TEMA 2. ARITMETICA Y TIPOS DE DATOS EN EL COMPUTADOR

PRIMERA PARTE: TIPOS SIMPLES

Introducción

La información en el ordenador se guarda en celdas que sólo pueden estar en dos estados posibles: encendido o apagado. Habitualmente estos estados suelen representarse con números: encendido, uno y apagado, cero. De alguna manera en cada celda podemos guardar o bien un cero o bien un uno, pero normalmente en un ordenador necesitaremos guardar información distinta de ceros y unos.

En esta primera parte del tema veremos la manera de representar *cualquier* tipo de información en el ordenador.

La información básica que podremos guardar en un ordenador va a ser de diferentes tipos:

- Información booleana
- Números enteros (positivos y negativos).
- Caracteres
- Números reales.

A continuación vamos a estudiar la forma de representación de todos estos tipos de datos.

Información booleana

La información booleana es básicamente *Verdadero* y *Falso.* Eso equivale a dos valores, que es justamente lo que podemos guardar en el ordenador. Generalmente el valor verdadero se asocia al 1 y el valor falso al 0 (en lenguaje de programación C, el valor falso es 0, y el valor verdadero cualquier valor distinto de cero).

Números enteros (positivos y negativos)

Representación binaria de valores enteros positivos

En general un número cualquiera **N** entero positivo se puede representar por un número indeterminado de símbolos que tendrán un determinado valor por sí mismos y en función de la posición que ocupan. De esa manera hablaremos de un sistema de numeración en base **B** como de la representación de un número mediante un conjunto de B símbolos diferentes de la siguiente manera:

$$N = \sum_{i=0}^{n} a_i \cdot B^i$$

y el número **N** se puede representar como

$$a_n a_{n-1} a_{n-2} \ldots a_2 a_1$$

La base con que trabajamos habitualmente es base 10 (**B**=10) y los diez símbolos con los que escribimos normalmente los números (el alfabeto usado) son los 10 dígitos {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Por ejemplo, el número 237 en realidad lo que nos esta diciendo es que el valor que tiene es:

$$2*10^2+3*10^1+7*10^0$$

Este tipo de operaciones son efectuadas de forma inconsciente porque estamos acostumbrados a trabajar en base 10, por lo que sin necesidad de cálculos sabríamos de qué cantidad estamos hablando, en este caso 237.

Puesto que el ordenador sólo puede utilizar el 0 y el 1, o sea un alfabeto de sólo dos símbolos, la base a utilizar será la binaria (B=2).

Aplicando la fórmula anterior, cualquier número lo representaremos como una sucesión de unos y ceros que cumplen:

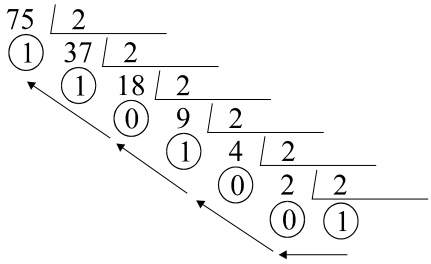
$$N = \sum_{i=0}^{n} a_i \cdot 2^i$$

Cambios de base

Paso de base decimal a base binaria:

El paso de base decimal a binaria se realiza mediante divisiones sucesivas

Ejemplo: Pasar 75)$_{10}$ a base binaria



$$75)_{10} = 1001011)_2$$

Paso de base binaria a base decimal:

Como las operaciones de suma y multiplicación las realizamos habitualmente en base decimal, no hay más que aplicar la fórmula $N = \sum_{i=0}^{n} a_i \cdot 2^i$

Ejemplo: Pasar $1001011)_2$ a base decimal

$1*2^0+1*2^1+0*2^2+1*2^3+0*2^4+0*2^5+1*2^6=1*1+1*2+0*4+1*8+0*16+0*32+1*64=1+2+8+64=75$

Para realizar el paso de binario a decimal es conveniente recordar las primeras potencias de 2:

$2^0 = 1$	$2^3 = 8$	$2^6 = 64$	$2^9 = 512$
$2^1 = 2$	$2^4 = 16$	$2^7 = 128$	$2^{10} = 1024$
$2^2 = 4$	$2^5 = 32$	$2^8 = 256$	

Códigos de representación intermedios: Octal y Hexadecimal

Como en base binaria los únicos dígitos que pueden utilizarse son el cero y el uno, normalmente es engorroso y largo escribir un número en base binaria, aunque sea ésta la única base que realmente utiliza el ordenador.

Para escribir números fácilmente convertibles a binario, pero con menor número de cifras se utilizan dos tipos de códigos intermedios: la base octal y la base hexadecimal.

En la representación octal la base es ocho, y el alfabeto los dígitos entre 0 y 7.

$\mathbf{B} = 8$

$\alpha = \{0, 1, 2, 3, 4, 5, 6, 7\}$

En la representación hexadecimal la base es dieciséis, y el alfabeto, que constará de dieciséis caracteres, es: los dígitos desde el 0 hasta el 9 (diez caracteres) más las letras (generalmente mayúsculas) desde la A hasta la F (6 caracteres)

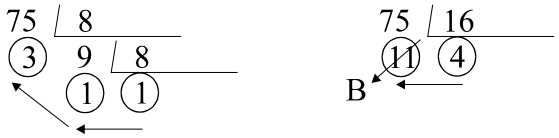
$\mathbf{B} = 16$

$\alpha = \{0, 1, 2, 3, 4, 5, 6, 7, A, B, C, D, E, F\}$

Paso de base decimal a base octal o hexadecimal:

La manera de pasar de base decimal a octal o hexadecimal es similar al paso a base binaria, pero dividiendo en cada caso sucesivamente por ocho o dieciséis.

Ejemplo: Pasar $75)_{10}$ a base octal y a base hexadecimal



$75)_{10} = 113)_8$ $\qquad$ $75)_{10} = 4B)_{16}$

Paso de base octal o hexadecimal a base decimal:

Al igual que en el caso binario nos limitaremos a utilizar la fórmula.

$N = \sum_{i=0}^{n} a_i \cdot 8^i$ en el caso octal y $N = \sum_{i=0}^{n} a_i \cdot 16^i$ en el caso hexadecimal

Ejemplo: Pasar $113)_8$ a base decimal

$3*8^0+1*8^1+1*8^2=3*1+1*8+1*64=3+8+64=75$

Ejemplo: Pasar $4B)_{16}$ a base decimal

$B*16^0+4*16^1=11*1+4*16=11+64=75$

Paso de base binaria a base octal o hexadecimal:

La base octal está basada en el ocho, mientras que la binaria esta basada en el dos. Puesto que ocho es dos elevado a tres, cada tres cifras binarias hacen una cifra octal. De esta manera, el paso de binario a octal se reduce a agrupar de tres en tres de derecha a izquierda las cifras binarias y evaluar su valor decimal.

Para la base hexadecimal tenemos un caso similar, ya que como sabemos que dieciséis es dos elevado a cuatro, podemos deducir que tenemos que agrupar las cifras en grupos de cuatro y evaluar, siempre recordando que valores superiores a 9 son representados mediante letras (A=10, B=11, C=12, D=13, E=14 y F=15)

Ejemplo: Pasar $1001011)_2$ a base octal y a base hexadecimal

$1001011)_2 = 001\ 001\ 011)_2 \rightarrow 001 = 1 \,/\, 001 = 1 \,/\, 011 = 3 \rightarrow 113)_8$

$1001011)_2 = 0100\ 1011)_2 \rightarrow 0100 = 4 \,/\, 1011 = 11 = B \rightarrow 4B)_{16}$

Paso de octal o hexadecimal a binario

Una vez visto el paso de binario a octal y hexadecimal, la transformación inversa es obvia: En el caso de la base octal sólo hay que pasar cada uno de los dígitos octales a tres cifras binarias y en el caso de la hexadecimal a cuatro cifras binarias. Sólo habría que recodar las transformaciones básicas en cada caso.

Octal	$000)_2 = 0)_8$	$011)_2 = 3)_8$	$110)_2 = 6)_8$
	$001)_2 = 1)_8$	$100)_2 = 4)_8$	$111)_2 = 7)_8$
	$010)_2 = 2)_8$	$101)_2 = 5)_8$	
Hexadecimal	$0000)_2 = 0)_{16}$	$0110)_2 = 6)_{16}$	$1100)_2 = C)_{16}$
	$0001)_2 = 1)_{16}$	$0111)_2 = 7)_{16}$	$1101)_2 = D)_{16}$
	$0010)_2 = 2)_{16}$	$1000)_2 = 8)_{16}$	$1110)_2 = E)_{16}$
	$0011)_2 = 3)_{16}$	$1001)_2 = 9)_{16}$	$1111)_2 = F)_{16}$
	$0100)_2 = 4)_{16}$	$1010)_2 = A)_{16}$	
	$0101)_2 = 5)_{16}$	$1011)_2 = B)_{16}$	

Representación de números enteros con signo (números enteros positivos y negativos)

Existen diferentes estrategias para realizar la representación del signo en los números.

Representación en signo y magnitud

La representación más sencilla y obvia es realizar algo similar a la manera habitual de representar los números con signo: Dedicar un espacio para indicar el signo del número. Así si hablamos de número '75' positivo, podemos indicarlo como '+75'. Y si hablamos del número '75' negativo lo indicaremos con el símbolo '-' precediendo al '75' es decir: '-75'. De alguna manera distinguimos la magnitud o valor del número (el 75) de su signo (+/-).

En la memoria del ordenador es imposible la representación del símbolo + o del símbolo -. Sólo se pueden representar unos y ceros. De manera que lo que haremos será dedicar un espacio concreto de la representación del número al signo, que también será un cero o un uno, pero que por estar en una cierta posición no indicará magnitud sino signo.

Si hablamos de un lugar determinado donde va a estar el signo, también tendremos que hablar de un número determinado de espacios para poder localizar el signo y la magnitud. Así, de ahora en adelante,

cualquier representación binaria de un número llevará pareja el número de bits en que se va a realizar la representación (y a su vez el método en que ha sido representado).

Ejemplo: Representar los números $75)_{10}$ y $-75)_{10}$ en base binaria, en signo magnitud en 8 bits (n=8)

$$75)_{10} = \underset{S}{0}\ 1\ 0\ 0\ 1\ 0\ 1\ 1)_{2SM}$$

$$-75)_{10} = \underset{S}{1}\ 1\ 0\ 0\ 1\ 0\ 1\ 1)_{2SM}$$

Si al realizar la representación apareciesen huecos (casillas sin rellenar) éstas se dejarán siempre a la derecha del signo y a la izquierda de la magnitud, y se rellenarán siempre con ceros.

Ejemplo: Representar los números $18)_{10}$ y $-18)_{10}$ en base binaria, en signo magnitud en 8 bits (n=8)

$18)_{10} = 10010)_2$

$$18)_{10} = \underset{S}{0}\ 0\ 0\ 1\ 0\ 0\ 1\ 0)_{2SM}$$

$$-18)_{10} = \underset{S}{1}\ 0\ 0\ 1\ 0\ 0\ 1\ 0)_{2SM}$$

El inconveniente de esta representación es básicamente la realización de operaciones (sumar o restar), en las que hay que comprobar el signo y dependiendo del signo realizar una operación u otra, comprobando cual de las magnitudes es mayor, etc.

Representación en complemento a 1

Dado x entero positivo o negativo, lo que haremos será aplicarle una transformación:

$$x` = (2^n - 1) + x$$

Y en las operaciones se tiene en cuenta el acarreo.

Ejemplo: Supongamos n=4. Representar todos los números posibles.

$2^4 = 10000 \rightarrow 2^4 - 1 = 1111$

$0 \rightarrow 1111 + 0 = 1111)_{2C1}$

$1 \rightarrow 1111 + 1 = (1)0000 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0000 + 1 = 0001)_{2C1}$

$2 \rightarrow 1111 + 10 = (1)0001 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0001 + 1 = 0010)_{2C1}$

$3 \rightarrow 1111 + 11 = (1)0010 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0010 + 1 = 0011)_{2C1}$

$4 \rightarrow 1111 + 100 = (1)0011 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0011 + 1 = 0100)_{2C1}$

$5 \rightarrow 1111 + 101 = (1)0100 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0100 + 1 = 0101)_{2C1}$

$6 \rightarrow 1111 + 110 = (1)0101 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0101 + 1 = 0110)_{2C1}$

$7 \rightarrow 1111 + 111 = (1)0110 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0110 + 1 = 0111)_{2C1}$

$8 \rightarrow 1111 + 1000 = (1)0111 \rightarrow$ (se tiene en cuenta el acarreo) $\rightarrow 0111 + 1 = 1000)_{2C1}$
(número no válido porque es igual que el –7)

$-1 \rightarrow 1111 - 1 = 1110)_{2C1}$

$-2 \rightarrow 1111 - 10 = 1101)_{2C1}$

$-3 \rightarrow 1111 - 11 = 1100)_{2C1}$

$-4 \rightarrow 1111 - 100 = 1011)_{2C1}$

$-5 \rightarrow 1111 - 101 = 1010)_{2C1}$

$-6 \rightarrow 1111 - 110 = 1001)_{2C1}$

$-7 \rightarrow 1111 - 111 = 1000)_{2C1}$

$-8 \rightarrow 1111 - 1000 = 0111)_{2C1}$ (número no válido porque es igual que el 7)

Viendo los resultados podemos ver que se cumplen algunas características.

- Los números positivos tienen como primer bit un cero, mientras que los negativos tienen un uno.
- Los números positivos tienen la misma representación que tenían pero completados con ceros a la izquierda hasta el número adecuado de bits de la representación.
- Los números negativos podríamos obtenerlos completando con ceros los números binarios positivos y cambiando ceros por unos y unos por ceros.

De esta forma podemos obtener la representación binaria en complemento a uno bien utilizando la fórmula (**$x' = (2^n - 1) + x$**) o bien siguiendo los siguientes pasos:

- Si el número a representar es positivo entonces se pasa a binario y se completa con ceros hasta la representación elegida.
- Si el número es negativo entonces se pasa su magnitud a binario, se completa con ceros hasta la representación elegida y finalmente se cambian ceros por unos y unos por ceros.

Con esta representación la suma y la resta se convierten en una sola operación: suma.

$$A - B = A + (-B)$$

Ejemplo 1: Supongamos n=4. Sumar A=5 y B=2.

$A = 5)_{10} = 0\ 1\ 0\ 1)_{2C1}$ $\quad B = 2)_{10} = 0\ 0\ 1\ 0)_{2C1}$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 0\ 1\ 0 \\ \hline 0\ 1\ 1\ 1 \end{array}$$

$A + B = 0\ 1\ 1\ 1)_{2C1}$

Ejemplo 2: Supongamos n=4. Sumar A=5 y B=-6.

$A = 5)_{10} = 0\ 1\ 0\ 1)_{2C1}$ $\quad B = -6)_{10} = -\ 0\ 1\ 1\ 0)_2 = 1\ 0\ 0\ 1)_{2C1}$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 1\ 0\ 0\ 1 \\ \hline 1\ 1\ 1\ 0 \end{array}$$

$A + (-B) = 1\ 1\ 1\ 0)_{2C1}$

Ejemplo 3: Supongamos n=4. Sumar A=5 y B=6.

$A = 5)_{10} = 0\ 1\ 0\ 1)_{2C1}$ $\quad B = 6)_{10} = 0\ 1\ 1\ 0)_{2C1}$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 1\ 1\ 0 \\ \hline 1\ 0\ 1\ 1 \end{array}$$

$A + B = 1\ 0\ 1\ 1)_{2C1}$ (*Hemos obtenido un número negativo de la suma de dos números positivo. Luego existe un error (error de overflow)*)

Ejemplo 4: Supongamos n=4. Sumar A=-5 y B=-6.

$A = -5)_{10} = -\ 1\ 0\ 1)_2 = 1\ 0\ 1\ 0)_{2C1}$ $\quad B = -6)_{10} = -\ 1\ 1\ 0)_2 = 1\ 0\ 0\ 1)_{2C1}$

$$\begin{array}{r} 1\ 0\ 1\ 0 \\ +\ 1\ 0\ 0\ 1 \end{array}$$

(1) 0 0 1 1 → El acarreo se vuelve a sumar → 0 0 1 1 + 1 = 0 1 0 0

A + B = 0 1 0 $0)_{2C1}$ (*Hemos obtenido un número positivo de la suma de dos números negativos. Luego existe un error (error de underflow)*)

El único pequeño inconveniente de esta representación es que el cero tiene dos representaciones de manera que tenemos representados 'sólo' 2^n-1 diferentes números, cuando podríamos tener representados 2^n.

Representación en complemento a 2

Dado x entero positivo o negativo, lo que haremos será aplicarle la misma transformación que antes pero añadiendo siempre uno, y no teniendo en cuenta luego el acarreo.

$$x`= (2^n - 1) + x + 1$$

Ejemplo: Supongamos n=4. Representar todos los números posibles.

$2^4 = 10000 \rightarrow 2^4 - 1 = 1111$

$0 \rightarrow 1111 + 0 + 1 = (1)\ 0\ 0\ 0\ 0 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 0\ 0\ 0)_{2C1}$

$1 \rightarrow 1111 + 1 + 1 = (1)\ 0\ 0\ 0\ 1 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 0\ 0\ 1)_{2C1}$

$2 \rightarrow 1111 + 10 + 1 = (1)\ 0\ 0\ 1\ 0 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 0\ 1\ 0)_{2C1}$

$3 \rightarrow 1111 + 11 + 1 = (1)\ 0\ 0\ 1\ 1 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 0\ 1\ 1)_{2C1}$

$4 \rightarrow 1111 + 100 + 1 = (1)\ 0\ 1\ 0\ 0 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 1\ 0\ 0)_{2C1}$

$5 \rightarrow 1111 + 101 + 1 = (1)\ 0\ 1\ 0\ 0 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 1\ 0\ 1)_{2C1}$

$6 \rightarrow 1111 + 110 + 1 = (1)\ 0\ 1\ 1\ 0 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 1\ 1\ 0)_{2C1}$

$7 \rightarrow 1111 + 111 + 1 = (1)\ 0\ 1\ 1\ 1 \rightarrow$ y NO se tiene en cuenta el acarreo → $0\ 1\ 1\ 1)_{2C1}$

$8 \rightarrow 1111 + 1000 + 1 = (1)\ 1\ 0\ 0\ 0 \rightarrow$ y NO se tiene en cuenta el acarreo → $1\ 0\ 0\ 0)_{2C1}$
(número no válido porque es igual que el –8)

$-1 \rightarrow 1111 - 1 + 1 = 1\ 1\ 1\ 1)_{2C1}$

$-2 \rightarrow 1111 - 10 + 1 = 1\ 1\ 1\ 0)_{2C1}$

$-3 \rightarrow 1111 - 11 + 1 = 1\ 1\ 0\ 1)_{2C1}$

$-4 \rightarrow 1111 - 100 + 1 = 1\ 1\ 0\ 0)_{2C1}$

$-5 \rightarrow 1111 - 101 + 1 = 1\ 0\ 1\ 1)_{2C1}$

$-6 \rightarrow 1111 - 110 + 1 = 1\ 0\ 1\ 0)_{2C1}$

$-7 \rightarrow 1111 - 111 + 1 = 1\ 0\ 0\ 1)_{2C1}$

$-8 \rightarrow 1111 - 1000 + 1 = 1\ 0\ 0\ 0)_{2C1}$

$-9 \rightarrow 1111 - 1001 + 1 = 0\ 1\ 1\ 1)_{2C1}$ (número no válido porque es igual que el 7)

Al igual que antes, viendo los resultados, podemos ver que se cumplen algunas características.

- Los números positivos tienen como primer bit un cero, mientras que los negativos tienen un uno (al igual que la representación en complemento a 1.)
- Los números positivos tienen la misma representación que tenían pero completados con ceros a la izquierda hasta el número adecuado de bits de la representación.
- Los números negativos podríamos obtenerlos completando con ceros los números binarios positivos y cambiando ceros por unos y unos por ceros y sumando uno al número resultante.

De esta forma podemos obtener la representación binaria en complemento a dos bien utilizando la fórmula (**$x' = (2^n - 1) + x + 1$**) o bien siguiendo los siguientes pasos:

- Si el número a representar es positivo entonces se pasa a binario y se completa con ceros hasta la representación elegida.
- Si el número es negativo entonces se pasa su magnitud a binario, se completa con ceros hasta la representación elegida y finalmente se cambian ceros por unos y unos por ceros.

Igual que antes, con esta representación la suma y la resta se convierten en una sola operación: suma.

$$A - B = A + (-B)$$

Ejemplo 1: Supongamos n=4. Sumar A=5 y B=2.

$A = 5)_{10} = 0\ 1\ 0\ 1)_{2C1}$ $\quad$ $B = 2)_{10} = 0\ 0\ 1\ 0)_{2C1}$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 0\ 1\ 0 \\ \hline 0\ 1\ 1\ 1 \end{array}$$

$A + B = 0\ 1\ 1\ 1)_{2C1}$

Ejemplo 2: Supongamos n=4. Sumar A=5 y B=-6.

$A = 5)_{10} = 0\ 1\ 0\ 1)_{2C1}$ $\quad$ $B = -\ 6)_{10} = -\ 0\ 1\ 1\ 0)_2 = 1\ 0\ 1\ 0)_{2C1}$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 1\ 1 \end{array}$$

$A + (-B) = 1\ 1\ 1\ 1)_{2C1}$

Ejemplo 3: Supongamos n=4. Sumar A=5 y B=6.

$A = 5)_{10} = 0\ 1\ 0\ 1)_{2C1}$ $\quad$ $B = 6)_{10} = 0\ 1\ 1\ 0)_{2C1}$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 1\ 1\ 0 \\ \hline 1\ 0\ 1\ 1 \end{array}$$

$A + B = 1\ 0\ 1\ 1)_{2C1}$ (*Hemos obtenido un número negativo de la suma de dos números positivo. Luego existe un error (error de overflow)*)

Ejemplo 4: Supongamos n=4. Sumar A=-5 y B=-6.

$A = -\ 5)_{10} = -\ 1\ 0\ 1)_2 = 1\ 0\ 1\ 1)_{2C1}$ $\quad$ $B = -\ 6)_{10} = -\ 1\ 1\ 0)_2 = 1\ 0\ 1\ 0)_{2C1}$

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ +\ 1\ 0\ 1\ 0 \\ \hline (1)\ 0\ 1\ 0\ 1 \end{array} \rightarrow \text{El acarreo se ignora en esta representación}$$

$A + B = 0\ 1\ 0\ 1)_{2C1}$ (*Hemos obtenido un número positivo de la suma de dos números negativos. Luego existe un error (error de underflow)*)

Caracteres

La representación de caracteres se realiza mediante una cierta correspondencia entre los caracteres que queremos representar y una serie de números binarios. Esta correspondencia entre las combinaciones de los símbolos disponibles y los caracteres es un convenio del mismo modo que puede serlo, por ejemplo, el código Morse.

La correspondencia más conocida y usada es el código ASCII (*American Standard Code for Information Interchange*), originalmente utilizado para comunicar información entre distintas máquinas por lo que, además de los caracteres normales utilizados en la escritura, puede representar una serie de caracteres con un cierto significado especial en los ordenadores, conocidos como *caracteres de control*.

El código ASCII original utilizaba siete bits para la representación de la información relativa a caracteres, y utilizaba un octavo bit (bit de control de errores) para comprobar la corrección de los otros siete, por ejemplo controlando la paridad de los bits (número par o impar de unos del número binario, por ejemplo el número binario 0110101, tiene número par de unos, por lo que el bit de control sería 0, mientras que 0100101 tendría como bit de control un 1.) Con los siete bits pueden llegar a representarse hasta un máximo de 2^7 caracteres (es decir, 128.)

Actualmente, y por la alta difusión del código ASCII para la representación de caracteres, se han introducido un número mayor de caracteres para poder representar caracteres internacionales (como vocales acentuadas, la 'ñ', la 'ç', etc.) Esta extensión del código ASCII se hace a costa del octavo bit y es específico de cada país. Una representación de uno de los códigos ASCII extendidos españoles podría ser el que se muestra en la tabla 1.

		Código ASCII estándar								Código ASCII extendido							
	Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Hex	Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	0000		DLE		0	@	P	`	p	Ç	É	á	▒	└	╨	α	≡
		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	0001	SOH	DC1	!	1	A	Q	a	q	ü	æ	í	▒	┴	╤	β	±
		1	17	33	49	65	81	97	11	129	145	161	177	193	209	225	241
2	0010	STX	DC2	"	2	B	R	b	r	é	Æ	ó	▓	┬	╥	Γ	≥
		2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	0011	ETX	DC3	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
		3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	0100	EOT	DC4	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
		4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	0101	ENQ	NAK	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
		5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	0110	ACK	SYN	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
		6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	0111	BEL	ETB	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
		7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	1000	BS	CAN	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
		8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	1001	HT	EM	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	θ	•
		9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	1010	LF	SUB	*	:	J	Z	j		è	Ü	¬	║	╩	┌	Ω	·
		10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	1011	VT	ESC	+	;	K	[	k	{	ï	¢	½	╖	╦	▉	δ	√
		11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	1100	FF	FS	,	<	L	\	l	|	î	£	¼	╝	╠	■	∞	ⁿ
		12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	1101	CR	GS	-	=	M	]	m	}	ì	¥	¡	╜	═	▌	∅	²
		13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	1110	SO	RS	.	>	N	^	n	~	Ä	₧	«	╛	╬	▐	∈	■
		14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	1111	SI	US	/	?	O	_	o	DEL	Å	*f*	»	┐	╧	■	∩	
		15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Caracteres de Control — Caracteres Gráficos

Caracteres de control:

- **NUL** ⇨ Carácter Nulo
- **SOH** ⇨ Comienzo de cabecera
- **STX** ⇨ Comienzo de texto
- **ETX** ⇨ Final de texto
- **EOT** ⇨ Fin de transmisión
- **ENQ** ⇨ Pregunta
- **ACK** ⇨ Confirmación positiva
- **BEL** ⇨ Señal acústica
- **BS** ⇨ Espacio atrás
- **HT** ⇨ Tabulador horizontal
- **LF** ⇨ Salto de línea
- **VT** ⇨ Tabulador vertical
- **FF** ⇨ Salto de página
- **CR** ⇨ Retorno de carro
- **SO** ⇨ Shift Out
- **SI** ⇨ Shift In
- **DLE** ⇨ Carácter de transparencia en tramas de datos
- **DC1** ⇨ Control dispositivo 1
- **DC2** ⇨ Control dispositivo 2
- **DC3** ⇨ Control dispositivo 3
- **DC4** ⇨ Control dispositivo 4
- **NAK** ⇨ Confirmación negativa
- **SYN** ⇨ Sincronismo
- **ETB** ⇨ Fin del bloque de texto
- **CAN** ⇨ Cancelar
- **EM** ⇨ Fin de dispositivo
- **SUB** ⇨ Sustitución
- **ESC** ⇨ Escape
- **FS** ⇨ Separador de ficheros
- **GS** ⇨ Separador de grupos
- **RS** ⇨ Separador de registros
- **US** ⇨ Separador de unidades

Otros caracteres:

- **SP** ⇨ Espacio en blanco
- **DEL** ⇨ Carácter de borrado

Números Reales

Es importante tener en cuenta que para la representación de números reales tendremos una cantidad finita de elementos, de manera que tendremos que trabajar con una cierta precisión (no se puede representar un número infinito de decimales.)

Un número real, en general, se puede representar de dos formas concretas: Mediante coma fija; y mediante coma flotante, también llamada notación científica.

Coma fija

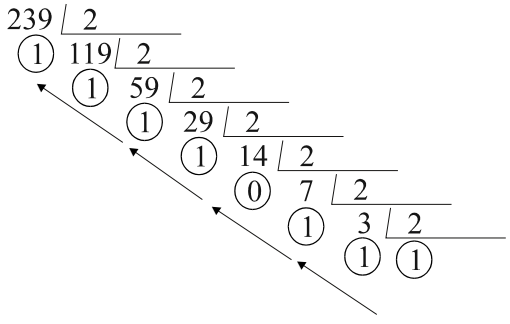
Para pasar de decimal a binario un número real deberemos pasar por un lado la parte entera y por otro la parte decimal.

La parte entera se pasa como hemos visto que se transformaban los números enteros: mediante divisiones sucesivas.

La parte decimal se pasa por multiplicaciones sucesivas, viendo la parte entera resultante de cada multiplicación.

Ejemplo: Pasar el número decimal 239'403 a binario.

Primero pasaremos la parte entera:



$239)_{10} = 11101111)_2$

La parte decimal mediante multiplicaciones sucesivas de las partes decimales:

0’403 * 2 = 0’806 (Obtenemos un 0)
0’806 * 2 = 1’612 (Obtenemos un 1)
0,612 * 2 = 1’224 (Obtenemos un 1)
0’224 * 2 = 0’448 (Obtenemos un 0)
0’448 * 2 = 0’896 (Obtenemos un 0)
0,896 * 2 = 1’792 (Obtenemos un 1)
0’792 * 2 = 1’584 (Obtenemos un 1)
...

$0’403)_{10} = 0’0110011...)_2$

El resultado será pues:

$$239’403)_{10} = 11101111’0110011...)_2$$

Importante: Un número decimal con un número finito de decimales, no tiene por qué tener un número finito de decimales en su representación binaria.

La pregunta que nos surgirá será ¿Cuántos decimales hay que obtener? La respuesta evidentemente depende de la cantidad de bits que tengamos para realizar la representación (y que comentaremos en la parte de coma flotante).

El número en coma fija es difícil de representar en el ordenador, básicamente porque no sabemos la posición en que va a quedar la coma, y como esta rodeada de unos y de ceros, no podríamos distinguirla de ellos.

Coma flotante

La representación en coma flotante se basa en colocar la coma en una posición determinada, de manera que siempre estará en esa posición y en la representación binaria no tendremos que preocuparnos por ella. El hecho de mover la coma, supondrá la aparición de una base (en nuestro caso 2) elevada a un exponente, que también tendremos que representar.

En general cualquier número real en cualquier base puede representarse en coma flotante moviendo la coma a la izquierda de la primera cifra significativa del número y multiplicando el número por la base elevada al exponente adecuado.

Por ejemplo:

$239’403)_{10} = 0’239403 * 10^3$

$0’00000345)_{10} = 0’345 * 10^5$

$16’76)_8 = 0’1676 * 8^2$

$11101111’011001)_2 = 0’11101111011001 * 2^8$

Una vez tenemos el número binario en coma flotante la representación se realizará separando por un lado la representación del número que contiene la coma (o mantisa) y del exponente que eleva la base.

La mantisa se escribe representando los valores que aparecen a la derecha de la coma (ya que el 0 y la coma siempre están en ese lugar.)

Al igual que en la representación de enteros, aquí también nos tendrán que informar del número de bits que debemos utilizar para la representación.

La representación binaria de un número real en coma flotante siempre (o casi siempre) utilizará la siguiente forma:

Signo / mantisa / exponente

El signo con la mantisa tiene una forma similar a un entero, de manera que puede representarse como los números enteros (complemento a dos), pero la forma habitual es mediante signo y magnitud.

El exponente es un número entero, de manera que cualquier método de representación de enteros serviría para representarlo. Sin embargo, es habitual que en la representación de números reales en coma flotante el exponente se represente con sesgo (es decir con un desplazamiento o exceso.)

La representación de números en binario con *n* bits es capaz de mostrar de forma natural números entre 0 y 2^n (todos positivos). Con el sesgo o desplazamiento lo que se hace es tener una aplicación que a los números negativos les haga corresponder números positivos. La transformación que hay que aplicar es:

$$X' = X + 2^{n-1}$$

Con esta transformación podemos representar números en el rango $-(2^{n-1}-1),\ldots -1, 0, 1,\ldots 2^n$

Ejemplo: Representa en coma flotante, con ocho bits para la mantisa (en signo magnitud) y 5 bits para el exponente (sesgado) el número 239'403)$_{10}$

$239'403)_{10} = 11101111'0110011)_2 = 0'11101111011011 * 2^8$

Mantisa: Signo: Positivo (0)

Magnitud: 11101111011011

Como el signo con la magnitud ocupa más de 8 bits, tendremos truncamiento (perdida de información menos significativa.) Es decir, sólo podremos representar realmente el **`0'1110111`**, en vez de **`0'11101111011011`**

Exponente: Valor: $8)_{10} = 1000)_2$

Sesgo: $2^{n-1})_{10} = 10000)_2$

Exponente con sesgo será:

$$1000 + 10000 = 11000$$

Resumiendo:

$$\underset{s}{0}\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 1\ \ /\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0$$

Operaciones en coma flotante: Multiplicación

Supongamos dos números A y B representados en coma flotante. En esa representación tendremos que:

$$A = mA * 2^{eA} \qquad B = mB * 2^{eB}$$

Si lo que queremos hacer es multiplicar A y B, lo que haremos será:

$$A * B = (mA * 2^{eA}) * (mB * 2^{eB}) = (mA * mB) * 2^{eA+eB}$$

Ejemplo: Multiplicar los números 1'19 y 0'36 en coma flotante (8 bits para la mantisa en signo y magnitud, y 6 para el exponente sesgado)

$A = 1'19)_{10} = 1'0011)_2 = 0'1001100... * 2^1$

$$A = \underset{s}{0}\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ /\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1$$

$B = 0'36)_{10} = 0'01011100001)_2 = 0'1011100... * 2^{-1}$

$$B = \underset{s}{0}\ \ 1\ \ 0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ /\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1$$

```
  0'1 0 0 1 1
x 0'1 0 1 1 1
    1 0 0 1 1
  1 0 0 1 1
```

```
        1 0 0 1 1
    1 0 0 1 1 0
-------------------
0'0 1 1 0 1 1 0 1 0 1
```

$A * B = 0'0110110101 * 2^{-1+1} = (0'10110101 * 2^{-1}) * 2^{0} = 0'10110101 * 2^{-1}$

$A * B = \underset{s}{0}\ \underline{1\ 0\ 1\ 1\ 0\ 1\ 0}\ /\ \underline{1\ 0\ 0\ 0\ 1}$

Operaciones en coma flotante: Suma

La suma es un poco más complicada. La manera de sumar valores que están multiplicados por un número es buscar que el número que multiplica sea el mismo para poder sacarlo factor común y luego sumar lo que nos queda:

$A = mA * 2^{eA} \qquad B = mB * 2^{eB}$

Si **eA > eB** entonces significa que **eB = eA – eaux**

$$A + B = (mA * 2^{eA}) + (mB * 2^{eB}) = (mA * 2^{eA}) + (mB * 2^{eA-eaux}) =$$
$$= (mA + mB * 2^{-eaux}) * 2^{eA}$$

Ejemplo: Sumar los números 1'19 y 0'36 en coma flotante (8 bits para la mantisa en signo y magnitud, y 6 para el exponente sesgado)

$A = 1'19)_{10} = 1'0011)_2 = 0'1001100... * 2^{1}$

$B = 0'36)_{10} = 0'01011100001)_2 = 0'1011100... * 2^{-1} = (0{,}1011100 * 2^{-1} * 2^{+1}) * 2^{-1}$

```
  0'1 0 0 1 1
+ 0'1 0 1 1 1
-------------
  1'0 1 0 1 0
```

$A * B = 1'01010 * 2^{-1} = (0'101010 * 2^{1}) * 2^{-1} = 0'10110101 * 2^{0}$

$A * B = \underset{s}{0}\ \underline{1\ 0\ 1\ 0\ 1\ 0\ 1}\ /\ \underline{1\ 0\ 0\ 0\ 0}$

SEGUNDA PARTE: TIPOS COMPLEJOS

Introducción

En la primera parte del tema se vieron los tipos simples de datos que es capaz de almacenar el ordenador: booleanos, caracteres, enteros y reales.

Sin embargo, en la mayoría de los casos la información tratada por el ordenador irá agrupada de una forma más o menos coherente en estructuras complejas compuestas por datos simples. Las estructuras de datos se distinguen por los elementos que las componen y por las operaciones que se pueden realizar con ellas o entre sus elementos.

En ocasiones interesará manejar las estructuras como elementos únicos, pero otras veces nos interesará manejar los elementos que componen las estructuras como elementos separados.

Clasificaciones de las agrupaciones de datos

Los datos complejos o *agrupados* pueden ser clasificados según varios criterios:

Agrupaciones contiguas y agrupaciones enlazadas

Las agrupaciones contiguas son aquellas que ocupan posiciones sucesivas o contiguas de bits en memoria. La posición de un elemento dentro de la agrupación se puede calcular sumando al comienzo de la agrupación una cantidad determinada.

Las agrupaciones enlazadas son aquellas que tienen la información dispersa en la memoria, y la manera de acceder desde un elemento a otro es mediante la dirección en memoria dónde está el siguiente elemento.

Agrupaciones homogéneas y agrupaciones heterogéneas

Las agrupaciones homogéneas son aquellas que guardan grupos de elementos iguales (todos los elementos guardados son enteros, reales,...)

Las agrupaciones heterogéneas son las que agrupan elementos (campos) de diferentes tipos de datos.

Agrupaciones estáticas y agrupaciones dinámicas

Las agrupaciones estáticas son las que guardan un número predeterminado de elementos, que no puede ser modificado en el momento en que se ejecuta el programa. Hay que elegir el número cuando escribimos el programa, antes de compilarlo.

Las agrupaciones dinámicas son aquellas que pueden ir modificando el número de elementos que contienen a medida que va siendo necesario guardar más información.

En este tema veremos diferentes agrupaciones de datos, desde las más simples a las más complejas: *arrays* (vectores o *arrays* unidimensionales, matrices o *arrays* multidimensionales y cadenas), registros o estructuras y listas dinámicas.

Las agrupaciones, en general, no son exclusivas de manera que podremos encontrar combinaciones de ellas si es necesario.

Matrices o arrays

Los *arrays* son agrupaciones de datos homogéneas, contiguas y estáticas. Los elementos que contienen son todos del mismo tipo. El número de elementos que podemos guardar se define inicialmente cuando escribimos el programa.

Podemos distinguir entre los siguientes tipos de *arrays*:

Vectores (o *arrays* unidimensionales)

Son agrupaciones en las que cada elemento tiene asociado un índice (un entero), de manera que se puede acceder a cada uno de los elementos mediante la utilización de ese índice.

Por ejemplo, la declaración de un vector en C se realiza de la siguiente manera:

```
Tipo Nombre[Tamaño];
```

Donde **`Tipo`** es el tipo de los datos guardados en el vector, **`Nombre`** el nombre que le vamos a dar al vector y **`Tamaño`** el número de elementos que es capaz de guardar el vector.

Es importante destacar que en C los índices comienzan en **`0`** y terminan en **`Tamaño – 1`**.

Ejemplo:

Un vector capaz de guardar 10 enteros, que se llamase *vect* se declararía:

```
int vect[10];
```

Una cosa que debe quedar clara en la declaración, es que sólo se reserva espacio. NO se pone ningún valor en el vector.

Operaciones:

Asignación: Se da un valor determinado a algún elemento del vector. La manera de hacerlo es mediante el nombre del vector y el índice que queremos asignar:

```
Nombre[indice] = Valor;
```

Ejemplo:

Para asignar el valor 4 al índice (o posición del vector) ocho:

```
vect [8] = 4;
```

Recorrido: Se pasa por los elementos del vector para realizar una tarea concreta en cada uno de ellos.

Recorrido completo

```
Desde i ← 0 hasta Tamaño hacer
  Procesar (Nombre [i])
Fin_desde
```

Ejemplo:

El recorrido completo del vector sería, en C:

```
for (i = 0; i < 10; i++)
   Procesar (vect[i]);
```

Recorrido parcial

```
i ← 0
Mientras (Nombre[i] cumpla una cierta condición)
  Procesar (Nombre [i])
Fin_desde
```

Inicialización: Para hacer la inicialización de todos y cada uno de los elementos del vector, habría que recorrerlo y asignar o pedir al usuario que introdujese cada uno de los elementos.

Ejemplo:

Supongamos que queremos pedir al usuario que nos dé todos los valores del vector:

```
for (i = 0; i < 10; i++)
```

```
   scanf ("%d", &vect[i]);
```

Supongamos que queremos poner todos los elementos del vector a cero:

```
for (i = 0; i < 10; i++)
   vect[i] = 0;
```

Ejemplo 1: Realizar una función que rellene los elementos de un vector.

```
#define TAM 100
void RellenarVector (int vect[TAM])
{
   int i;

   for (i = 0; i < TAM; i++)
      scanf ("%d", &vect[i]);
}
```

Es importante destacar respecto de los vectores que cuando van a ser modificados (paso de parámetros por referencia) no se realiza nada especial, por el hecho de que en C SIEMPRE se pasan por referencia.

Ejemplo 2: Realizar una función que calcule la media de los elementos contenidos en un vector.

```
#define TAM 100

float MediaVector (int vect[TAM])
{
   int i;
   int acc = 0;

   for (i = 0; i < TAM; i++)
      acc = acc + vect[i];

   return ( (float)acc / TAM);
}
```

Búsqueda en vectores: Para buscar un cierto elemento en un vector se deben recorrer todos y cada uno de los elementos y comparar hasta encontrar el que se busca. A este proceso se le llama búsqueda secuencial.

Ejemplo: Realizar una función que diga si un cierto valor 'x' se encuentra o no en un vector 'v'.

```
int Buscar1 (int vect[TAM], int x)
{
   int i, encontrado = 0;

   for (i = 0; i < TAM; i++)
      if (v[i] == x)
         encontrado = 1;

   return (encontrado);
}
```

De todas formas este procedimiento puede mejorarse si cuando encontramos el elemento salimos. Es más, podemos incluir la comparación en la condición del bucle:

```
int Buscar2 (int vect[TAM], int x)
{
```

```
   int i;

   i = 0;
   while ( (i < TAM) && (v[i] == x) )
         i++;

   if (i == TAM)
      return (0);
   else
      return (1);
}
```

Podemos evitar la comparación de '`i < TAM`' si estamos seguros de que el elemento '`x`' está en el vector. La mejor manera de estar seguros es si lo ponemos al final del vector. A esta forma de buscar se le llama búsqueda secuencial con centinela.

```
int Buscar3 (int vect[TAM + 1], int x)
{
   int i;

   vect[TAM] = x;

   i = 0;
   while (v[i] == x)
         i++;

   if (i == TAM)
      return (0);
   else
      return (1);
}
```

Si el vector estuviese ordenado se puede aplicar una técnica especial de búsqueda llamada búsqueda binaria o dicotómica, según el siguiente algoritmo:

1. Se mira el elemento central
2. Si es el elemento buscado terminar
3. Si no lo es:
   3.a. Determinar en que 'zona' del vector está (por arriba o por debajo del central)
   3.b. Repetir el proceso en la nueva zona.

```
int Buscar4 (int vect[TAM], int x)
{
   int primero, ultimo, central;

   primero = 0;
   ultimo = TAM - 1;
   central = (primero + ultimo) / 2

   while ( (primero < ultimo) && (v[central] ¡= x) )
   {
      if (x < v[central])
         ultima = central - 1;
      else
         primero = central + 1;
   }

   if (x == v[central])
      return (1);
   else
      return (0);
}
```

Cadenas (o *strings*)

Hablaremos de cadenas como un caso especial de vectores en el que guardamos caracteres. Una cadena en C, no es más que un vector que contiene caracteres, pero que tiene como característica especial que sólo se utiliza una parte del vector, de manera que se coloca un delimitador al final de los caracteres utilizados, que en C es el carácter con valor 0 (habitualmente indicado como '\0').

Por ejemplo: Supongamos una cadena de 10 caracteres, en la que deseamos guardar la palabra 'hola'. De los diez posibles caracteres, sólo vamos a utilizar cuatro. El contenido del vector será pues:

0	1	2	3	4	5	6	7	8	9
'h'	'o'	'l'	'a'	'\0'	?	?	?	?	?

La declaración de una cadena se realizará como la de un vector en el que vayamos a guardar caracteres:

```
char Nombre [Tamaño];
```

Ejemplo:

Una cadena capaz de guardar 9 caracteres válidos:

```
char cad [10];
```

Se debe recordar que hay que tener en cuenta que el delimitador ocupa una posición.

Operaciones sobre cadenas

Asignación: Guarda una cadena en una variable de tipo cadena.

```
char cad[20];
strcpy (cad, "hola" ); // sprintf (cad, "valor = %d", j);
```

Concatenar cadenas

```
strcat (cad1, cad2);
```

Longitud de una cadena

```
strlen (cad);
```

Comparación de cadenas

```
strcmp (cad1, cad2);
```

La función devolverá cero si las cadenas son iguales, un valor positivo si la primera cadena precede a la segunda ortográficamente o un valor negativo en cualquier otro caso.

Recorrido de los elementos de una cadena

El recorrido de los elementos de una cadena es similar al recorrido de los elementos de un vector, pero teniendo en cuenta que no todos los elementos son válidos, sólo hasta el número de elementos utilizado.

```
for (i = 0; i < strlen (cad); i++)
   Analizar (cad[i]);
```

Matrices (o *arrays* bidimensionales)

Son agrupaciones similares a los vectores pero en las que cada elemento tiene asociados dos índices enteros, de manera que se puede acceder a cada uno de los elementos mediante la utilización de esos índices.

La declaración de una matriz en C se realiza de la siguiente manera:

```
Tipo Nombre[Tamaño1][Tamaño2];
```

Donde **Tipo** es el tipo de los datos guardados en la matriz, **Nombre** el nombre que le vamos a dar a la matriz y **Tamaño1** y **Tamaño2** el número de filas y columnas que tiene la matriz.

Operaciones:

Asignación

Se da un valor determinado a algún elemento de la matriz. La manera de hacerlo es mediante el nombre de la matriz y los índices que queremos asignar:

```
Nombre [indice1][indice2] = Valor;
```

Ejemplo:

Para asignar el valor 4 a los índices (o posición de la matriz) ocho, cinco:

```
mat[8][5] = 4;
```

Recorrido

Se pasa por los elementos de la matriz para realizar una tarea concreta en cada elemento.

Recorrido completo
Desde i ← 0 hasta Tamaño1 hacer
Desde j ← 0 hasta Tamaño2 hacer
Procesar (Nombre [i][j])
Fin_desde
Fin_desde

Ejemplo:

El recorrido completo de una matriz de diez por veinte elementos sería en C:

```
int mat[10][20];

for (i = 0; i < 10; i++)
   for (j = 0; j < 20; j++)
      Procesar (mat[i][j]);
```

Inicialización

Para hacer la inicialización de todos y cada uno de los elementos de la matriz, habría que recorrerla y asignar o pedir al usuario que introdujese los valores para cada uno de los elementos.

Ejemplo:

Supongamos que queremos pedir al usuario que nos dé todos los valores de la matriz:

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 20; j++)
      scanf ("%d", &mat[i][j]);
```

Supongamos que queremos poner todos los elementos del vector a cero:

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 20; j++)
      mat[i][j] = 0;
```

El resto de operaciones serían similares a las vistas en vectores, pero siempre teniendo en cuenta que el acceso a cada uno de los elementos de la matriz se realiza mediante la utilización de dos índices.

Arrays multidimensionales

Son agrupaciones similares a los vectores o las matrices, pero en las que cada elemento tiene asociados más de dos índices enteros, de manera que se puede acceder a cada uno de los elementos mediante la utilización de esos índices.

La declaración de un *array* multidimensional se realiza en lenguaje C de la siguiente manera:

```
Tipo Nombre[Tamaño1][Tamaño2]...[Tamañon];
```

Donde **Tipo** es el tipo de los datos guardados en el *array* multidimensional, **Nombre** el nombre que le vamos a dar y **Tamaño1, Tamaño2,...** el tamaño de las diferentes dimensiones del *array*.

Las operaciones que se puedan definir serán similares a las vistas en vectores o matrices pero ampliando el número de índices de forma adecuada, y recordando que para acceder a un elemento concreto hay que determinar el valor de todos y cada uno de los índices.

Estructuras o registros

Las estructuras o registros son agrupaciones heterogéneas, contiguas y estáticas. Los elementos contenidos en la agrupación pueden ser de distintos tipos y se suelen llamar *campos de información.* Cada instancia de esta estructura estará relacionada con la información referente a un objeto concreto, como por ejemplo la información relacionada con una persona (Nombre, edad, D.N.I.,...) o la información relacionada con un libro (Título, autor, precio, disponible o agotado,...)

En C los registros o estructuras son nuevos tipos de datos que podemos utilizar a lo largo de nuestro programa. Es decir, una vez declarada la estructura podemos utilizar y declarar variables de ese nuevo tipo (igual que si declarásemos variables de tipo entero o real.)

La manera de declarar una estructura en C es la siguiente:

```
struct Nombre
{
   Tipo1 Campo1;
   Tipo2 Campo2;
   ...
};
```

Donde **Nombre** es el nombre que va a recibir nuestra estructura, **Tipo1** será el tipo de datos del primer campo de información, **Campo1** será el nombre del primer campo de información, **Tipo2** el tipo de información guardado en el segundo campo de información, **Campo2** el nombre del segundo campo de información, y así para todos los campos de información que queramos tener.

Ejemplo: Declarar un registro capaz de guardar la información de una persona

Pondremos como información de una persona: Su nombre, su edad, su D.N.I. y si es Hombre o Mujer.

```
struct persona
{
   char Nombre[50];
   int Edad;
   long int DNI;
   char Sexo;
};
```

Una vez declarado el nuevo tipo de dato, éste es similar a los tipos de datos simples vistos hasta este momento, y podemos declarar variables de este nuevo tipo. En C:

```
struct Nombre NomVar;
```

Donde '**struct Nombre**' es el nuevo tipo y **NomVar** el nombre de la variable de este tipo.

Esta variable aglutina diferentes campos. La manera de acceder a cada uno de los campos es mediante el operador de acceso '**.**'

```
NomVar.Campo1 / NomVar.Campo2 / ...
```

El tratamiento de cada uno de estos campos depende de su tipo: si es un entero lo asignaremos y trataremos como entero, si es un vector como un vector, si es una cadena como una cadena, etc.

Ejemplo: A partir de la estructura persona declarar una variable de este nuevo tipo

```
struct persona Per;
```

Ejemplo: Asignar los valores Nombre=Ricardo Edad=32 DNI=20200200 Sexo=H

```
struct persona Per;

printf ("Dame el nombre: ");
scanf ("%s", Per.Nombre);
/* Recordar que los vectores, y como consecuencia las cadenas siempre se
pasan por referencia en C, de manera que no hay que poner '&' */

printf ("Dame la edad: ");
scanf ("%d", &Per.Edad);

printf ("Dame el D.N.I.: "),
scanf ("%ld", &Per.DNI);

printf ("Dame el sexo: ");
scanf ("%c", &Per.Sexo);
```

Introducción a las estructuras dinámicas: Punteros

Existe en casi todos los lenguajes de programación un tipo de datos simple denominado *puntero*. Un puntero es un tipo de dato que es capaz de guardar una dirección de memoria. Las direcciones de memoria sirven al ordenador para saber en qué lugar de la memoria se sitúa exactamente una determinada información, por lo que cualquier variable lleva asociada una dirección de memoria.

Los operadores básicos con punteros en lenguaje C son:

- **`&`** Obtiene la dirección de memoria en donde se encuentra una variable, a partir del nombre de la variable.
- **`*`** Obtiene el contenido (valor) que se guarda en una determinada posición de memoria.
- **`malloc`** Reserva espacio en memoria para una determinada información y devuelve la dirección de memoria en donde se encuentra ese espacio.
- **`free`** Libera el espacio reservado para una información (le dice al ordenador que ese espacio puede ser utilizado por otros usuarios.)

Con este tipo de datos y los registros podemos construir agrupaciones de elementos que pueden ir creciendo en memoria a medida que va aumentando la información que deseamos guardar en el ordenador, de manera que no tenemos que fijar el número máximo que queremos guardar en la agrupación en el momento de escribir el programa.

En general podremos construir agrupaciones dinámicas incluyendo un puntero como campo de un registro, de manera que a través de este puntero puedo acceder a otros elementos de la agrupación.

Una definición de un registro de este tipo podría ser en C:

```
struct Nodo
{
   Valor Info;
   struct Nodo * Sig;
};
```

A estos registros que contienen un puntero se les suele llamar *nodos*, y son la base de las agrupaciones dinámicas, ya que gracias a estos punteros podemos enlazar todos los elementos de información que queramos (limitados tan solo por la capacidad física de la memoria del ordenador) y además gracias a la instrucción '**malloc**' podemos reservar memoria para nuevos elementos a medida que nos vaya haciendo falta.

La manera de acceder a los campos de un registro a partir de un puntero es mediante el operador 'contenido' "*****" y el operador de acceso "**.**"

```
struct Nodo * p_aux;

(*p_aux).Info / (*p_aux).Sig
```

Esta combinación es equivalente al operador de acceso en punteros '**->**' (un menos seguido de un mayor) de manera que quedaría:

```
p_aux->Info / p_aux->Sig
```

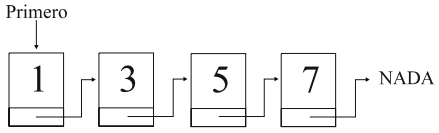
Listas enlazadas

Una lista es una agrupación de elementos entre los que existe uno que podemos llamar *primero* y a partir del cual se puede acceder al resto de los elementos uno a uno siguiendo los punteros que los enlazan.

Las listas enlazadas están definidas por:

1. Una estructura de tipo nodo.
2. Un puntero que nos marca el primer elemento, a partir del cual puede accederse a cualquier otro elemento de la agrupación.

La idea a la que se quiere llegar sería la siguiente: La lista de números 1, 3, 5, 7 quedaría en forma enlazada como sigue:



Primero sería el puntero que señalaría al primer elemento de la lista. Mediante el puntero situado en cada uno de los nodos es posible acceder al siguiente elemento desde uno cualquiera.

A '**NADA**' en C se le llama '**NULL**'

Operaciones

Creación

Cuando creamos una lista, en principio estará sin elementos. La creación de una lista llevará asociada la declaración del tipo *nodo* y una variable que sea capaz de guardar dónde está el primero de los elementos, así como la asignación de que no hay ningún elemento en la lista.

```
struct Nodo
{
   Valor Info;
   struct Nodo * Sig;
};
...
struct Nodo * Primero;
...
Primero = NULL;
```

La declaración del tipo Nodo se debe realizar al comenzar el programa, después de los "includes" y los "defines".

La variable de tipo puntero se declara en el lugar adecuado de la declaración de variables. Y la asignación en el punto del programa en el que deseemos iniciar la variable.

Búsqueda de un elemento

En el caso de listas dinámicas, sólo podremos realizar búsqueda secuencial, es decir, partiendo del primer elemento ir mirando si el que estamos buscando es o no el indicado.

```
int Buscar (struct Nodo * primero, int x)
{
   struct Nodo * p_aux;

   p_aux = primero;
   while ( (p_aux != NULL) && (p_aux->Info != x) )
      p_aux = p_aux->Sig

   if (p_aux == NULL)
      return (0);
   else
      return (1);
}
```

Inserción de un elemento delante del primero

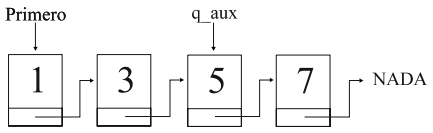
Supongamos que queremos insertar el elemento '**x**' delante del primero:

```
struct Nodo * p_aux;
...
/* Reservamos memoria para el nuevo elemento */
p_aux = malloc (sizeof (struct Nodo) );

/* Actualizamos su informacion */
p_aux->Info = x;

/* Enlazamos el elemento en la lista */
p_aux->Sig = primero;

/* El primero ahora es el que hemos insertado */
primero = p_aux;
```

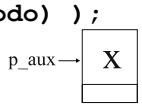
Inserción de un elemento detrás de uno dado

Supongamos que queremos insertar el elemento '**x**' detrás de uno marcado con '**q_aux**':

```
void InsertarDetras (struct Nodo * primero, struct Nodo * q_aux, Valor x)
{
   struct Nodo * p_aux;
```

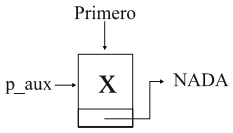



```
   p_aux = malloc (sizeof (struct Nodo) );
   p_aux->Info = x;
```

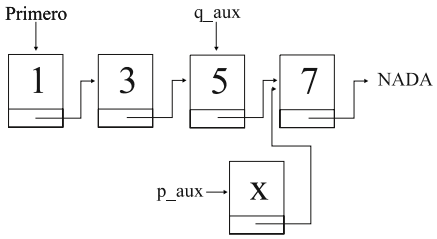



```
   /* Si no habia elementos solo tendremos el que acabamos de crear */
   if (primero == NULL)
   {
```

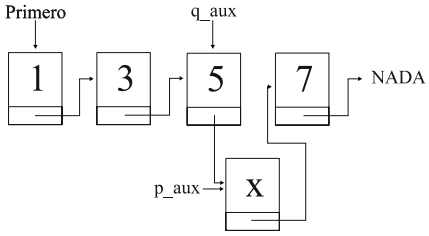
```
            p_aux->Sig = NULL;
            primero = p_aux;
```




```
         }
         else
         {
            p_aux->Sig = q_aux->Sig;
```




```
            q_aux->Sig = p_aux;
```

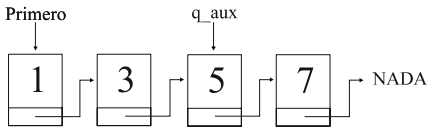



```
         }
      }
```

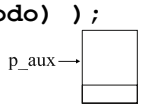
Inserción de un elemento delante de uno dado

Supongamos que queremos insertar el elemento '**x**' delante de uno marcado con '**q_aux**':

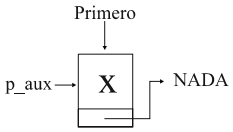
```
      void InsertarDetras (struct Nodo * primero, struct Nodo * q_aux, Valor x)
      {
         struct Nodo * p_aux;
```



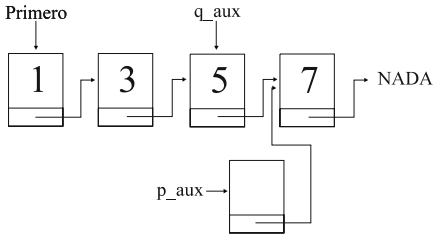

```
         p_aux = malloc (sizeof (struct Nodo) );
```



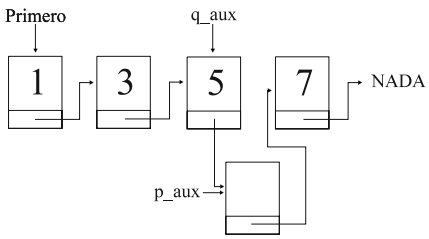

```
         /* Si no había elementos sólo tendremos el que acabamos de crear */
         if (primero == NULL)
         {
            P_aux->Info = x
            p_aux->Sig = NULL;
            primero = p_aux;
```



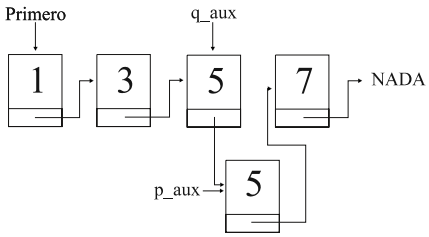

```
         }
         else
         {
            p_aux->Sig = q_aux->Sig;
```



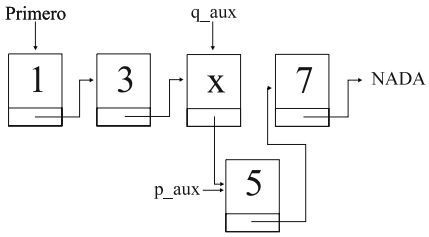
```
q_aux->Sig = q_aux;
```




```
p_aux->Info = q_aux->Info;
```




```
q_aux->Info = x;
```



Ventajas e inconvenientes de las agrupaciones estáticas frente a las agrupaciones dinámicas:

- Mayor facilidad para insertar y eliminar elementos en las agrupaciones dinámicas.
- Más fácil y rápido realizar búsquedas en vectores.
- Si el número de elementos es conocido o no va a variar mucho serán más adecuadas las agrupaciones estáticas
- Si el número de elementos va a ser cualquiera serán más adecuadas las agrupaciones dinámicas.
- En general, para un mismo número de elementos las agrupaciones dinámicas ocuparán más memoria que las estáticas (por cada elemento se tiene también un puntero en las agrupaciones dinámicas.)