QBASIC I

EL QBASIC ES SENCILLO Y NO REQUIERE UNA ESTRUCTURA BASICA

Veamos en un momento como sería un programa en lenguaje PASCAL que haga lo mismo que el nuestro:

```
Program Hola;
uses crt;
begin
        clrscr;
        writeln("Hola Mundo");
end.
```

Y lo mismo en lenguaje C:

```
#include < stdio.h >
int main(void)
{
        clrscr();
        cprintf("Hola Mundo");
        return(0);
}
```

Podemos ver que en los dos están las instrucciones para borrar la pantalla (clrscr) y la de escribir (writeln en PASCAL y cprintf en C), pero ¿Por que llevan punto y coma al final? y ¿Qué son esos símbolos y esas otras palabras tan raras?...
En otros lenguajes de programación hace falta escribir un "esqueleto básico" donde montar las instrucciones, en QBasic no. Nos limitamos a escribir lo que queremos que haga el programa y ya está.
Estas estructuras básicas son necesarias en los otros lenguajes más avanzados, pero en QBasic como sólo vamos a aprender así lo tenemos más fácil sin tener que escribir cosas que todavía no entendemos.

ESCRIBIR COMENTARIOS

Se usan comentarios dentro del listado del programa para explicar su funcionamiento. Lo que pongamos en los comentarios no altera para nada el funcionamiento del programa.
Los comentarios Ayudan a documentar el programa, sobre todo cuando son programas largos
En QBasic para escribir un comentario ponemos un apóstrofo ('). Veamos el ejemplo con comentarios:

```
'       Nombre: PROGRAMA HOLA MUNDO
'       Sirve para: Escribe en la pantalla "Hola Mundo"
'       Hecho por : Kike Barrueto  2005 Agosto 10
'
CLS                                 'Borra la pantalla
PRINT "Hola Mundo"                  'Escribe "Hola mundo"
```

Una cabecera describiendo el programa y a continuación a la derecha de algunas instrucciones, en este caso de todas, hemos explicado para que sirven.
En los programas ya terminados es conveniente poner la cabecera siempre y explicar las instrucciones más complicadas.
El formato de la cabecera vendrá especificado en el proyecto, por ejemplo que contenga el nombre del programa, una descripción corta, el autor y la fecha. En nuestros programas no es necesario ser tan estrictos.
Una Idea: poner comentarios para que un trozo de programa no se ejecute.

```
'       Cls
'       Print "Hola Mundo" este programa no hace nada
```

VARIABLES

Vamos a escribir un programa un poco más complicado que nos pregunte nuestro nombre y nos salude. El resultado (lo que se ve en pantalla al ejecutarlo) podría ser el siguiente:

¿Cómo te llamas? Kike
Hola Kike
El programa borrará la pantalla luego nos preguntará por nuestro nombre, nos dejará escribirlo con el teclado y cuando lo hagamos lo guardará en "algún sitio", después escribirá "Hola " seguido de nuestro nombre que lo habrá sacado del "sitio" donde lo guardó antes.
Vamos a ver como será el código:

```
CLS
INPUT "¿Cómo te llamas? ", nombre$
PRINT "hola "; nombre$
```

Ahora tenemos una instrucción nueva, INPUT, que lleva dos parámetros separados por una coma. Esta instrucción lo que hace es dejar al usuario del programa que escriba algo con el teclado y cuando pulse la tecla Enter lo guarda en memoria.
La primera parte "¿Cómo te llamas?" la escribe en pantalla tal como está para que el usuario sepa lo que se le pregunta.
La segunda parte es una Variable, es decir el nombre que le hemos puesta a un "trozo" de la memoria del ordenador dónde vamos a guardar lo que ha escrito el usuario y que le hemos llamado nombre$ para después poder localizarlo.
La siguiente instrucción, PRINT, es la misma que en el anterior programa. Lo que hace es escribir lo que lleva detrás. En esta ocasión va a escribir dos cosas, primero "Hola " con su espacio detrás tal como aparece dentro de las comillas, y a continuación todo seguido escribirá el valor que haya guardado en la variable nombre$. No escribirá Hola nombre$ porque nombre$ no está entre comillas.
Esto de las variables es una de las cosas más importantes de los lenguajes de programación. Los programas van a manejar datos que estarán grabados en la memoria del ordenador. Si no existieran las variables tendríamos que saber en que posición exacta los guardamos y después para recuperarlos habría que escribir los números de las direcciones de memoria y saber exactamente cuanto largo son los datos. Esto es muy complicado (Esto es lo que se hace en los Lenguajes Ensambladores), para hacerlo aquí más sencillo usaremos los nombres de variables y QBasic se encargará de ver en que lugar físico de la memoria mete la información y después de saber recuperarla cuando se la pidamos.

TIPOS DE DATOS

Nuestros programas van a poder trabajar con varios tipos de información, como son letras, números con o sin decimales, etc. Nuestro programa anterior usaba una variable de texto o de cadena de caracteres para almacenar nuestro nombre, pero en otros casos habrá que almacenar números más o menos grandes, por ejemplo.
Veamos ahora los tipos de datos que hay en QBasic.

NOMBRE	SUFIJO	DESCRIPCIÓN	MÍNIMO Y MÁXIMO
STRING	**$**	**Texto con cualquier carácter**	**0 a 32767 caracteres**
INTEGER	**%**	**Números enteros**	**-32768 a 32767**
LONG	**&**	**Números enteros más grandes**	**2.147.483.647 a -2.147.483.648**
SINGLE	**!**	**Números con decimales**	**$2{,}8x10^{45}$ a $-2{,}8x10^{45}$**
DOUBLE	**#**	**Números con más decimales**	**$4{,}9x10^{324}$ a $-4{,}9x10^{324}$**

Vamos ahora a ver lo que significa esta tabla.
En QBasic las variables van a ser de uno de estos cinco tipos.
Si queremos almacenar palabras, nombres, fechas, horas, códigos postales, números de teléfono, DNIs, matrículas, etc. tendremos que usar una variable de tipo Cadena (STRING) cuyo nombre llevará al final el carácter $.
Para guardar números tendremos que usar alguno de los otros cuatro tipos de variables.
Si nuestros números no van a llevar decimales usaremos una variable de tipo entero (INTEGER) que ocupa poca memoria (2 bytes) y además el ordenador trabaja con ellas muy rápido.
Si los números enteros van a llegar a ser mayores de 32.767 tendremos que utilizar una variable de tipo Entero largo (LONG) en la que "caben" hasta números mayores que dos mil millones.

Si vamos a usar números reales (que pueden llevar decimales) tendremos que usar variables de precisión sencilla (SINGLE).
Para algún cálculo que necesite obtener resultados con muchos decimales usaremos las variables de doble precisión (DOUBLE) que son más lentas de manejar y ocupan más memoria que las anteriores.
Si intentamos guardar en una variable un valor más grande (o más pequeño si es negativo) de lo que permite su tipo de datos, se producirá un "Error de Tiempo de Ejecución" y el programa se parará.

NOMBRES DE VARIABLES

Cada variable llevará un nombre que nosotros decidamos para usarla en nuestro programa. Tenemos que elegir nombres que no sean demasiado largos y que tengan algo que ver con lo que va a contener la variable.

Los nombres de variables deben cumplir estas condiciones:

- Deben tener entre 1 y 40 caracteres.
- Pueden incluir letras y números.
- No pueden llevar la ñ ni letras acentuadas.
- El primer carácter tiene que ser una letra.

Estos nombres de variable son válidos.

- Edad
- fechaNacimiento
- jugador1
- iva16porCiento
- vidasRestantes

Estos otros nombres no son válidos

- 2jugadores (No empieza por una letra)
- año (Contiene la ñ)
- elPatioDeMiCasaEsParticularYCuandoLlueveSeMoja (Muy Largo)
- puntuación (Lleva acento)

Hay que tener en cuenta que QBasic no distingue de mayúsculas y minúsculas, por lo tanto una variable que se llame nivel será la misma que Nivel, NIVEL o niVEL. De hecho cuando cambiemos las mayúsculas y minúsculas de una variable se cambiarán todas las que ya haya escritas en el resto del programa, es decir, si hemos escrito en un sitio puntos y después escribimos PuntoS en otro sitio, la anterior también se cambiará a PuntoS.
Por convenio los nombres de las variables se escriben todo en minúsculas para diferenciarlos visualmente de las Palabras Claves que se van poniendo en mayúsculas automáticamente. Si una variable lleva dos palabras se suele escribir la primera letra de la segunda palabra en mayúsculas para separar un poco, por ejemplo nombreJugador totalActivo o nuevoRegistro.
En las variables que deberían llevar la “ñ” hay varias formas: Para año se suele escribir cosas como ano, anno o anyo. De los acentos pasamos. En Visual Basic se pueden usar nombres de variables con ñ y acentos, pero en QBasic no.

DECLARACIÓN DE VARIABLES

En otros lenguajes de programación (la mayoría) hay que Declarar las variables al principio del programa. Esto significa escribir una instrucción que diga algo así como "Voy a usar una variable que se va a llamar nombre y va a ser de tipo cadena" y así con cada una de las variables que vayamos a usar en el programa.
En QBasic NO hace falta declarar las variables.
La primera vez que escribimos el nombre de una variable QBasic reserva en memoria el espacio para utilizarla y ya está disponible durante el resto del programa, pero ¿Cómo sabe QBasic del tipo que queremos que sea la variable?
En la tabla de tipos de datos había una columna que ponía Sufijo y que para cada tipo de datos tenía un símbolo distinto.

NOMBRE	SUFIJO
STRING	**$**
INTEGER	**%**
LONG	**&**
SINGLE	**!**
DOUBLE	**#**

Para indicar el tipo escribiremos uno de estos sufijos detrás del nombre de la variable, por ejemplo si nombre va a contener texto escribiremos nombre$ (Como hicimos en el ejemplo). A esto se llama declaración implícita, ya que dentro del nombre va incluído el tipo de datos.
Ahora ¿Que pasa si escribimos en distintos sitios el mismo nombre de variable con distintos sufijos?, por ejemplo:

```
variable$ = "Esto es un texto"
variable% = 2000
PRINT variable$
PRINT variable%
```

Lo que ocurrirá es que QBasic entiende que son dos variables completamente distintas, cada una de un tipo y nos da como resultado:

```
Esto es un texto
2000
```

Esto puede producir un Error de Lógica, es decir, el programa funciona perfectamente, pero a lo mejor no hace lo que esperábamos. Para evitar esto hay que poner a las variables nombres más descriptivos, y si en este caso de verdad queríamos dos variables es mejor ponerles nombres distintos para evitar confusiones.
En la práctica a partir de ahora vamos a poner el sufijo $ a las variables de cadena y las demás las dejamos sin sufijo para que sean reales que admiten números muy grandes con o sin decimales. Si fuéramos a hacer programas muy depurados sería conveniente usar los otros tipos de variables cuando fuera necesario para que fueran más eficientes con el consumo de memoria.
Las variables al ser usadas la primera vez antes de asignarles (guardar en ellas) ningún valor contienen un texto de longitud cero si son de cadena o el número 0 si son de otro tipo. En otros lenguajes de programación esto no es así y contienen cualquier valor que hubiera en esa posición de memoria de otros programas que la hubieran usado antes, por lo que se pueden producir errores. Aquí no.
Lo último que queda es saber que el Tiempo de Vida de una variable es desde que se usa por primera vez (desde que se declara o se ejecuta la línea donde su nombre aparece por primera vez) hasta que el programa termina, después su valor se pierde. No podemos contar con que tengamos valores en memoria guardados de la vez anterior que ejecutamos el programa.

LEER DESDE EL TECLADO (INPUT)

La orden que usaremos en QBasic para leer lo que el usuario escribe en el teclado es INPUT. Ya la hemos usado, pero vamos a verla con más tranquilidad.
Su sintaxis más común sería:

```
INPUT "Pregunta al usuario", variable
```

- "Pregunta al usuario" es una cadena de texto entre comillas que aparecerá en pantalla para que el usuario la vea y sepa lo que se le está preguntando. No puede ser una variable ni una expresión, solamente una cadena entre comillas. En otros lenguajes de programación no se da esta posibilidad y hay que escribir un rótulo justo antes usando otra instrucción.
- variable es el nombre de la variable dónde se almacenará la información que escriba el usuario.

Al ejecutar esta instrucción se escribe en la pantalla el texto que hay entre comillas y a la derecha aparece el cursor intermitente para que el usuario escriba. El programa estará parado

hasta que el usuario escriba lo que quiera y pulse la tecla "Enter". Entonces se almacena en la variable lo que ha escrito el usuario (El valor anterior que pudiera tener la variable se pierde) y el programa sigue con la siguiente instrucción que haya.
El usuario podrá desplazarse con las flechas del teclado por el texto que está escribiendo para corregir algún error antes de pulsar "Enter"
Para que lo que escriba el usuario no salga justo pegado al texto de la pregunta lo que se suele hacer es escribir un espacio en blanco al final dentro de las comillas.
Si escribimos...

INPUT "¿Cómo te llamas?",variable$

El resultado después de que el usuario termine de escribir sería

¿Cómo te llamas?Kike

Pero si lo ponemos con el espacio al final...
INPUT "¿Cómo te llamas? ",variable$

El resultado es parecido pero se ve mejor
¿Cómo te llamas? Kike

La variable tiene que ser del tipo correcto, si le pedimos al usuario que escriba una palabra, la variable debe de ser de tipo cadena y si no tendrá que ser de tipo numérico con o sin decimales.
Si solo ponemos la variable, sin pregunta al usuario, aparecerá un interrogante. Es decir, si ponemos...

INPUT nombre_variable
Aparecerá en la pantalla
?_

Para que no aparezca indicación alguna, solo el cursor, hay que poner las comillas vacías, por ejemplo...

INPUT "", nombre_variable

Esto puede ser útil cuando la pregunta que hacemos al usuario incluye algún cálculo o expresión de cadena. Como INPUT no permite evaluar expresiones en la pregunta al usuario, esta pregunta habrá que mostrarla antes en una instrucción PRINT normal de las que veremos más adelante con su punto y coma al final, y a continuación poner el INPUT de esta forma para leer la información del usuario.

ESCRIBIR EN PANTALLA - PRINT

Como ya hemos dicho, la forma más básica de dar al usuario los resultados de nuestro programa es a través de la pantalla.
En temas posteriores se explica como conseguir crear las pantallas de los programas para que cada cosa aparezca en un sitio determinado y con distintos colores, recuadros, etc, pero por ahora para aprender a programar nos limitaremos a escribir cosas una debajo de otra como en MS-DOS.
Usaremos la pantalla de texto de forma que cuando lleguemos a escribir en la línea más baja de la pantalla todo subirá hacia arriba y desaparecerá lo que hubiera en la primera línea.
Para aclarar la pantalla y no liarnos con lo anterior hemos puesto en todos los ejemplos CLS como primera instrucción para que se borre de la pantalla lo que hubiera de ejecuciones anteriores del programa actual o de otro, y se empiece a escribir en la parte superior.
Ahora vamos a ver con más detalle la instrucción PRINT que es la que usamos parara escribir en la pantalla.
Una aclaración. Si a alguien le parece que PRINT significa imprimir tiene razón, en informática a escribir en la pantalla del ordenador también se le puede llamar imprimir en pantalla. Además esta orden PRINT también se utiliza para imprimir como se verá más adelante.

La sintaxis de la instrucción es:
PRINT texto
Donde texto puede ser una cadena de caracteres entre comillas, que se escribirá tal cual.

```
PRINT "Esto es un texto"
```

O el nombre de una variable.

```
mensaje$ = "Prueba superada"
PRINT mensaje$
```

En este caso se escribirá el valor de que tenga la variable, en nuestro ejemplo se escribiría Prueba Superada.
La variable también puede ser de tipo numérico...

```
PRINT total%
```

También podemos escribir el resultado de operaciones matemáticas...

```
PRINT 2+2
```

En este caso se escribirá 4. Las operaciones matemáticas (Expresiones) las veremos con detalle en el tema siguiente.
Después de escribir algo el cursor (invisible) pasa a la línea de abajo, por ejemplo

```
PRINT "Uno"
PRINT "Dos"
PRINT "Tres"
```

Escribiría...

```
Uno
Dos
Tres
```

Pero en algún caso nos puede interesar que no sea así y que se escriba lo siguiente a continuación en la misma línea. Para hacerlo no tenemos más que escribir un punto y coma al final de la instrucción sobre la que queremos seguir, por ejemplo:

```
PRINT "Uno";
PRINT "Dos";
PRINT "Tres"
PRINT "Cuatro"
```

Escribiría...

```
UnoDosTres
Cuatro
```

Normalmente en una instrucción PRINT se suelen escribir varias cosas, como vimos en el programa saludador que primero escribía la palabra Hola y después nuestro nombre que estaba almacenado en la variable nombre$. Para hacer esto no hay más que separar con punto y coma (;) las distintas "cosas" que queremos escribir, por ejemplo:

```
nombrePrograma = "Super Juego"
nombreUsuario = "KB"
PRINT "Hola "; nombreUsuario; ", bienvenido a "; nombrePrograma
```

Escribiría...

Hola KB, bienvenido a Super Juego
Observa que los espacios entre palabras hay que ponerlos en algún sitio dentro de las comillas, de lo contrario saldría todo junto, incluso alguna vez es necesario hacer...

```
PRINT unaVariable$ ; " "; otraVariable$
```

...para que no salga todo junto.
Ahora vamos a probar con esto:

```
PRINT 2; 3; 4
```

Visto la anterior, el resultado tendría que ser...

234
pero no, es

2 3 4

QBasic escribe siempre los números con un espacio delante y otro detrás. Lo que ha escrito exactamente es: Espacio 2 Espacio Espacio 3 Espacio Espacio 4 Espacio, el último espacio no lo vemos pero también lo ha escrito.
Una consecuencia de esto será que los números por ahora nunca van a salir justo en la parte izquierda de la pantalla, sino una posición más adelante, por ejemplo...

```
PRINT "Prueba"
PRINT 25
PRINT 74
PRINT "Fin prueba"
```

Escribiría...

```
Prueba
 25
 74
Fin Prueba
```

Pero de esto no nos tenemos que preocupar. Si intentamos arreglarlo vamos a complicar nuestros programas innecesariamente. Ya habrá tiempo de dibujar pantallas en los temas de ampliación.
Una cosa que sí podemos hacer ya es usar lo que se llama "Posiciones de tabulación". Esto es que QBasic divide cada línea de la pantalla en posiciones que comienzan cada 14 caracteres, la primera en la posición 1, la segunda en la 14, la tercera en la 28, etc. Y si nosotros separamos las expresiones de la orden PRINT con comas en vez de puntos y comas los textos se escribirán en estas posiciones en vez de justo pegado a la anterior. Vamos con un ejemplo:

```
PRINT "Uno", "Dos", "Tres"
PRINT "Cuatro", "Cinco", "Seis"
PRINT "Siete", "Ocho", "Nueve"
```

Que escribiría en pantalla...

```
Uno       Dos       Tres
Cuatro    Cinco     Seis
Siete     Ocho      Nueve
```

Esto será muy útil en los listados que veremos más adelante. Por supuesto en una misma instrucción podemos separar unas cosas con comas y otras con puntos y comas, según haga falta.

Introducción a los operadores

En este tema vamos a ver como construir las expresiones matemáticas y lógicas necesarias para que nuestros programas sea capaces de hacer cálculos.
Una expresión sería lo equivalente a las fórmulas que escribimos en una hoja de cálculo (Excel), es decir una sucesión de números, operadores (signos más, menos, etc.) y nombres de variables, entre otras cosas, colocados en el orden correcto.

Operador de asignación

Lo primero que vamos a ver en este tema es un operador que nos permita guardar "algo" en una variable.
En el programa Saludador para guardar nuestro nombre en la variable nombre$ usábamos directamente la instrucción INPUT que se encargaba de leer los datos del teclado y guardarlos directamente en la variable. Aquí no empleábamos el operador de asignación, pero después hemos hecho cosas como:

```
mensaje$ = "Prueba superada"
```

o al declarar las constantes hacíamos:

```
CONST iva = 16
```

En estos ejemplos se puede ver que hemos usado el signo Igual para "Asignar" a la variable que hay a su izquierda el valor de la "Expresión" que hay a su derecha, por lo tanto ya podemos decir que el signo igual es el operador de asignación en el lenguaje Basic.
En el caso más sencillo:

```
total = 500
```

hacemos que en la variable total se almacene el número 500, perdiéndose el valor que tuviera anteriormente.
También podemos hacer:

```
total = total + 100
```

En este caso estamos usando la variable total como un "acumulador" ya que vamos a almacenar en ella el valor que tenga antes de la asignación más cien. Dicho de otra forma, si la variable valía 500 ahora le vamos a almacenar su valor de 500 más 100, con lo que al final de la asignación pasará a valer 600. En el siguiente apartado se explican algunas cosas sobre estas asignaciones.
Es importante tener claro que a una variable solo le podemos asignar los datos adecuados a su tipo, por ejemplo si a una variable de cadena le asignamos una expresión numérica o a una variable numérica le asignamos una cadena se producirá un Error de Tiempo de Ejecución y el programa se parará.
También hay que tener en cuenta que si a una variable de tipo entero le asignamos una expresión cuyo valor es con decimales, el número almacenado se redondeará, por ejemplo...

```
num% = 10 / 3
```

hará que num% valga 3 en vez de 3.3333, igual que

```
num% = 20 / 3
```

hará que num% valga 7 en vez de 6.6666. Esta vez se ha redondeado hacia arriba.
Tiene que quedar claro que la variable "destino" siempre va a la izquierda del signo igual, y la expresión a la derecha.
Las expresiones nunca van a ir solas. Siempre van en una asignación o en una estructura condicional que ya veremos más adelante.

Contadores y acumuladores

Estas dos palabras se usan mucho en programación para referirse a variables que van incrementando su valor a lo largo de la ejecución del programa.

Normalmente serán de tipo numérico y no se tratan de ninguna forma en especial, solo que al asignarles un valor se hace de forma que el anterior no se pierda, por ejemplo:

nivel = nivel + 1
total = total + subtotalLinea
vidas = vidas - 1
tamaño = tamaño * 2

Es muy importante inicializarlas de forma correcta siguiendo estas normas:

- Si vamos a sumar o restar la inicializaremos a cero al principio del programa para que no empiecen con valores residuales.
- Si vamos a multiplicar las inicializamos a 1, porque si valen cero todo lo que multipliquemos por ella seguirá valiendo cero.

La diferencia entre acumuladores y contadores es que los acumuladores se incrementan con cualquier número, como por ejemplo el total de una factura, mientras que los contadores se incrementan siempre con el mismo número, normalmente 1.

Operadores aritméticos

Llamamos operadores aritméticos a los signos que usaremos para hacer operaciones aritméticas como sumas, restas, etc.

OPERADOR	NOMBRE
+	**Suma**
-	**Resta**
*****	**Multiplicación**
/	**División**
MOD	**Resto de división (Módulo)**
^	**Potencias**

El operador suma lo que hace es sumar valores. Se pueden encadenar tantas sumas como queramos, por ejemplo

total = 2 + 74 + 7 + 25 + 82

El operador resta resta un valor a otro, por ejemplo:

neto = peso - tara

El operador asterisco hace multiplicaciones, por ejemplo:

elDoble = n * 2

El operador división divide un valor entre otro, por ejemplo:

kilos = gramos / 1000

El operador MOD nos devuelve el resto de una divisíón. Es especialmente útil para deducir si un número es par, ya que al dividir un par entre 2 el resto siempre es 0. Se usa entre los dos operandos igual que los símbolos anteriores, veremos su funcionamiento y su utilización en los temas siguientes.
El operador "acento circunflejo" sirve para calcular potencias, por ejemplo 5^2 es cinco al cuadrado o también 5*5. También podríamos calcular potencias decimales como 4^2.5. Para escribir este símbolo pulsa mayúsculas y la tecla correspondiente dos veces, oirás un pitido y solo entonces aparecerá escrito en la pantalla. Si aparece dos veces borra uno.

Operadores relacionales o de comparación

Estos operadores hacen que una expresión devuelva un valor lógico, es decir, en vez de un número devolverá VERDADERO o FALSO. Esto nos será muy útil en las estructuras condicionales que veremos en los siguientes temas, dónde veremos muchos ejemplos de su utilización.

OPERADOR	NOMBRE
=	Igual
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
<>	Distinto de

En QBasic el signo igual sirve tanto para asignaciones como para comparaciones, pero nunca se confunden ya que QBasic los evalúa de una u otra forma según dónde estén escritos.

Operadores lógicos

Estos operadores también devuelven un valor VERDADERO o FALSO a partir de los valores de las dos expresiones lógicas que unen. Igual que los anteriores se explicarán en los temas siguientes dedicados a estructuras condicionales.

OPERADOR	NOMBRE
AND	Operador Y
OR	Operador O
NOT	Operador monario de negación
XOR	Operador O exclusivo
EQV	Operador de equivalencia
IMP	Operador de implicación

Los tres primeros son las puertas lógicas elementales del álgebra de Boole, los otros se pueden construir a partir de las anteriores y por eso no suelen estar en otros lenguajes de programación y no hablaremos de ellos aquí.
AND devuelve verdadero si las dos expresiones que une son verdaderas, en caso contrario devuelve falso. Por ejemplo:

```
esNavidad = ((mes = 12) AND (dia = 25))
```

Podemos asegurar que es Navidad si el mes es 12 y el día es 25. Si el mes no es diciembre no será navidad aunque estemos a 25, tampoco si es un día de diciembre distinto de 25 y mucho menos si ni es diciembre ni es 25. Usamos por primera vez los paréntesis para dar más claridad, más adelante se explica cómo hacerlo.
OR devuelve verdadero si alguna de las dos expresiones que une es verdadera, o las dos lo son. Si las dos son falsas devuelve falso, por ejemplo:

```
puedesComprar = ((tuDinero > 0) OR (precio = 0))
```

En este caso la variable puedesComprar sería verdadero si tu dinero es mayor que cero (aprovechamos para dar un ejemplo del operador >) o si el precio es gratis, o las dos cosas. Solo sería falso si no tienes dinero y el producto a comprar vale dinero, con lo que las dos partes de la expresión serían falsas y el resultado también.
NOT es un operador "monario". Esto significa que sólo tiene un operando, a diferencia de los otros que siempre están entre dos operandos. Lo que hace NOT es invertir el resultado de una expresión, es decir, si es verdadera devuelve falso y si es falsa devuelve verdadero.
Para detallar los posibles valores que devuelven los operadores lógicos se construyen las llamadas "Tablas de la verdad" que representan todas las combinaciones posibles y los valores devueltos por cada operador.

TABLA DE LA VERDAD PARA AND				
VERDADERO	**AND**	**VERDADERO**	**=**	**VERDADERO**
VERDADERO	**AND**	**FALSO**	**=**	**FALSO**
FALSO	**AND**	**VERDADERO**	**=**	**FALSO**
FALSO	**AND**	**FALSO**	**=**	**FALSO**

TABLA DE LA VERDAD PARA OR				
VERDADERO	**OR**	**VERDADERO**	**=**	**VERDADERO**
VERDADERO	**OR**	**FALSO**	**=**	**VERDADERO**
FALSO	**OR**	**VERDADERO**	**=**	**VERDADERO**
FALSO	**OR**	**FALSO**	**=**	**FALSO**

TABLA DE LA VERDAD PARA NOT			
NOT	**VERDADERO**	**=**	**FALSO**
NOT	**FALSO**	**=**	**VERDADERO**

Observa que la variable puedesComprar y la esNavidad del ejemplo anterior pueden ser de cualquier tipo numérico para poder ser verdaderas o falsas. QBasic entiende que una variable (o el resultado de una expresión) es falsa si vale 0 y verdadera en cualquier otro caso. Otros lenguajes tienen un tipo de datos específico para estas situaciones, pero aquí puede valer cualquier tipo de datos numérico.
Es normal declarar constantes al principio de los programas para poder usar las palabras VERDADERO y FALSO en las expresiones y darle más claridad. Se haría:

```
CONST FALSO = 0
```

pero para verdadero podemos hacer...

```
CONST VERDADERO = NOT FALSO
```

con lo que hacemos que VERDADERO sea justo lo contrario de FALSO. Muy lógico.

FUNCIONES INCLUIDAS EN EL LENGUAJE

El lenguaje BASIC incluye un montón de funciones que nos harán algunos cálculos sin necesidad de tener que programar nosotros todo lo necesario.

Una función es una palabra que, insertada dentro de una expresión, llama a un pequeño "programita" que hace los cálculos y nos devuelve un resultado. Veamos este ejemplo:

```
CLS
INPUT "Escribe un número: ", num
raiz = SQR(num)
PRINT "Su raíz cuadrada es ";raiz
```

Este programa nos pide que escribamos un número y lo guarda en la variable num. A continuación calcula la raíz cuadrada del número usando la función SQR y gualda el resultado en la variable raiz para al final escribirlo en pantalla. No hemos tenido que programar ni conocer las operaciones mátemáticas necesarias para calcular la raíz cuadrada de un número, lo ha hecho automáticamente la función.

Observa la forma de decirle a la función cual es el número que queremos que utilice para calcular la raíz cuadrada: Lo metemos entre paréntesis después del nombre de la función. Este número se dice que es un Parámetro que pasamos a la función. Si alguna función no necesita parámetros no ponemos nada, en otros lenguajes hay que poner los paréntesis vacíos detrás del nombre de la función, pero en QBasic no.

El parámetro no tiene por que ser un número constante, puede ser a su vez otra expresión.

En la instrucción PRINT podemos incluir cualquier expresión, por lo tanto en el ejemplo anterior nos podíamos haber ahorrado la variable raiz escribiendo directamente:

```
CLS
INPUT "Escribe un número: ", num
PRINT "Su raíz cuadrada es "; SQR(num)
```

Las funciones tienen tipo de datos como las variables, es decir, nos devuelven un resultado que puede ser cadena, entero, etc. y por lo tanto pueden llevar un sufijo de identificación de tipo a continuación del nombre, por ejemplo la función TIMER nos devuelve el número de segundos que han pasado desde las doce de la noche en formato numérico:

```
segundos = TIMER
```

mientras que la función TIME$ nos devuelve la hora actual en formato de cadena de texto con la hora, los minutos y los segundos separados por dos puntos:

```
horaActual$ = TIME$
PRINT "Son las "; horaActual$
```

En este caso para guardar el valor devuelto por TIME$ hemos tenido que usar una variable de cadena. Ambas funciones no llevan parámetros porque si lo que hacemos es preguntar la hora no tenemos que decir nada más, ya la propia función verá como saca la hora del reloj interno del ordenador.

Si una función lleva varios parámetros se pondrán separados por comas dentro de los paréntesis, por ejemplo:

```
PRINT STRING$ (20,"Z")
```

La función STRING$ devolverá una cadena con veinte zetas, es decir "ZZZZZZZZZZZZZZZZZZZZ". Para hacerlo necesita saber dos cosas: cuantas y cual letra tiene que repetir, por eso le damos dos parámetros separados por una coma.

Observa también en el ejemplo que el primer parámetro debe de ser un número (o una expresión numérica) y el segundo tiene que ser una cadena (o una expresión de cadenas). Para saber de que tipo es cada parámetro y el tipo del resultado que devuelve la función es muy importante consultar la ayuda de QBasic todas las veces que haga falta. Si intentamos pasar parámetros de otro tipo se producirá un error de tipos y el programa se parará.

También se producirá un error de "Llamada a función no válida" si la función no es capaz de hacer los cálculos con los parámetros que le hemos pasado, aunque sean del tipo correcto. Por ejemplo, sabemos que no existe raíz cuadrada para los números menores que cero, por lo que se producirá un error de este tipo si hacemos

```
PRINT SQR (-14)
```

En caso de que pasemos como parámetro una expresión, por ejemplo

```
PRINT SQR(miVariable)
```

hay que tener mucho cuidado de que esta expresión no pueda llegar a valer menos que cero.

Estas son algunas de las funciones más usadas. En temas posteriores irán apareciendo más. Para verlas todas y todos los tipos de datos que necesitan y devuelven mira la ayuda del QBasic.

FUNCIÓN	DESCRIPCIÓN	EJEMPLO
SQR(num)	Raíz cuadrada	SQR(16) devuelve 4
INT(decimal)	Pasa a entero qitando decimales	INT(1.6) devuelve 1
CINT(decimal)	Redondea a entero (Max = 32767)	CINT(1,6) devuelve 2
CLNG(decimal)	Redondea a entero largo	CLNG(100000.2) devuelve 100000
SIN(num)	Calcula el seno	SIN(40) devuelve 0.64278
COS(num)	Calcula el coseno	COS(40) devuelve 0.76604
TAN(num)	Calcula la tangente	TAN(40) devuelve 0.839009
LEN(Cadena)	Devuelve la longitud de la cadena	LEN("RONDA") devuelve 5
RTRIM(Cadena)	Devuelve una cadena sin espacios al final	RTRIM("Fin. ") devuelve "Fin."
UCASE$(Cadena)	Devuelve la cadena pasada a mayúsculas (No funciona col la ñ ni con los acentos)	UCASE$("Toma Castaña") devuelve "TOMA CASTAñA"
MID$(Cadena, inicio, largo)	Devuelve una subcadena del tamaño indicado a parir de la posición indicada.	MID$("Programación",5,4) devuelve "rama"
STRING$(largo, carácter)	Devuelve una cadena formada por un número de caracteres repetidos	STRING$(10,"#") devuelve "##########"
TIME$	Devuelve la hora actual como cadena	Devuelve por ejemplo "16:45:08"
DATE$	Devuelve la fecha actual como cadena en formato mes-dia-año	Devuelve por ejemplo "01-20-2006"
INKEY$	Devuelve la letra de la última tecla pulsada antes de llegar a ejecutar la función.	Devolvería "A" si la última tecla pulsada fue la "A mayúscula"

En una de las ampliaciones de este curso se habla de un conjunto de funciones muy útiles para el manejo de cadenas y en otro se explica en detalle la función INKEY$ que es especialmente útil para detectar teclas pulsadas en menús, juegos, preguntas que se responden con sí o no, etc.

ESTRUCTURAS CONDICIONALES IF

INTRODUCCIÓN A LOS BLOQUES DE CONTROL

Hasta ahora al ejecutar nuestros programas se han ejecutado todas las instrucciones que aparecen en el listado del código fuente del programa, desde la primera hasta la última.

A partir de ahora vamos a poder conseguir que una o varias instrucciones solo se lleguen a ejecutar si se cumple una condición, que si no se cumple se ejecuten otras, o incluso que algunas instrucciones se ejecuten más de una vez (Esto lo veremos en los siguientes temas).

ALTERNATIVA SIMPLE. INSTRUCCIÓN IF

Empecemos con un ejemplo. Un programa que nos pida la nota de un examen y nos felicite si está aprobado.

```
CLS
INPUT "Escribe la nota del examen: ", nota
IF nota >= 14 THEN
        PRINT "Enhorabuena, has aprobado."
END IF
PRINT "Gracias por usar el programa."
```

Ahora ejecutemos el programa. Hemos sacado un 14:

```
Escribe la nota del examen: 14
Enhorabuena, has aprobado.
Gracias por usar el programa.
```

Volvamos a lanzar el programa, esta vez hemos sacado un 4:

```
Escribe la nota del examen: 4
Gracias por usar el programa.
```

Esta última vez se ha saltado una de las instrucciones del listado ¿Por qué?

Porque hemos usado la instrucción IF (si en inglés), una de las más importantes de todo lenguaje de programación estructurado, y le hemos puesto como condición que el valor almacenado en la variable nota sea mayor o igual que cinco con lo que hacemos que todas las instrucciones que haya entre IF y END IF solo se ejecuten si esta expresión es VERDADERA.

Vamos a ver con más tranquilidad la sintaxis de la instrucción IF

```
IF condición THEN
        instrucciones
END IF
```

SI la condición es VERDADERA, ENTONCES (THEN) se ejecutarán las instrucciones hasta llegar a END IF (FIN SI). Después se seguirá con el resto del programa

Si la condición es falsa se saltará todo el bloque IF y se ejecutarán las siguientes instrucciones del programa que haya después del END IF.

SANGRÍA

En el ejemplo anterior puedes ver que las instrucciones que hay dentro del bloque IF están escritas un poco más a la derecha que las demás, no justo en el margen izquierdo.

Esto se hace por comodidad, para poder saber de un vistazo cuales son las instrucciones que hay dentro del bloque. En la mayoría de lenguajes no es obligatorio (En COBOL sí), pero siempre es muy recomendable hacerlo así para dar mayor claridad al código fuente del programa.

Para hacer esto basta con pulsar la tecla TABULADOR antes de escribir la instrucción, mejor que escribir espacios. En las opciones de QBasic se puede especificar el tamaño de la tabulación (Cuantos espacios se escriben). Por defecto tiene 8, pero es mejor un número más bajo entre 3 y 5 como ya veremos más adenante.

Esta técnica también se conoce como encolumnado o indentación (En inglés se llama indent).

ANIDACIÓN

Como hemos visto en la sintaxis de la instrucción IF, dentro del bloque de instrucciones a ejecutar se puede usar cualquier instrucción, por lo tanto también se pueden usar otras instrucciones IF. A esto es a lo que se llama anidación de instrucciones. Veamos un ejemplo que nos diga si un número es par (usando el operador MOD) y en este caso nos diga también si el número es mayor que 10:

```
CLS
INPUT "Escribe un número: ", num
IF num MOD 2 = 0 THEN
        PRINT "Es un número par"
        IF num > 10 THEN
                PRINT "Es mayor que 10"
        END IF
END IF
```

Ejemplos de la ejecución de este programa pueden ser...

```
Escribe un número: 7
```

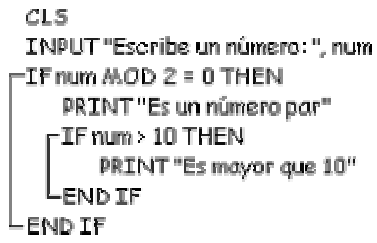
```
Escribe un número: 8
Es un número par
```

Escribe un número: 12
Es un número par
Es mayor que 10

Aquí si la primera condición se cumple se escribe "Es par" y además se hace otra comprobación que se podrá cumplir o no. Nunca se va a llegar a la segunda comprobación si la primera no se ha cumplido.

Es muy importante observar que cada IF lleva su END IF correspondiente y que el bloque de instrucciones del segundo IF está encolumnado todavía más a la derecha que el anterior.

Cuando escribimos un programa en papel es común unir con una línea cada IF con su END IF para aclarar el listado.



INSTRUCCIÓN IF SIMPLIFICADA

Si el bloque de instrucciones de una instrucción IF sólo va a llevar una instrucción podemos escribirla en la misma línea detrás del THEN y ahorrarnos el END IF.

Para el ejemplo del examen podíamos haber hecho:

```
CLS
INPUT "Escribe la nota del examen: ", nota
IF nota >= 5 THEN PRINT "Enhorabuena, has aprobado."
PRINT "Gracias por usar el programa."
```

Esto es útil en algunos casos, pero si vemos que nos vamos a liar es mejor poner el END IF como hemos visto antes, aunque si nos acostumbramos a poner siempre los encolumnados no hay por qué equivocarse.

ALTERNATIVA DOBLE.
INSTRUCCIÓN IF-THEN-ELSE

En el ejemplo del examen estaría bien que "si no" aprueba, decirle que ha suspendido. Lo podemos hacer de esta forma:

```
CLS
INPUT "Escribe la nota del examen: ", nota
IF nota >= 5 THEN
        PRINT "Enhorabuena, has aprobado."
ELSE
        PRINT "Lo siento, has suspendido."
END IF
PRINT "Gracias por usar el programa."
```

En este caso se comprueba la condición del IF. Si es verdadera se ejecuta su bloque de instrucciones y después se sigue con lo que venga detrás del END IF, igual que antes. Ahora viene lo nuevo, si la condición no se cumple se ejecuta el bloque de instrucciones del ELSE hasta el END IF, y después se sigue con lo que haya detrás.

De esta forma nos aseguramos de que siempre se ejecuta uno y solo uno de los dos bloques de instrucciones según la condición sea verdadera o falsa.

Dentro del bloque del ELSE también puede ir cualquier tipo de instrucción, incluido otro bloque IF o del tipo que sea.

Veamos un ejemplo "especialmente malo" de un programa que nos diga el nombre de un mes a partir de su número. En temas posteriores simplificaremos bastante este problema.

```
CLS
INPUT "Escribe el número del mes: ", mes
IF mes = 1 THEN
    PRINT "Enero"
ELSE
    IF mes = 2 THEN
        PRINT "Febrero"
    ELSE
        IF mes = 3 THEN
            PRINT "Marzo"
        ELSE
            IF mes = 4 THEN
                PRINT "Abril"
            ELSE
                IF mes = 5 THEN
                    PRINT "Mayo"
                ELSE
                    IF mes = 6 THEN
                        PRINT "Junio"
                    ELSE
                        IF mes = 7 THEN
                            PRINT "Julio"
                        ELSE
                            IF mes = 8 THEN
                                PRINT "Agosto"
                            ELSE
                                IF mes = 9 THEN
                                    PRINT "Septiembre"
                                ELSE
                                    IF mes = 10 THEN
                                        PRINT "Octubre"
                                    ELSE
                                        IF mes = 11 THEN
                                            PRINT "Noviembre"
                                        ELSE
                                            IF mes = 12 THEN

                                            ELSE

                                            END IF
                                        END IF
                                    END IF
                                END IF
                            END IF
                        END IF
                    END IF
                END IF
            END IF
        END IF
    END IF
END IF
PRINT "S'ACABÓ"
```

Muy sencillo: Si es enero lo escribimos, si no miramos si es febrero, si no a ver si es marzo, etc. así hasta diciembre. Si no es diciembre sacamos un mensaje de mes no válido. Observa que cada IF tiene su ELSE y su END IF correspondiente abajo.

Normalmente nunca llegaremos a estas estructuras tan anidadas, pero aquí se puede ver la importancia de que el tamaño de tabulación no sea muy grande.

Observa que si haces que la ventana del navegador sea más estrecha podría pasar (según los programas) que algunas líneas de las más largas pasan abajo y se nos estropea toda nuestra jerarquía. En QBasic y en los otros editores de programación esto no ocurre porque no tienen salto de línea automático para que no pase eso, pero de todas formas es muy incómodo tener que ir moviendo la barra de desplazamiento horizontal a izquierda y derecha para ver nuestro listado.

ALTERNATIVA MÚLTIPLE.

INSTRUCCIÓN ELSEIF

Vamos a arreglar un poco el ejemplo anterior para no tener tantos END IF.

```
CLS
INPUT "Escribe el número del mes: ", mes
IF mes = 1 THEN
        PRINT "Enero"
ELSEIF mes = 2 THEN
        PRINT "Febrero"
ELSEIF mes = 3 THEN
        PRINT "Marzo"
ELSEIF mes = 4 THEN
        PRINT "Abril"
ELSEIF mes = 5 THEN
        PRINT "Mayo"
ELSEIF mes = 6 THEN
        PRINT "Junio"
ELSEIF mes = 7 THEN
        PRINT "Julio"
ELSEIF mes = 8 THEN
        PRINT "Agosto"
ELSEIF mes = 9 THEN
        PRINT "Septiembre"
ELSEIF mes = 10 THEN
        PRINT "Octubre"
ELSEIF mes = 11 THEN
        PRINT "Noviembre"
ELSEIF mes = 12 THEN
        PRINT "Diciembre"
ELSE
        PRINT "Mes no válido"
END IF
PRINT "S'ACABÓ"
```

A la Palabra Clave ELSE le hemos colocado directamente la otra condición para simplificar un poco el Algoritmo, pero viene a ser prácticamente lo mismo. Veamos la sintaxis de forma un poco más clara:

```
IF condición THEN
        bloqueInstrucciones
ELSEIF otra condición THEN
        bloqueInstrucciones
ELSE
        bloqueInstrucciones
```

END IF

Si la condición del IF se cumple se ejecutan sus instrucciones y ya está.

Si no, se comprueba la condición del primer ELSEIF y si es verdadera se ejecutan sus instrucciones y ya está.

Si no, se comprueba la condición del siguiente ELSEIF y si es verdadera se ejecutan sus instrucciones y ya está.

Si la condición del último ELSEIF no se cumple se ejecuta el bloque ELSE si existe.

Puede haber tantos bloques ELSEIF con su condición como sea necesario, pero solo un bloque ELSE (o ninguno) al final. Si no hay bloque ELSE puede suceder que no se ejecute nada porque no se cumpla ninguna de las condiciones.

Las condiciones no tienen que estar relacionadas de ninguna forma, aquí siempre hemos preguntado por el mes, pero podíamos haber comprobado cualquier otra cosa, o poner los meses desordenados.

Esta estructura ELSEIF no se usa mucho y no existe en algunos otros lenguajes de programación, en su lugar se usan los IF anidados como vimos en el ejemplo larguísimo anterior o la estructura SELECT que veremos en el tema siguiente.

Una última cosa antes de acabar con los IF. Si escribimos ENDIF todo junto, QBasic y Visual Basic nos lo corrigen automáticamente. Esto es porque en alguna versión muy antigua del lenguaje BASIC se escribía así.

ESTRUCTURA DE SELECCIÓN SELECT

En el ejemplo de instrucciones IF anidadas nos salía un pedazo de listado para decir el nombre del mes. Después lo arreglamos un poco con las instrucciones ELSEIF. Ahora vamos a hacerlo todavía un poco mejor.

```
CLS
INPUT "Escribe el número del mes: ", mes
SELECT CASE mes
       CASE 1
              PRINT "Enero"
       CASE 2
              PRINT "Febrero"
       CASE 3
              PRINT "Marzo"
       CASE 4
              PRINT "Abril"
       CASE 5
              PRINT "Mayo"
       CASE 6
              PRINT "Junio"
       CASE 7
              PRINT "Julio"
       CASE 8
              PRINT "Agosto"
       CASE 9
              PRINT "Septiembre"
       CASE 10
              PRINT "Octubre"
       CASE 11
              PRINT "Noviembre"
       CASE 12
              PRINT "Diciembre"
       CASE ELSE
              PRINT "Mes no válido"
END SELECT
PRINT "S'ACABÓ"
```

Hemos usado una estructura nueva: La instrucción SELECT.
Esta estructura es equivalente a las anteriores de IF anidados, pero es más fácil de manejar y el programa queda más estructurado.
Si has entendido la estructura IF no te será muy difícil entender esta.
Lo primero es escribir las Palabras Clave SELECT CASE seguidas de una expresión. Esta expresión es normalmente simplemente una variable que puede ser de cadena o numérica. En este caso no tiene que devolver VERDADERO o FALSO como en los IF ya que no se usan operadores relacionales ni lógicos, solo los aritméticos cuando hace falta.
A continuación para cada resultado posible se pone la Palabra Clave CASE y la expresión a comparar con la del principio. Si la comparación es verdadera se ejecuta el bloque de instrucciones entre este CASE y el siguiente y se sale de la estructura. Si la condición es falsa se ejecuta el bloque del CASE ELSE si existe y si no nada.
Las expresiones de los cases se pueden poner de una de las siguientes formas:

```
CASE 1
```

Una expresión (En este caso un número), igual que en el ejemplo.

```
CASE 1, 2, 3
```

Varias expresiones separadas por comas.

```
CASE 1 TO 3
```

Un intervalo de valores, ambos inclusive, separados por la palabra clave TO. En este caso si la expresión inicial era de tipo entero serán válidos los resultados 1, 2 y 3, pero si era de tipo real serán válidos todos los números posibles entre el 1 y el 3 como por ejemplo el 1.517512 y el 2.17521.

```
CASE IS > 2
CASE IS = 5
CASE IS <> 8
CASE IS <= 6
```

Usando operadores relacionales. En el primer ejemplo serán válidos todos los valores mayores que 2, en el segundo sólo el 5, en el tercero cualquiera menos el 8 y en el último los que sean menor o igual que 6.
Si se nos olvida la palabra clave IS, QBasic la escribirá por nosotros.
Normalmente escribiremos las expresiones de los CASE de la forma más sencilla posible evitando intervalos muy extraños y teniendo cuidado con los operadores relacionales para no dejarnos "fuera" ningún valor posible, pero podemos llegar a escribir una estructura SELECT tan mal hecha como esta sin que el programa de ningún error de ejecución.

```
SELECT CASE n
       CASE 1
       CASE 1
       CASE 3
       CASE 2 TO 4
       CASE IS < 5
       CASE IS <= 5
       CASE <> 100
       CASE 16 TO 34
END SELECT
```

Hay expresiones repetidas, valores que entran en varias expresiones como el 3, un desastre. ¿Que ocurriría aquí?
Lo primero es que nunca ocurriría nada porque no hemos puesto bloques de instrucciones en los CASE, pero si las hubiéramos puesto pasaría lo siguiente. QBasic empieza a comprobar por los CASE hasta que encuentre uno que le venga bien. Cuando lo encuentra ejecuta su bloque de instrucciones y sale del SELECT CASE aunque otros bloques posteriores también hubieran servido. En este caso si el valor de n es 1 se ejecuta el primer CASE 1, el segundo no se llega a ejecutar nunca. Si el valor de n es 3 se ejecuta el CASE 3 y se sale, aunque los cuatro siguientes también hubieran servido. Si el valor de n es 100 no se ejecuta nada porque ningún CASE sirve y tampoco hay un CASE ELSE.
Las expresiones de los cases normalmente serán simplemente números (o cadenas) y algún intervalo alguna vez para poder estar seguro de que se va a ejecutar siempre el bloque correcto y nuestro programa va a funcionar bien.
Entre el SELECT CASE y el primer CASE no puede haber nada.
Si los bloques de instrucciones van a llevar sólo una instrucción sencilla podemos ponerla a continuación de la expresión del CASE separándola con dos puntos, por ejemplo nuestro ejemplo de los meses quedaría así:

```
SELECT CASE mes
       CASE 1: PRINT "Enero"
       CASE 2: PRINT "Febrero"
       CASE 3: PRINT "Marzo"
       CASE 4: PRINT "Abril"
       CASE 5: PRINT "Mayo"
       CASE 6: PRINT "Junio"
       CASE 7: PRINT "Julio"
       CASE 8: PRINT "Agosto"
       CASE 9: PRINT "Septiembre"
       CASE 10: PRINT "Octubre"
```

```
        CASE 11: PRINT "Noviembre"
        CASE 12: PRINT "Diciembre"
        CASE ELSE: PRINT "Mes no válido"
END SELECT
```

con lo que conseguimos un listado casi la mitad más corto.
Por supuesto las expresiones también pueden ser de cadenas:

```
INPUT "Escribe el nombre de un periférico del ordenador: ", perif$
SELECT CASE perif$
        CASE "Teclado", "Ratón"
                PRINT "Es un periférico de entrada"
        CASE "Monitor", "Impresora"
                PRINT "Es un periférico de salida"
        CASE "Módem"
                PRINT "Es un periférico de entrada/salida"
        CASE ELSE
                PRINT "Este periférico no lo conozco"
END SELECT
```

Y por supuesto los bloques de instrucciones de los CASE pueden contener cualquier tipo tipo de instrucciones anidadas en su interior como bloques IF, otro SELECT, etc.
Si estos bloques se hacen muy largos no te asustes, solucionaremos el problema cuando lleguemos a Programación Modular donde el SELECT servirá como menú para dar entrada a distintos procedimientos o subprogramas.
Casi todos los lenguajes tienen una estructura equivalente a SELECT, pero en el caso del lenguaje C es bastante mala. Hay que tener en cuenta de que en los lenguajes de los sistemas de bases de datos (SQL) existe una instrucción SELECT que no tiene nada que ver con esto, sirve para sacar información de las bases de datos.

ESTRUCTURA DE REPETICIÓN FOR...NEXT

1.9.1 - Introducción a estructuras de control repetitivas

En los siguientes temas vamos a ver las instrucciones que existen en la Programación Estructurada para conseguir que un bloque de instrucciones se puedan ejecutar más de una vez sin necesidad de escribirlas repetidas en el listado del código fuente del programa.

En lenguaje ensamblador y en las versiones antiguas da BASIC se usan instrucciones de tipo GOTO que continúan la ejecución del programa en otra parte, pero esto da lugar a programas muy reliados (Código "espagueti") que son muy difíciles de depurar y pueden contener errores. Para solucionar el problema en la Programación Estructuradas existen estructuras de control que encierran un conjunto de instrucciones (con una instrucción al principio y otra al final) y lo que hacen es ejecutar el bloque de instrucciones entero un número determinado de veces, mientras se cumpla una condición o hasta que se cumpla una condición, según sea la estructura. A estas estructuras también se las conoce como "Bucles" o "Lazos".

Cómo detener un programa

Al usar estas estructuras nos podemos encontrar con el problema de que si el programa no está escrito correctamente nunca se salga de la estructura de control produciéndose el efecto llamado "Bucle infinito" que puede llegar a bloquear el ordenador.

En un programa normal ya compilado y terminado que se ejecute bajo Windows puede pasar que el ordenador se bloquee y aparezca una pantalla azul recuerdo de Bill Gates de tipo "El sistema está ocupado o no responde...", con lo que casi seguro que vamos a tener que reiniciar el ordenador.

En los entornos de programación esto normalmente no llegará a ocurrir. En caso de que nuestro programa se bloquee puede que se agoten los recursos del sistema y el programa se detenga dando un error de tiempo de ejecución y volviendo al editor de código.

Si el programa se queda bloqueado se puede pulsar la siguiente combinación de teclas:

Control + Pausa

para detener la ejecución del programa y volver al editor de código donde habrá que repasar el código para que esto no ocurra y el programa funcione siempre bien. En algunos casos tras pulsar esta combinación de teclas habrá que pulsar una vez la tecla ENTER para desbloquear el programa.

Instrucción FOR...NEXT

Empecemos con un ejemplo como siempre. Vamos a escribir un programa que escriba los números del 1 al 5 usando las instrucciones que ya conocemos.

```
CLS
PRINT 1
PRINT 2
```

```
PRINT 3
PRINT 4
PRINT 5
```

Como se puede ver es un programa bastante tonto. Hay cinco instrucciones casi iguales. Solo cambia el valor de la expresión que cada vez vale lo que en la instrucción anterior más uno, por lo tanto también podíamos haber hecho esto:

```
CLS
n = 0
n = n + 1
PRINT n
n = n + 1
PRINT n
n = n + 1
PRINT n
n = n + 1
PRINT n
n = n + 1
PRINT n
```

Se puede comprobar que el resultado es el mismo que en el programa anterior y ahora sí que tenemos cinco pares de instrucciones completamente idénticos. Vamos a hacer el mismo programa con la nueva instrucción FOR ("Para" en castellano):

```
CLS
FOR n = 1 TO 5
        PRINT n
NEXT
```

Ya está. Mira que sencillo, pero ahora viene la explicación.

Esto lo que hace es que se ejecute lo que hay entre el FOR y el NEXT cinco veces siguiendo estos pasos:

- La primera vez n vale 1, como pone en la instrucción.
- Se ejecuta el bloque de instrucciones con n valiendo 1
- AUTOMÁTICAMENTE n se incrementa en 1, pasando a valer 2
- Se comprueba que n es menor o igual que 5, y como lo es se sigue.
- Se ejecuta el bloque de instrucciones con n valiendo 2
- AUTOMÁTICAMENTE n se incrementa en 1, pasando a valer 3
- Se comprueba que n es menor o igual que 5, y como lo es se sigue.
- Se ejecuta el bloque de instrucciones con n valiendo 3
- AUTOMÁTICAMENTE n se incrementa en 1, pasando a valer 4
- Se comprueba que n es menor o igual que 5, y como lo es se sigue.
- Se ejecuta el bloque de instrucciones con n valiendo 4
- AUTOMÁTICAMENTE n se incrementa en 1, pasando a valer 5
- Se comprueba que n es menor o igual que 5, y como lo es se sigue.
- Se ejecuta el bloque de instrucciones con n valiendo 5
- AUTOMÁTICAMENTE n se incrementa en 1, pasando a valer 6
- Se comprueba que n es menor o igual que 5, y como ya no lo es se sale del bucle y se ejecuta la siguiente instrucción que venga detrás del NEXT.

Todo esto puede parecer muy complicado, pero con la práctica conseguiremos que esta sea una de las instrucciones más fáciles de entender de la programación, sólo habrá que detenerse a pensar en estos pasos cuando algún programa no haga lo que queremos y no demos con el error.

Veamos la sintaxis de la instrucción FOR:

```
FOR contador = inicio TO final
        bloqueInstrucciones
NEXT
```

contador es la variable que usaremos como contador (el FOR la modifica automáticamente) y tendrá que ser de tipo numérico, normalmente entero aunque también puede ser real. Ya hemos hablado de los contadores en el tema de los operadores de asignación.

inicio es una expresión numérica cuyo valor tomará el contador la primera vez.

final es una expresión numérica cuyo valor lo usará el FOR de forma que solo entrará si el contador no supera al valor de esta expresión. En nuestro ejemplo el final era 5 y cuando el contador (n) llegaba a valer 6 ya no entrábamos.

Ahora vamos a ver dos normas muy importantes que hay que seguir siempre con los contadores de los FOR

- No debemos modificar el valor de esta variable dentro del bucle, ya lo hace automáticamente la instrucción FOR. Dicho de otra forma: No debemos asignar ningún valor a esta variable hasta después de terminar el FOR.
- Una vez terminado el FOR no debemos leer el valor de la variable contador porque su valor queda indeterminado. Podremos usar esta variable más adelante si previamente le asignamos un valor antes de intentar leerla.

Estas normas nos las podríamos saltar sin dar un error de ejecución, pero puede que el mismo algoritmo de distintos resultados en distintas versiones de BASIC, ya que el contador es manejado internamente por el intérprete del lenguaje de programación y puede que no siempre se haga de la misma forma.

Los valores inicio y fin no tienen por que ser expresiones constantes. En este ejemplo escribiremos los números desde uno hasta donde quiera el usuario:

```
CLS
INPUT "Escribe hasta dónde quieres llegar: ", max
FOR n = 1 TO max
        PRINT n
NEXT
```

No es necesario que tengamos que usar siempre el valor del contador para calcular algo. Este FOR escribe "Hecho en Ronda" siete veces:

```
FOR n = 1 TO 7
        PRINT "Hecho en Ronda"
NEXT
```

y este hace exáctamente lo mismo:

```
FOR n = 82 TO 88
        PRINT "Hecho en Ronda"
NEXT
```

El siguiente escribe los pares del 2 al 10, es decir, 2, 4, 6, 8, 10.

```
FOR n = 1 TO 5
        PRINT n * 2
```

```
NEXT
```

En QBasic hay una forma de hacer esto más fácilmente:

```
FOR n = 2 TO 10 STEP 2
        PRINT n
NEXT
```

Antes veíamos que el FOR incrementa automáticamente al contador en 1 en cada pasada. Usando la palabra clave STEP seguida de una expresión numérica conseguimos modificar este incremento.

Otro ejemplo con STEP que se explica solo.

```
CLS
INPUT "Escribe un número: ", s
PRINT "Estos son los números del 0 al 100 de "; s; " en "; s
FOR n = 0 TO 100 STEP s
        PRINT n
NEXT
```

Todo esto funciona muy bien, espero que se entienda. Pero puede surgir una duda, supongamos que escribimos el número 7 y el programa escribe de siete en siete, dando este resultado:

```
Escribe un número: 7
Estos son los números del 0 al 100 de 7 en 7
0
7
14
21
28
35
42
49
56
63
70
77
84
91
98
```

Como se puede ver, no se ha alcanzado el 100 es porque el siguiente valor que sería el 105 ya supera al 100 que es valor final del FOR y no se ejecuta.

También puede ocurrir que la expresión del STEP valga 0. En este caso el FOR incrementará en cero el contador con lo que nunca se llegará al valor final y se producirá un bucle infinito. Habrá que pulsar Ctrl+Pausa para detener el programa y corregir el código.

Ahora ya podemos hacer que un FOR funcione hacia atrás, escribiendo la expresión final menor que la inicial y una expresión negativa en el STEP. Como ejemplo un FOR que escriba los números del 10 al 1.

```
FOR n = 10 TO 1 STEP -1
        PRINT n
NEXT
```

Si no usamos el STEP negativo y escribimos el valor final menor que el inicial, nunca se ejecutará el bloque FOR. Si un programa no funciona bien porque un FOR no se ejecuta nunca será conveniente revisar esto.

Como siempre, dentro del bloque FOR puede ir cualquier tipo de instrucciones, incluido otro FOR. Veamos un ejemplo:

```
FOR i = 1 TO 8
        FOR j = 1 TO 5
                PRINT "Hola"
        NEXT
NEXT
```

¿Cuantas veces escribirá "Hola" este programa? Si el FOR interior se ejecuta entero 8 veces y cada vez escribe "Hola" 5 veces, en total lo hará 8 por 5 igual a 40 veces, es decir, el producto.

Este tipo de instrucciones son especialmente útiles en algoritmos que ya veremos más adelante como el recorrido de matrices.
Hay que tener cuidado de no usar la misma variable contador para los dos FOR, ya que romperíamos la regla de no modificar el valor del contador del primer FOR y el programa no funcionaría bien.

QBasic permite escribir a continuación del NEXT el nombre del contador del FOR, por ejemplo:

```
FOR i = 1 TO 8
        FOR j = 1 TO 5
                PRINT "Hola"
        NEXT j
NEXT i
```

Esto puede ser útil para saber en un listado muy complicado a que FOR corresponde cada NEXT, pero si encolumnamos correctamente nuestro programa esto no será necesario.

El FOR en Basic es bastante flexible. En otros lenguajes funciona de otra forma o incluso puede que ni siquiera exista, ya que como veremos a continuación no es imprescindible para construir un algoritmo.

ESTRUCTURA DE REPETICIÓN WHILE...WEND

En este tema vamos a ver una estructura repetitiva más primitiva que el PARA ya que no maneja automáticamente el contador y por lo tanto es más difícil de utilizar, pero usada corréctamente puede ser bastante más flexible.

Recordemos el ejemplo de escribir los números del 1 al 5 con la instrucción FOR.

```
FOR n = 1 TO 5
        PRINT n
NEXT
```

Ahora veremos como se hace lo mismo en QBasic usando la instrucción WHILE (Mientras).

```
n = 1
WHILE n <= 5
        PRINT n
        n = n + 1
WEND
```

Esto lo que hace es ejecutar el bloque de instrucciones (Lo que hay entre el WHILE y el WEND) una y otra vez **mientras** se cumpla la condición del WHILE. Un poco más difícil que con el FOR. Vamos a verlo paso a paso:

- Usaremos como contador la variable n y por tanto la tenemos que inicializar nosotros al valor que queramos que tenga la primera vez, en este caso 1.
- Se comprueba la condición del WHILE: como n vale 1 que es menor o igual que 5, entramos.
- Se ejecutan las instrucciones del bloque con n valiendo 1. La última instrucción incrementa n en 1, con lo que pasa a valer 2.
- Volvemos al WHILE y se comprueba su condición: como n vale 2 que es menor o igual que 5, entramos.
- Se ejecutan las instrucciones del bloque con n valiendo 2. La última instrucción incrementa n en 1, con lo que pasa a valer 3.
- Volvemos al WHILE y se comprueba su condición: como n vale 3 que es menor o igual que 5, entramos.
- Se ejecutan las instrucciones del bloque con n valiendo 3. La última instrucción incrementa n en 1, con lo que pasa a valer 4.
- Volvemos al WHILE y se comprueba su condición: como n vale 4 que es menor o igual que 5, entramos.
- Se ejecutan las instrucciones del bloque con n valiendo 4. La última instrucción incrementa n en 1, con lo que pasa a valer 5.
- Volvemos al WHILE y se comprueba su condición: como n vale 5 que es menor o igual que 5, entramos.
- Se ejecutan las instrucciones del bloque con n valiendo 5. La última instrucción incrementa n en 1, con lo que pasa a valer 6.
- Volvemos al WHILE y se comprueba su condición: como n vale 6 que ya no es menor o igual que 5, no entramos y pasamos a la siguiente instrucción que haya detrás del WEND.

Se puede ver que el funcionamiento es parecido al del FOR, solo que aquí lo tenemos que controlar todo nosotros. Las dos reglas que dijimos sobre los contadores del FOR ya aquí no tienen sentido porque de hecho nosotros vamos a tener que incrementar el contador haciendo una asignación y una vez terminado podemos estar seguro del valor que tiene la variable.

Una norma que sí conviene respetar (Aunque no siempre es necesario) es que la instrucción que incrementa el contador sea la última del bloque, ya que si está en otro sitio ejecutaremos unas instrucciones con un valor y las demás con el otro, con lo que nos podemos liar. Un error

muy típico es que se nos olvide de poner la instrucción de incrementar el contador, produciendo un bucle infinito que hará que nuestro programa no termine nunca. Si un programa se bloquea es conveniente revisar esto.

También puede pasar que no lleguemos a entrar al MIENTRAS porque la condición ya sea falsa la primera vez, por ejemplo:

```
contador = 120
WHILE contador < 100
        PRINT "Esto no se va a llegar a escribir nunca.”
        contador = contador + 1
WEND
```

Hasta ahora hemos hablado de contador, pero como veremos en los ejemplos podemos usar un acumulador, o ninguno de los dos, ya que la condición del WHILE puede ser cualquiera y no hay porqué contar ni acumular algo siempre.

Veamos algunos ejemplos de MIENTRAS:

Rutina que escribe del 0 al 100 de 2 en 2:

```
c = 0
WHLE c <= 100
        PRINT c
        c = c + 2
WEND
```

Escribir de 50 hasta 1 hacia atrás:

```
c = 50
WHILE c >= 1
        PRINT c
        c = c - 1
WEND
```

Calcular el factorial de un número que pedimos al usuario (Este ejemplo se mejorará en otro tema más adelante):

```
INPUT "Escribe un número para calcular su factorial: ", num
c = 1
factorial = 1
WHILE c <= num
        factorial = factorial * c
        c = c + 1
WEND
PRINT "El factorial de”; num; "es”; factorial
```

Leer números por teclado hasta que se escriba el 0:

```
INPUT "Escribe números (0 para salir):”, num
WHILE num <> 0
        INPUT "Escribe números (0 para salir):”, num
WEND
```

Este último ejemplo presenta lo que se conoce como lectura anticipada. Antes de llegar al WHILE hemos tenido que conseguir el valor de la variable num porque si no lo hacemos puede pasar que num valga cero y por lo tanto no lleguemos a entrar al bucle. Esto es útil en casos

como la lectura de ficheros secuenciales, pero otras veces conviene evitarlo para no repetir instrucciones. Veremos como hacerlo en el siguiente tema.

Una última cosa es que hay una teoría en informática que dice que cualquier algoritmo puede ser programado usando solamente instrucciones MIENTRAS. Es decir, ni bloques IF, ni ELSE, ni CASE, ni FOR, ni otras estructuras que veremos más adelante. Yo no lo he comprobado, pero si a alguien le ha gustado mucho este tema ya puede empezar a hacerlo.

ESTRUCTURA DE REPETICIÓN DO...LOOP

ESTRUCTURA DO...LOOP

Esta estructura es similar a la WHILE, solo que la condición se especifica al final del bloque, con lo que puede ser más fácil de entender y nunca tendremos que hacer "lecturas anticipadas" ya que siempre se entra por lo menos una vez.

La principal novedad de este bloque es que podemos repetir las instrucciones **MIENTRAS** se cumpla la condición o **HASTA** que se cumpla.

El siguiente ejemplo escribe los números del 1 al 5 usando una instrucción DO...LOOP WHILE que hace que el bucle se ejecute MIENTRAS nuestro contador sea menor que 5.

```
n = 0
DO
        n = n + 1
        PRINT n
LOOP WHILE n < 5
```

Observa que el contador se incrementa al principio del bucle, y por lo tanto la primera vez que escribamos n, ya tendrá el valor de 1. La última vez escribirá 5 y al llegar a la condición se comprobará que NO es menor que 5 y ya salimos.

Ahora haremos lo mismo con la instrucción DO... LOOP UNTIL que ejecutará el bloque HASTA QUE el contador llegue a valer 5.

```
n = 0
DO
        n = n + 1
        PRINT n
LOOP UNTIL n = 5
```

Esto es parecido. Observa que la condición es justo la contraria. La última vez n vale 5 y después de escribirla se comprueba que la condición es verdadera y se sale del bucle.

Las instrucciones DO...LOOP UNTIL son normalmente las más fáciles de comprender. En la ayuda de QBasic recomiendan que se dejen de usar las instrucciones WHILE...WEND para usar mejor las DO...LOOP, pero algunas veces será mejor usar las WHILE...WEND como en el caso de los ficheros secuenciales que ya veremos más adelante.

DEPURACIÓN DE DATOS DE ENTRADA

Ahora ya estamos en condiciones de ver una forma de conseguir que un programa que requiere la intervención del usuario para introducir datos de entrada no avance HASTA QUE el usuario no escriba los datos correctos.

En el tema de Entrada/Salida vimos que QBasic es capaz de controlar que no se metan valores fuera de rango, por ejemplo que si el programa pide un entero no se pueda escribir un numero mayor de 32767. Aquí lo que vamos a ver es como conseguir que se pedimos un mes el

programa no avance hasta que el usuario escriba un número entre 1 y 12. Esto lo deberíamos hacer en cualquier programa siempre que pidamos al usuario que escriba algo.

Para hacer este control lo que hacemos es meter la instrucción INPUT dentro de un bloque REPETIR, del que no salimos HASTA que la respuesta sea correcta o MIENTRAS sea incorrecta.

Vamos a ver unos ejemplos que aclararán todas las posibles situaciones.

Leer un número menor que 100

```
CLS
DO
        INPUT "Escribe un número menor que 100: ",num
LOOP UNTIL num < 100
```

Aquí no seguimos hasta que el número sea menor que 100. En el siguiente ejemplo seguiremos repitiendo la pregunta mientras el número sea mayor o igual que 100

```
CLS
DO
        INPUT "Escribe un número menor que 100: ",num
LOOP WHILE num >= 100
```

Puedes volver a comprobas aquí que para el mismo problema la condición del LOOP WHILE es justo la inversa a la del LOOP UNTIL.

Ahora pedimos un mes, que tiene que ser entre 1 y 12

```
CLS
DO
        INPUT "Escribe un mes (0 a 12): ",mes
LOOP UNTIL (mes >= 1) AND (mes <= 12)
```

En este caso tenemos un intervalo y por lo tanto hay que controlar dos condiciones que uniremos con el operador AND, con lo que no seguimos HASTA que la primera se cumpla Y la segunda también.

También lo podíamos haber hecho con un bloque DO...LOOP WHILE

```
CLS
DO
        INPUT "Escribe un mes (0 a 12): ",mes
LOOP WHILE (mes < 1) OR (mes > 12)
```

Ahora no salimos MIENTRAS alguna de las dos condiciones se cumpla, son justo las contrarias a la del ejemplo anterior, ya que usamos el operador OR que también se puede decir que es el contrario al AND.

Por último vamos a ver un problema muy típico: Un programa que nos pide una clave de acceso para poder seguir. Este es el caso más sencillo en el que no pasamos hasta que no demos con la clave, otros problemas más complicados serían que el programa terminara tras varios intentos fallidos o que admitiera varias claves llévándonos según la que sea a una parte del programa.

```
CONST ClaveCorrecta$="Ábrete Sésamo"
CLS
```

```
DO
        INPUT "Escribe la clave: ",clavePrueba$
LOOP UNTIL clavePruebas = claveCorrecta$
PRINT "Ya has entrado"
```

Recordar también que en un programa terminado para explotación la clave nunca va a estar en el listado del programa, sino que será obtenida de algún tipo de fichero o base de datos para que el usuario tenga la posibilidad de cambiarla.

ARRAYS

INTRODUCCIÓN A LOS ARRAYS

Hasta ahora hemos usado variables para guardar la información que maneja el programa. Una variable con su nombre y su tipo para cada dato (nombre, cantidad, nivel, precio, edad...) que nos hacía falta. Pero ¿Que ocurre si nuestro programa tiene que manejar una colección de datos muy grande como los nombres de una agenda o los precios de un catálogo? Se podría hacer usando 100 variables distintas pero el algoritmo llegaría a ser terriblemente complicado y muy poco funcional. En este apartado vamos a ver cómo podemos asignar un mismo nombre de variable para muchos datos y referirnos a cada uno de ellos de forma individual usando un número al que llamaremos índice.

Usaremos unas estructuras de datos llamadas arrays, de las que se dice que son "Estructuras estáticas de almacenamiento interno".

- Son estáticas porque su tamaño se declara al principio del programa y ya no se puede modificar. Si hacemos un programa que trabaje con 100 nombres siempre lo hará con 100 aunque solo lleguemos a usar 20, y nunca podrá trabajar con más de 100.
- Son de almacenamiento interno porque están en la memoria RAM del ordenador, como variables que son, y su tiempo de vida sólo dura mientras se ejecuta el programa. Al terminar se pierde su contenido.

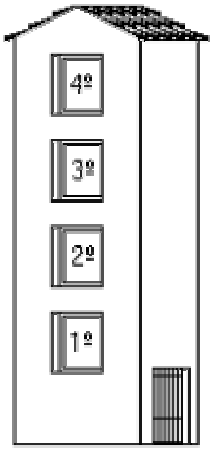
En programación normalmente se conocen como "Arrays" aunque su nombre más correcto en castellano sería "formaciones".

Como veremos en los apartados siguientes, los arrays de 1 dimensión se llaman vectores, los de 2 dimensiones se llaman matrices y los de 3 o más se llaman poliedros o arrays multidimensionales.

En informática muchas veces se llama vector a cualquier array. En la ayuda de QBasic (y de Visual Basic) siempre llaman Matriz a cualquier array, por lo que hay que tener cuidado de no liarse.

VECTORES

Imaginemos este bloque de pisos de cuatro plantas.



Vamos a escribir un programa de la forma que sabemos hasta ahora que nos pregunte el nombre de la persona vive en cada piso y una vez que ha recopilado toda esa información, nos deje preguntarle quien vive en el piso que nosotros queramos.

```
CLS
INPUT "Nombre de quien vive en el 1º: ", nombre1$
INPUT "Nombre de quien vive en el 2º: ", nombre2$
INPUT "Nombre de quien vive en el 3º: ", nombre3$
INPUT "Nombre de quien vive en el 4º: ", nombre4$
DO
        INPUT "Escribe un piso para ver quien vive en él: ",n
LOOP WHILE (n < 1) OR (n > 4)
SELECT CASE n
        CASE 1: PRINT "En el 1º vive "; nombre1$
        CASE 2: PRINT "En el 2º vive "; nombre2$
        CASE 3: PRINT "En el 3º vive "; nombre3$
        CASE 4: PRINT "En el 4º vive "; nombre4$
END SELECT
```

El resultado podría ser este:

```
Nombre de quien vive en el 1º: Paca
Nombre de quien vive en el 2º: Manolo
Nombre de quien vive en el 3º: Lola
Nombre de quien vive en el 4º: Pepe
Escribe un piso para ver quien vive en él: 3
En el 3º vive Lola
```

Un listado un poco largo para hacer algo tan sencillo. Si en vez de cuatro pisos fueran 40 tendríamos un programa casi diez veces más largo con muchas partes casi iguales, pero que no podemos meter en bucles repetitivos porque cada variable es distinta. Observa que comprobamos que el piso sea entre 1 y 4, a partir de ahora va a ser muy importante depurar los datos de entrada, ya veremos por qué.

Ahora vamos a escribir un programa que haga lo mismo que el anterior, pero usando un VECTOR.

```
DIM nombre$ (1 TO 4)
FOR n = 1 TO 4
        PRINT "Nombre de quien vive en el"; n; "º: ";
        INPUT "", nombre$(n)
NEXT
DO
```

```
        INPUT "Escribe un piso para ver quien vive: ",n
LOOP WHILE (n < 1) OR (n > 4)
PRINT "En el";n;"º vive ";nombre$(n)
```

El resultado sería similar, pero el listado es mucho más corto, especialmente todo el SELECT CASE anterior que se ha transformado en una sola instrucción.

Vamos a ver este programa línea a línea.

La primera instrucción es nueva. En ella lo que hacemos es DECLARAR una variable que se va a llamar nombre$ (como lleva el $ va a ser de tipo texto) y va a poder guardar cuatro valores a los que accederemos con subíndices que van desde el 1 hasta el 4. Para determinar este intervalo de valores hay que usar números enteros constantes, no valen expresiones matemáticas.

En el siguiente bloque FOR, que se ejecutará 4 veces lo que hacemos es ir pidiendo al usuario que escriba los nombres. La primera vez guardamos lo que escriba en la posición 1 del vector porque hacemos referencia al índice 1 entre paréntesis a continuación del nombre del vector. La siguiente vez al índice 2, la siguiente vez al 3 y la última vez que se ejecute el bucle hacemos referencia al índice 4. A esto es a lo que se llama "Recorrer el vector" ya que hemos hecho algo con cada uno de sus elementos. Normalmente esto lo haremos siempre con un bucle FOR, que es lo más cómodo.
La prengunta de "nombre de quien vive en..." no está en el INPUT porque esta instrucción no evalúa expresiones, por eso está antes en un PRINT normal con un punto y coma al final para que el cursor no pase a la siguiente línea.

Ahora ya tenemos dentro del vector los cuatro nombres para usarlos como queramos haciendo referencia al nombre del vector y al subíndice entre paréntesis. Para hacer referencia a los subíndices de un array se puede usar cualquier expresión, no tiene porqué ser un número constante, pero hay que tener cuidado de no hacer nunca referencia a índices que no existan

En el siguiente bloque pedimos al usuario que escriba el número de un piso y lo guardamos en la variable N. Obligamos a que sea un número entre 1 y 4.

Al final viene lo espectacular. Para acceder a cualquiera de los índices del vector se puede hacer directamente tomando el valor de la variable N como subíndice sin necesidad de controlar cada valor por separado como antes.

Vamos con otro ejemplo para que todo esto vaya quedando cada vez más claro.

En temas anteriores teníamos un programa para escribir el nombre de un mes a partir de su número. Lo hicimos con muchos IF anidados, después con ELSEIF y por último con SELECT CASE que ya quedaba mucho más corto, pero de ninguna forma nos libramos de escribir todo el SELECT CASE y todos los meses en cualquier parte del programa dónde queramos que se escriba el nombre de algún més.

Ahora vamos a plantear el problema de otra forma:

```
DIM mese$ (1 TO 12)
mese$(1)="Enero"
mese$(2)="Febrero"
mese$(3)="Marzo"
mese$(4)="Abril"
mese$(5)="Mayo"
mese$(6)="Junio"
mese$(7)="Julio"
mese$(8)="Agosto"
mese$(9)="Septiembre"
```

```
mese$(10)="Octubre"
mese$(11)="Noviembre"
mese$(12)="Diciembre"
'(...)
PRINT mese$(2) 'Escribe Febrero
'(...)
n=6
PRINT mese$(n) 'Escribe Junio
```

Hemos declarado un vector de cadenas de 12 posiciones llamado mese$.
Al principio del programa llenamos el vector con los nombres de los meses, cada uno en su lugar correcto.
Donde nos haga falta el nombre de un mes sólo tendremos que usar el vector y referirnos al mes que queramos. De esta forma meses(3) nos devolverá "Marzo" y si n vale 11 entonces meses(n) nos devolverá "Noviembre".

En los dos ejemplos que hemos puesto los índices de los vectores han empezado en el 1, pero esto no tiene que ser siempre así. En QBasic pueden empezar por cualquier número incluso negativo, aunque lo más normal es que siempre empiecen por 1 o por 0. El valor más bajo posible para los índices es -32768 y el más alto es 32767. Por supuesto el final del intervalo no puede ser menor que el principio. Veamos algunos ejemplos más de declaración de vectores.

```
DIM alumno$(1 TO 30) '30 cadenas
DIM nota (1 TO 30) '30 números reales
DIM ventas_verano%(6 TO 9) '4 enteros
```

También podemos declarar vectores de un solo elemento, aunque esto puede que no tenga mucho sentido.

```
DIM número (1 TO 1)
```

Para determinar el tamaño de cualquier vector usamos la siguiente fórmula:

ÚLTIMO INDICE - PRIMER ÍNDICE + 1

Así en el ejemplo de los meses resulta 12 - 1 + 1 = 12 elementos, muy sencillo ¿no? pero a veces terminaremos contando con los dedos.

Para inicializar un vector (Borrarlo entero) no hace falta recorrerlo, podemos hacer:

```
ERASE (nombre_vector)
```

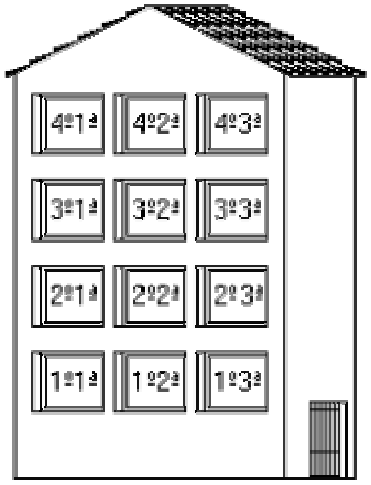
Y el vector se quedará lleno de ceros si es numérico o de cadenas vacías ("") si es de cadenas. Recordemos que en QBasic todas las variables están a cero al principio, pero en otros lenguajes no.

Ahora vamos a ver el problema más típico de los vectores (y en general de todos los arrays). No podemos intentar acceder a subíndices que no existen. En caso de que llegue a ocurrir en QBasic se producirá un error de tipo "Subíndice fuera del intervalo" y el programa se detendrá sin mayores consecuencias.
En muchos otros lenguajes no ocurrirá nada, pero si estamos leyendo sacaremos datos de otras posiciones de memoria que no son nuestras de más allá del final del vector y si estamos escribiendo lo haremos sobreescribiendo otros datos importantes para el programa o incluso para el sistema operativo (DOS o WINDOWS) con lo que casi seguro conseguiremos que el ordenador se quede bloqueado.
Imagina que en un programa (No de QBasic) ocurre esto y va a parar a un registro estratégico del sistema operativo un número que equivale de alguna forma a la llamada al programa de formatear el disco duro, no veas el estropicio.

De ahí la importancia de depurar los datos de entrada, especialmente los que van a servir como índices para arrays. Si en nuestro primer ejemplo no depuramos el dato del piso que pedimos al usuario y este escribe uno que no está entre cero y cuatro, al acceder al vector se produciría este error.

MATRICES

Ahora imaginemos este otro bloque de pisos, para el que tenemos que hacer un programa similar al anterior.



Vamos a escribir el programa. Será igual que el anterior, habrá que controlar las cuatro plantas pero además dentro de cada una habrá que controlar las tres puertas que hay en cada rellano (1ª, 2ª y 3ª).

```
DIM nombre$ (1 TO 4, 1 TO 3)
FOR piso = 1 TO 4
        FOR puerta = 1 TO 3
                PRINT "Nombre de quien vive en el"; piso; "º"; puerta; "ª: ";
                INPUT "", nombre$(piso, puerta)
        next
NEXT
PRINT "Para saber quien vive en un piso..."
DO
        INPUT "  Escribe el piso: ",piso
LOOP WHILE (piso < 1) OR (piso > 4)
DO
        INPUT "  Escribe la puerta: ",puerta
LOOP WHILE (puerta < 1) OR (puerta > 3)
PRINT "En el";piso;"º";puerta;"ª vive ";nombre$(piso, puerta)
```

Se puede ver que es muy parecido, pero hemos utilizado una MATRIZ, que es un array de dos dimensiones, mientras que un vector es un array de una sola dimensión. Por si hay alguna duda de lo que hace este programa vamos a ver un posible resultado.

```
Nombre de quien vive en el 1º 1ª: Paca
Nombre de quien vive en el 1º 2ª: Gloria
Nombre de quien vive en el 1º 3ª: Fernando
Nombre de quien vive en el 2º 1ª: Mari
Nombre de quien vive en el 2º 2ª: Juan
```

```
Nombre de quien vive en el 2º 3ª: Manolo
Nombre de quien vive en el 3º 1ª: Lola
Nombre de quien vive en el 3º 2ª: Rosa
Nombre de quien vive en el 3º 3ª: Mario
Nombre de quien vive en el 4º 1ª: Pepe
Nombre de quien vive en el 4º 2ª: Nacho
Nombre de quien vive en el 4º 3ª: Luisa
Para ver quien vive en un piso...
  Escribe la planta: 3
  Escribe la puerta: 2
En el 3º 2ª vive Rosa
```

Lo más novedoso es la forma de declarar la matriz, igual que el vector pero esta vez habrá que usar dos intervalos separados por una coma. Y por lo tanto siempre que nos refiramos a la matriz habrá que usar dos subíndices.

Para recorrer la matriz hay que usar dos FOR anidados. Esta vez la hemos recorrido por filas (pisos en nuestro ejemplo) ya que el "FOR piso" está fuera y hasta que no se ejecute entero el "FOR puerta" en cada piso no pasamos al siguiente. Para recorrerla por columnas (puertas) bastaría con intercambiar los FOR:

```
FOR puerta = 1 TO 3
        FOR piso = 1 TO 4
                '(...)
        next
NEXT
```

Para el ordenador es completamente intrascendente que lo hagamos de una forma u otra, él no entiende de filas horizontales ni columnas verticales, de hecho almacena todos los elementos seguidos uno detrás de otro y hace operaciones matemáticas con los dos subíndices para determinar la única posición del elemento.

Esto se puede ver fácilmente pensando en la posición de los buzones de correos en el portal del bloque de pisos.

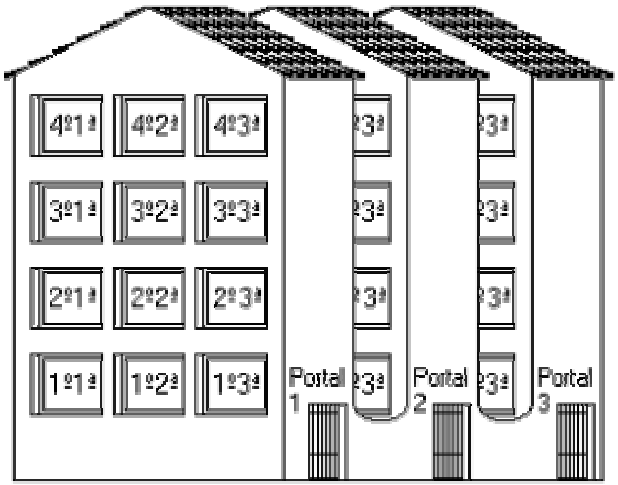
Cada piso tiene su buzón, el del 2º 2ª es el quinto porque tiene delante los tres de la primera planta y es el segundo de la segunda. Para calcular las posiciones se hace algo así como ((PLANTA-1)*Nº_total_de_PUERTAS)+PUERTA, que sale ((2-1)*3)+2 = 5.

A nosotros esto no nos interesa porque lo hace QBasic automáticamente. Si programamos en otros lenguajes más primitivos sí que habría que preocuparse de esto porque sólo existen vectores de una dimensión.

1.12.4 - POLIEDROS

Lo más normal es usar vectores (1 dimensión) o matrices (2 dimensiones), pero QBasic puede llegar a manejar arrays con hasta 60 dimensiones!!!

Veamos un caso rebuscado de un array de tres dimensiones ampliando los ejemplos anteriores de los bloques de pisos.



En este caso nuestro bloque tiene tres portales (1 a 3) y cada uno de ellos tiene cuatro plantas y tres puertas igual que antes.

Para declarar el poliedro habría que hacer:

```
DIM nombre$ (1 TO 3, 1 TO 4, 1 TO 3)
```

La primera dimensión va a ser para los portales, la segunda para los pisos y la tercera para las puertas.

Para recorrer el array haríamos:

```
FOR portal = 1 TO 3
        FOR piso = 1 TO 4
                FOR puerta = 1 TO 3
                        '(...)
                NEXT
        NEXT
NEXT
```

Esta vez tenemos tres bucles FOR anidados y las instrucciones que pongamos dentro de todo se ejecutarán 36 veces que es el producto de todas las dimensiones.

Ya empezamos a liarnos pensando en recorrer el array primero por portales o no, etc... Todavía podemos tener una representación visual del problema, pero si usamos arrays de cuatro, cinco o más dimensiones ya esto no será así y la habremos liado del todo. En estos casos lo mejor será plantear el problema de otra forma o usar estructuras de datos más flexibles que ya veremos más adelante.

AHORRAR ESPACIO EN LOS ARRAYS DE CADENAS

Hasta ahora no hemos sido demasiado estrictos en el ahorro de memoria ya que ni siquiera estamos declarando las variables. Nuestros programas de QBasic son muy sencillos y usan muy pocas variables, pero al trabajar con arrays hay que darse cuenta que basta con declarar una matriz de 20 por 20 elementos para que se gasten 400 posiciones de memoria.

En las cadenas QBasic no nos limita el número de caracteres que vamos a poder meter en ellas y para que esto funcione se gasta cierta cantidad de memoria (unos 10 bytes), que si multiplicamos por 400 posiciones de memoria son casi 4 KB, una cantidad respetable para programas tan pequeños.

Puede ocurrir que en nuestra matriz de cadenas no lleguemos a almacenar palabras más largas de por ejemplo 15 letras con los que nos convendría limitar el tamaño de las cadenas y ahorrarnos esos 10 bytes en cada posición.

Para declarar un array de cadenas de tamaño limitado haríamos:

DIM matriz(1 TO 20, 1 TO 20) AS STRING * 15

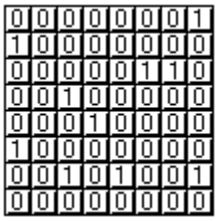
Las palabras clave AS STRING declaran explícitamente la variable como de tipo cadena (Esta será la forma normal de hacerlo en Visual Basic), por eso ya no es necesario poner el $ al final del nombre. Si a continuación ponemos un asterisco y un número estamos limitando el tamaño de las cadenas con lo que QBasic ya no tendrá que determinarlo automáticamente constantemente, con lo que ahorraremos memoria y el programa funcionará algo más rápido.

Si asignamos a estas variables una cadena de más de 15 caracteres no ocurre nada, simplemente la cadena se corta y sólo se almacenan los primeros 15 caracteres, los demás se pierden.

En los tipos numéricos no podemos ahorrar nada, pero en los arrays de cadenas deberíamos hacer siempre esto, especialmente si van a contener muchos elementos.

MATRICES ORLADAS

Imaginemos el juego del buscaminas en el que tenemos una matriz de 8 por 8 elementos (Casillas) que van a contener un 1 si contienen una mina o un cero si están vacías.



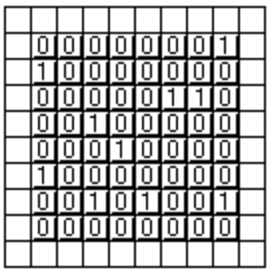
Cuando el usuario destape una casilla, si no contiene una mina, habrá que escribir el número de minas que la rodean. Para hacer esto habrá que sumar los valores de las casillas que hay arriba a la izquierda, arriba, arriba a la derecha, a la derecha, abajo a la derecha, abajo, abajo a la izquierda y a la izquierda.

Si estamos en una casilla del interior del tablero esto funciona perfectamente, pero si estamos en el borde nos encontramos con el problema de que haríamos referencia a posiciones de fuera de la matriz con lo que se produciría un error de "Subíndice fuera del intervalo".

Para evitar esto habría que comprobar que si estamos en el borde superior no se cuenten las casillas superiores, si estamos en el inferior no se cuenten las de abajo, si estamos en una esquina solo se cuenten las tres de dentro, etc. con lo que tendríamos en total nueve rutinas diferentes, así como que controlar cual de ellas usar según por donde esté la casilla que estamos destapando.

Para solucionar este problema lo que se suele hacer es dotar a la matriz de más filas y columnas de forma que todos los elementos que realmente nos interesan estén rodeados por un "borde" de elementos vacíos que estarán ahí sólo para poder hacer referencia a ellos sin salirnos de la matriz.

Una representación gráfica de la matriz orlada podría ser esta, donde hay un borde de "casillas" que no vamos a usar nunca para poner minas, pero a las que podemos referirnos siempre que sea necesario sin salirnos de la matriz. En el ejemplo que nos ocupa, estas posiciones contendrían el valor 0 ya que están vacías.



Esta técnica se aplica mucho en juegos de este tipo, en programas de imágenes y en cualquier lugar donde se tenga que acceder a posiciones de una matriz y puedan producirse problemas en los bordes. En el caso de vectores habría que añadir un elemento más al principio y otro al final, para los poliedros sería como envolverlo completamente y en el caso de arrays de más de cuatro dimensiones ya es difícil imaginarse como se haría.

Hay que recordar que en QBasic se pueden usar subíndices negativos, pero en la mayoría de otros lenguajes de programación esto no es así.

El único inconveniente de las matrices orladas es que ocupan más memoria, exáctamente (2*(alto+ancho))+4 posiciones más. Este gasto de recursos está justificado en la mayoría de los casos ya que se consigue simplificar mucho los programas.

FORMACIONES DINÁMICAS

En otros lenguajes de programación más avanzados como C o Pascal existen lo que se llaman "Estructuras de almacenamiento dinámicas" que son técnicas de programación que mediante la utilización de punteros (Variables especiales que contienen direcciones de memoria) permiten acceder directamente a la memoria del ordenador y colocar los datos en estructuras como listas enlazadas, árboles, pilas, colas, tablas hash, etc cuyo tamaño cambia constantemente durante la ejecución del programa sin estar definido de ninguna forma en el código del programa. Por eso se dice que son estructuras de almacenamiento dinámicas.

Como QBasic (ni Visual Basic) no trabaja directamente con punteros no podemos llegar a utilizar estas estructuras. En algún caso puede suceder que necesitemos almacenar elementos en un array (Vector, matriz, etc...), pero no podamos saber de antemano cuanto grande va a tener que ser y tampoco nos merezca la pena poner un límite arbitrario que se alcanzará en seguida o bien no llegará a ser usado nunca desperdiciando mucha memoria.

Para solucionar en parte este problema QBasic nos da la posibilidad de declarar los arrays usando la instrucción REDIM en lugar de DIM y después en cualquier parte del programa poder redimensionarlos tantas veces como queramos usando valores que pueden ser variables.

Veamos un ejemplo:

```
REDIM mat(1 TO 1)
INPUT "¿Cuantos números vamos a almacenar?", maxi
REDIM mat(1 TO maxi)
```

```
FOR n = 1 TO maxi
        INPUT mat(n)
NEXT
INPUT "¿Cuantos números vamos a almacenar ahora?", maxi2
REDIM mat(1 TO maxi2)
FOR n = 1 TO maxi2
        INPUT mat(n)
NEXT
```

Se puede ver que declaramos un vector de enteros de sólo un elemento usando REDIM, a continuación pedimos al usuario que escriba un número y después volvemos a redimensionar el vector con este número de elementos para ya recorrerla como hemos hecho siempre. A continuación repetimos el proceso volviendo a redimensionar el vector, perdíendose la primera serie de números que almacenó en ella el usuario.

Aquí hay que aclarar algunas cosas:

- La primera orden REDIM mat(1 to 1) nos la podíamos haber ahorrado, pero siempre es costumbre declarar las variables al principio del programa para tener una visión global de las variables que hay sin tener que recorrer el listado completo.
- Si al declarar un array con la orden DIM usamos variables en vez de valores constantes lo declaramos como dinámico y después podrá ser redimensionado.
- Para redimensionar un array dinámico ya existente habrá que hacerlo siempre con la orden REDIM y en QBasic tendremos que usar las dimensiones que ya tiene, es decir, si antes era una matriz con dos dimensiones al redimensionarlo tenemos que especificar dos intervalos, ni más ni menos, y por supuesto el mismo tipo de datos.
- Al redimensionar un array se borran todos los valores que tuviera, quedando entero inicializado a valores cero o cadenas vacías.

Visto esto sólo queda por decir que siempre que sea posible se evite el uso de arrays dinámicos ya que el manejo interno de la memoria es mucho menos eficiente y podría dar lugar a errores como los típicos de "Memoria agotada". Para evitar que los arrays sean dinámicos usar siempre valores constantes (números) para definir sus dimensiones al declararlos con la orden DIM al principio del programa.