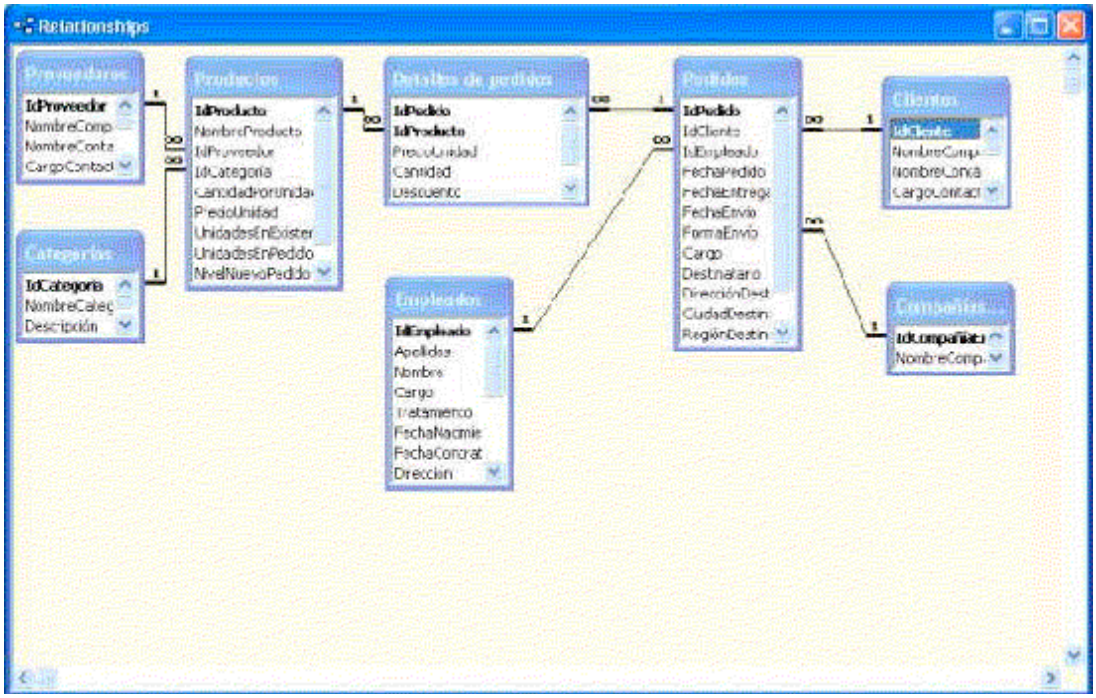
Si estoy interesado únicamente en mantener un archivo de esta forma, lo más sencillo es hacer un pequeño programa a medida para ello. Pero imaginemos que no sólo queremos guardar información sobre alumnos, sino también sobre personal de servicios, profesores, departamentos, unidades organizativas de la Universidad, centros, aulas, asignaturas, etc. En este caso, el volumen de información a manejar es muy grande y se requieren programas que la gestionen de forma eficiente. Además, surge otra cuestión. La información no sólo se almacena para usarla de forma estática (que también), sino para poder obtener, a partir de ella, relaciones (qué alumnos están matriculados en la asignatura de Fundamentos de Informática y Bases de Datos), listados (impresión ordenada de todos los alumnos), gráficos con las notas medias por años, etc., etc., etc.

En este caso suelen emplearse programas especialmente pensados para este propósito. Se denominan ***bases de datos***. Es importante tener en cuenta que estos programas no sólo facilitan el almacenamiento de información, de forma ordenada, sino también su gestión: mantenimiento, relaciones, listados, informes, etc. Nosotros, en el segundo cuatrimestre nos centraremos en el estudio del programa Access. Hemos de tener en cuenta que se trata de programas que facilitan la construcción, gestión y mantenimiento de las bases de datos particulares que los usuarios quieran construir.

Este programa de gestión de bases de datos contiene todos los elementos necesarios para realizar estas tareas:

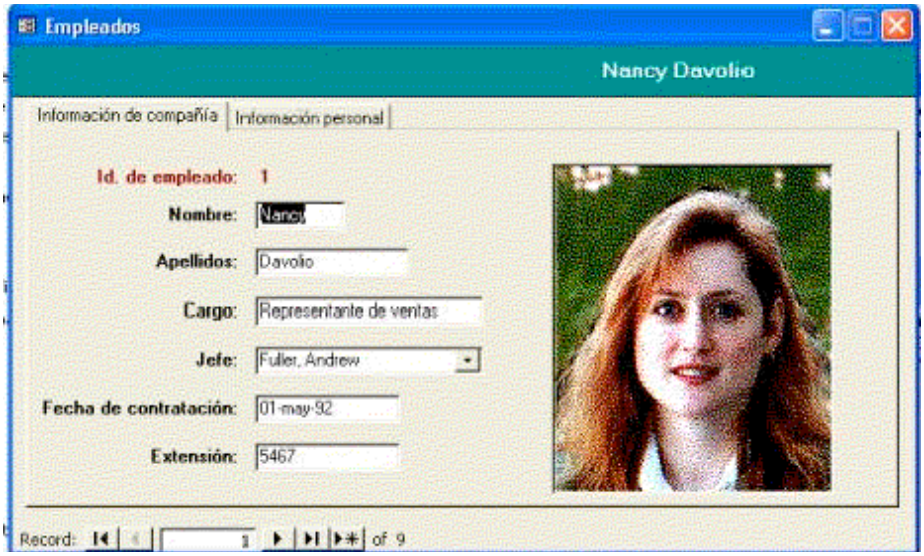
a) tablas. Se trata del elemento necesario para almacenar información de interés. En realidad, tendríamos que verlo como un archivo almacenado en forma de tabla, en el que cada fila sería un registro, compuesto por varios campos. En el lenguaje de bases de datos el archivo se denominará tabla, la fila tupla y los campos se denominarán atributos, pero se trata del mismo concepto.
b) consultas. Permiten recuperar información de las diversas tablas que contienen las bases de datos. Por ejemplo, si hemos construido una base de datos sobre videoclubs podemos preguntar por todas las películas que se encuentran en formato DVD (se trataría de una consulta afectando a la tabla que contiene las películas) o bien todos los actores que intervienen en las películas que estén en formato DVD (en este caso la consulta afecta a dos tablas: la de películas y la de actores).
c) formularios: ventanas que permiten realizar la introducción de datos de forma sencilla y cómoda.
d) informes: listados con la información que se desee, permitiendo darle la presentación que se considere oportuna.

La siguiente captura de pantalla de este programa muestra las tablas de una base de datos específica, así como las relaciones existentes entre ellas:



En definitiva, no sólo se permite el almacenamiento de los datos; también establecer relaciones entre los datos almacenados, hacer búsquedas de información fijando criterios de búsqueda, facilitar la

introducción de nuevos datos, etc. Este último aspecto es muy importante, y para ello se dispone de los formularios. En la imagen siguiente se muestra un formulario diseñado para facilitar la entrada de datos sobre una tabla concreta:



Access ofrece formas de introducir los datos directamente sobre las tablas, pero para aplicaciones profesionales no resulta una solución adecuada.

En realidad, siendo cuidadosos con el uso del lenguaje, deberíamos decir que Access en un sistema de gestión de bases de datos. Usándolo puedo crear tantas bases de datos como sea necesario. Access ayuda en la tarea de creación, gestión y mantenimiento. Hay que señalar que Access es un sistema de gestión de bases de datos pensado especialmente para sistemas de tamaño moderado. Por tanto, no resultaría eficaz en la gestión de grandes masas de datos: transacciones en Internet, aplicaciones bancarias, gestión de grandes almacenes, etc. Estos ámbitos de aplicación requieren otras aplicaciones de gestión de bases de datos más potentes.

4.Principios de Sistemas Operativos

Ya hemos indicado que el sistema operativo es una aplicación especialmente importante, ya que sin ella ninguna otra podría ejecutarse en el ordenador.

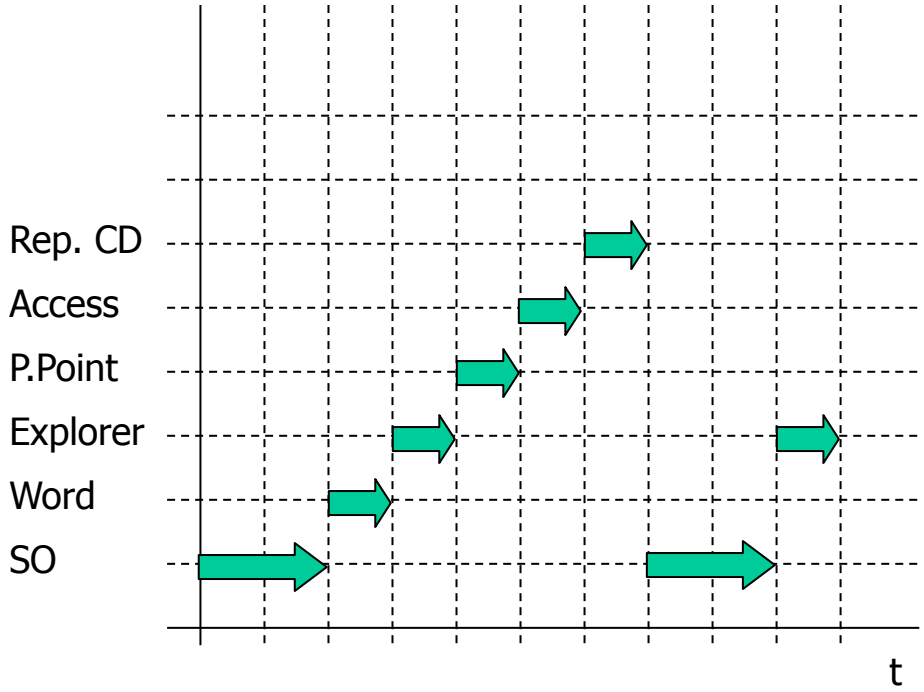
En realidad, para ser precisos, convendría decir que un sistema operativo se compone de un conjunto de programas que trabajan de forma coordinada para conseguir los siguientes objetivos:

a) facilitar al usuario y al resto de aplicaciones el uso de los recursos hardware de la máquina
b) gestionar de forma eficiente los recursos hardware del ordenador, repartiendo su uso entre los programas y usuarios que hacen uso del ordenador

Esas dos tareas generales se pueden desglosar en una serie de funciones más específicas centradas en recursos concretos:

a) gestión de uso del procesador. Para permitir que varios programas se ejecuten de forma simultánea (procesador de texto, navegador de Internet, hoja de cálculo, etc.) es necesario hacer que el tiempo de funcionamiento de la CPU se reparta entre todas las tareas en ejecución (incluyendo las necesarias para el propio funcionamiento del sistema operativo). Esto debe realizarse de forma transparente al usuario, de modo que quién trabaje en la máquina no tenga la sensación de que algunas aplicaciones han quedado desatendidas. Esta forma de repartir el tiempo entre diversas tareas se denomina **tiempo compartido**. El gráfico siguiente muestra la típica distribución de rodajas de tiempo entre las diversas tareas:

b)



gestión del espacio de almacenamiento, mediante el mantenimiento de un sistema de archivos ordenado de forma jerárquica y estructurado gracias al uso de carpetas
c) gestión del espacio de memoria. Aquí el problema es cómo repartir la memoria principal entre los distintos programas en

ejecución, de forma que se optimice su funcionamiento. Está claro que el sistema operativo ocupará continuamente parte de este espacio de memoria

d) control y comunicación con los periféricos. Para ello el sistema operativo se apoya en unos programas especiales denominados ***controladores*** (drivers) de dispositivos. Estos programas tienen la misión de permitir al sistema operativo la comunicación con dichos periféricos. La mayor parte de las veces esta información debe ser en ambos sentidos: del ordenador al periférico y viceversa
e) protección de los recursos, evitando el uso indebido de los recursos y de la información. Esto es especialmente importante cuando el ordenador se encuentra conectado, cuando usuarios remotos podrían hacer uso malintencionado del equipo
f) ofrecer una interfaz de usuario sencilla para que los usuarios puedan hacer uso de los recursos del ordenador de forma sencilla. Se duele diferenciar entre las interfaces en modo texto y las interfaces en modo gráfico, generalmente pensadas en torno al uso de ventanas

Hay diversos criterios a usar para clasificar los sistemas operativos. Algunos de ellos se consideran a continuación:

a) según el número de usuarios que admiten. Según este punto de vista se podría distinguir entre sistemas operativos **monousario** y **multiusuario**. Los primeros están pensados para máquina en que sólo trabajará una persona, mientras que los segundos permiten repartir el uso de los recursos entre varias personas. En ellos es necesario definir el perfil de uso de cada usuario. Esto es lo que se denomina ***cuenta***. A cada usuario se le asigna una cuenta distinta (para acceder a la misma será necesario identificarse mediante el nombre de usuario y la clave), que lleva asociado el uso de una determinada carpeta (a partir de la cual podrá empezar a almacenar sus datos), una serie de permisos para acceder a otras informaciones, un conjunto de aplicaciones disponibles, etc.
b) según el número de tareas que se pueden ejecutar a la vez (entre las que se repartirá el tiempo de CPU). De acuerdo a este criterio se puede hablar de sistemas operativos **monotarea** (sólo una tarea a la vez, de forma que hasta que una no finalice no podrá ejecutarse ninguna otra) y sistemas operativos **multitarea** (la mayoría de los sistemas operativos modernos podrían encuadrarse en esta categoría).

c) según la interfaz de usuario. Se distingue entre sistemas con interfaz de usuario en modo texto (de comandos) y en modo gráfico.

La siguiente tabla muestra la clasificación de algunos sistemas operativos de acuerdo a estos criterios de clasificación.

Sistema operativo	Multiusuario	Multitarea	Interfaz
MS-DOS			Comandos
Windows 95, 98		X	gráfica
Windows NT, 2000, 2003 Server, XP, Vista	X	X	gráfica
Linux, Unix	X	X	comandos/ gráfica
Mac Os, Mac Os X	X	X	gráfica

5.Otras aplicaciones

Se englobarían aquí todas las demás aplicaciones que se pueden ejecutar en un ordenador. Especialmente podríamos distinguir las siguientes:

a) procesadores de texto. Uno de los más difundidos es Microsoft-Word. Se trata de un programa destinado a ayudar en la creación y mantenimiento de documentos de texto, aunque permite la inclusión de gráficos y aporta una pequeña herramienta para crear dibujos sencillos
b) hojas de cálculo. Microsoft-Excel es un buen ejemplo. Permite manejar información ordenado en celdas organizadas en forma de tabla, permitiendo definir relaciones entre celdas y trabajar de forma conjunta con los valores de grupos de ellas (por ejemplo para sumar su contenido, para determinar su media, etc.). Se utilizan, por ejemplo, para tareas de contabilidad, mantenimiento de listas de alumnos, etc.
c) gestores de bases de datos, como Microsoft-Access. Permiten construir bases de datos en las que organizar cualquier conjunto de información
d) aplicaciones de diseño gráfico, que permiten la realización de planos en 2 y 3 dimensiones
e) aplicaciones de programación: programas destinados a ayudar a escribir nuevos programas

f) diseñadores de presentaciones. Microsoft-PowerPoint, mediante el que se pueden realizar presentaciones con usos educativos, científicos, comerciales, etc.
g) software de entretenimiento
h) aplicaciones de uso específico: programas de contabilidad, programas de puntos de venta, programas de gestión de clientes, de gestión de almacenes, etc., etc., etc.