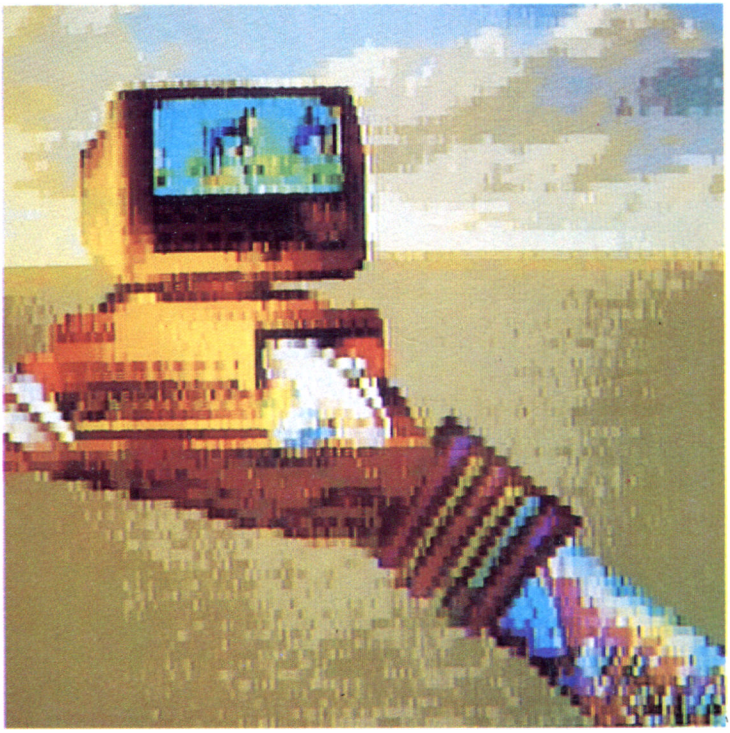
GRAN BIBLIOTECA

AMSTRAD



LENGUAJE BASIC I

EL ORDENADOR A NUESTRO SERVICIO

EDICIONES INGELEK

GRAN BIBLIOTECA AMSTRAD

8

LENGUAJE BASIC I

Director editor:
Antonio M.ª Ferrer Abelló

Director de producción:
Vicente Robles

Director de la obra:
Fernando López Martínez

Redactor técnico:
Antonio García Verdugo

Colaboradores:
Pilar Manzanera Amaro

Diseño:
Bravo/Lofish

Maquetación:
Carlos González Amezúa

Dibujos:
José Ochoa

Fotografía:
Grupo Gálata

ISBN del tomo: 84-7708-028-3
ISBN de la obra: 84-7708-004-6.
Fotocomposición: Andueza, S. A.
Imprime: Héroes, S. A.
Depósito Legal: M-43100-1986
Precio en Canarias, Ceuta y Melilla: 435 ptas.

LENGUAJE BASIC I

INTRODUCCIÓN

ntes de empezar el estudio de este popular lenguaje, conviene hacer un poco de historia. Todo empezó en 1964 tras el diseño por parte de John G. Kemeny y Thomas Kurtz de un lenguaje de programación que ellos mismos llamaron *«Beginner's All-purpose Symbolic Instruction Code»*, más conocido como BASIC. El objetivo de estos dos caballeros era conseguir un lenguaje de programación «accesible» para cualquier persona con unos determinados conocimientos; se trataba en definitiva de diseñar un lenguaje fácil (en comparación, claro está, con otros existentes entonces: FORTRAN, ALGOL, COBOL...).

No se puede negar que el objetivo de Kemeny y Kurtz se alcanzó sobradamente: casi todos los ordenadores domésticos y medios incorporan este lenguaje en ROM, o lo acompañan como software gratuito al aparato. El primer caso lo tenemos en los Amstrad CPC, los cuales acceden al BASIC al conectarlos. Por el contrario, el BASIC del Amstrad PCW 8256 se proporciona junto con el sistema y se carga en la RAM.

El lenguaje BASIC puede ser compilado o interpretado. Para simplificar las cosas, los fabricantes incluyen en sus ordenadores el lenguaje interpretado. La razón es evidente si seguimos la idea bajo la cual se concibió BASIC: ha de ser accesible a cualquier persona. Como ya sabemos, un lenguaje compilado, sea cual sea, lleva consigo unos problemas mucho mayores en cuanto a depuración y corrección, y esto se aleja del pensamiento inicial. La interpretación permite, por el contrario, el aprendizaje por «ensayo y error».

Pero no todo son elogios para este lenguaje; no al menos en lo que se refiere a su evolución. Desde que apareció, los fabricantes de ordenadores y compañías de software han creado su propio dialecto, con vistas al mercado al cual quieren acceder. Esto significa que pasar un programa de una máquina a otra no siempre será tan sencillo como parece. En casos extremos, la traducción será imposible. Vivir esta experiencia puede hacernos pensar que nunca se aprende BASIC hasta un nivel suficiente como para dominar cualquier ordenador.

Sin embargo, existe una parte «básica» del BASIC que, por lo general, no cambia de una versión a otra; son estos conocimientos mínimos los que nos permiten teclear en cualquier ordenador un pequeño programa, y lo que es más importante: razonar en BASIC, lo cual significa poder asimilar nuevas variaciones sin dificultades. Los comandos e instrucciones más comunes son, en gran parte, el contenido de este primer libro sobre el lenguaje BASIC, dedicado en particular a los ordenadores Amstrad.

Algo Sobre El Teclado

Existen varias teclas que nos serán necesarias desde el primer momento para comunicarnos con el ordenador a través de BASIC. La primera y la más importante es RETURN.

En pocas palabras, al pulsar RETURN el ordenador entiende: «haz lo que acabo de escribir». Y es que, como ya sabemos, los ordenadores son bastante tontos; tanto, que no saben cuándo hemos terminado de escribir y queremos que se interprete. Esta tecla habrá que pulsarla siempre que se termine de escribir una línea, o instrucción, o lo que sea, porque la máquina no va a leer nada hasta que se le dé una confirmación: RETURN.

Otras teclas interesantes son las de borrar (BORR). Tanto en el PCW como en los CPC el resultado de una pulsación es el borrado del carácter bajo el cursor o a su izquierda.

La corrección de una línea con un error se realiza con esas dos teclas, y con la ayuda de las flechas del cursor (a izquierda y derecha). Si hemos cometido una falta en algún lugar de la línea desplazaremos el cursor hasta allí con las flechas; a continuación borraremos la letra y escribiremos la correcta. Por último —no lo repetiremos muchas veces— se pulsa RETURN.

Estas son las indicaciones básicas acerca de los teclados Amstrad, pero son suficientes para empezar a aprender. Lo más importante no es esto, sino experimentar, probar todo lo que se dice aquí y también lo que nos venga a la cabeza en cada momento. Este libro debe abrirse frente a un ordenador conectado, y la práctica es realmente el único medio por el cual comprenderemos completamente el *cómo* del BASIC.

8.0.1. Algunos sistemas, como el PCW, incluyen su intérprete BASIC en disco.

LOS NÚMEROS EN BASIC

inguna información puede ser mostrada en pantalla sin el comando PRINT. Tan importante es que necesitamos conocerlo ahora para poder aplicar lo expuesto en este capítulo.

La forma general del comando PRINT, es decir, la forma de escribirlo para que pueda ser correctamente interpretado, es:

PRINT (datos para imprimir) <RETURN>

Este comando es tan utilizado que existe una forma abreviada de escribirlo: el signo de interrogación (?).

En esta primera parte hablaremos de números, de manera que un ejemplo adecuado es:

PRINT 40 <RETURN> o bien ? 40

El resultado en la pantalla es tan simple como se podía esperar de

una sola instrucción: un 40. Si cometemos alguna falta al escribir, BASIC mostrará su más conocido mensaje de error: Syntax error (error de sintaxis), lo cual significa que el intérprete no reconoce tal cosa como instrucción válida. Aunque provoquemos este error, no ocurre nada irreversible; simplemente se teclea de nuevo nuestra instrucción.

NOTACIÓN

Normalmente, nuestro Amstrad utiliza los números como nosotros lo haríamos: 30, 387493, −37643, etc... Igualmente ocurre con los decimales, aunque la coma decimal se representa con un punto: 34.2, 3.1415, 2.171828, etc... Un detalle importante es que no podemos distinguir los millares con puntos: en BASIC, mil no es 1.000, sino 1000.

El problema surge cuando necesitamos representar números demasiado grandes o demasiado pequeños. Acudiremos entonces a la *notación exponencial,* la cual consiste en multiplicar por potencias de 10 para evitar excesivos ceros. Así, mil millones (escrito normalmente 1000000000) para BASIC es 1E+09. ¿Qué significa esto? En notación científica diríamos 1 por 10 elevado a 9 (es decir, un uno seguido de nueve ceros).

Lo mismo ocurre con números demasiado pequeños; por ejemplo, 0.000000001 será para BASIC 1E−09, esto es, 1 por 10 elevado a −9. BASIC pasa a notación exponencial en cuanto comprueba que el número de cifras es demasiado grande para sus características internas. Estas, evidentemente, varían de una máquina a otra: el PCW, por ejemplo, aceptará esos mil millones en notación normal; hay que añadir más ceros para que pase a la exponencial.

Es muy posible que no utilicemos esta notación casi nunca, pero aun así debemos conocerla.

LAS OPERACIONES DE BASIC

Este lenguaje se comporta perfectamente como una calculadora. Además de las operaciones más frecuentes encontraremos las trigonométricas y otras más específicas, como el cociente y resto de una división. Vamos a repasar todas con unos ejemplos. La multiplicación se representa con un asterisco (*) y la división con una barra inclinada a la derecha (/).

```
print 97-365
print 34.234+348.3
print 24*7
print 365/12
```

Las potencias se expresan con el símbolo de flecha vertical (↑),el cociente de la división, o división entera, se obtiene con la barra inclinada a la izquierda (\ ) y el resto de la división con MOD:

```
print 2↑8 da 256 (2 elevado a 8).
print 7\2 es 3.
print 7 mod 2 es 1 (7 entre 2 es 3, de resto 1).
```

Es posible agrupar las operaciones en una misma línea, pero esto representa un problema: ¿Por dónde empieza BASIC a operar? Podemos utilizar paréntesis para agrupar las operaciones según nuestro deseo. Así, en la expresión:

```
print 4+(2*(4+((1/8)-4)))
```

BASIC comenzará a operar por el paréntesis más anidado. El proceso sería más o menos así:

```
4+(2*(4+((1/8)-4)))      ; (1/8) es 0.125
4+(2*(4+(0.125-4)))      ; (0.125-4) es -3.875
4+(2*(4-3.875))          ; (4-3.875) es 0.125
4+(2*0.125)              ; (2*0.125) es 0.25
4+0.25                   ; 4+0.25 es 4.25
```

BASIC entrega el resultado 4.25.

Esto puede llegar a ser muy complicado. Además, BASIC establece un *orden de prioridades*. Esto significa que en el caso de dos operaciones no agrupadas por paréntesis una de ellas se efectúa antes. El orden de prioridades varía de una máquina a otra, aunque nunca de manera muy significativa.

No termina aquí la capacidad de este lenguaje para operar con números. Existen varias *funciones;* en su mayoría, tienen la siguiente forma general:

```
función (argumento)
```

La raíz cuadrada *(SQuare Root)* se escribe SQR(n), siendo n un número positivo. Si introducimos un argumento negativo, BASIC de-

PRIORIDAD DE OPERADORES EN LOS PCW

PRIORIDAD	OPERADOR	TIPO DE OPERACION
1	^	Potenciación
2	-	Cambio de signo
3	*	Multiplicación
3	/	División
4	\	Cociente de la división
5	MOD	Resto de la división
6	+	Suma
6	-	Resta

PRIORIDAD DE OPERADORES EN LOS CPC

PRIORIDAD	OPERADOR	TIPO DE OPERACION
1	^	Potenciación
2	MOD	Resto de la división
3	-	Cambio de signo
4	*	Multiplicación
4	/	División
5	\	Cociente de la división
6	+	Suma
6	-	Resta

vuelve el error Improper argument (argumento inadecuado). La raíz cuadrada de 2 (1.4142) ó 9 (3) se calculan con:

```
print sqr(2)
print sqr(9)
```

Las funciones trigonométricas seno, coseno y tangente se obtienen con SIN(n), COS(n), TAN(n). El argumento se expresa —normalmente— en radianes, aunque los Amstrad CPC permiten hacerlo en grados con la orden DEG. Si la cantidad indicada entre los paréntesis es excesiva, y dado que estas funciones son periódicas, se produce el error Improper argument.

BASIC dispone además de logaritmos, en base 10 (LOG10(n)) y en base e (LOG(n)). Estas funciones admiten argumentos entre 0 y el límite superior propio de la máquina —normalmente 1e+38—. La función EXP(n) da el valor de e elevado a la potencia dada. Su argumento ha de estar entre −88 y 88, aproximadamente.

Existen constantes imprescindibles en algunos cálculos; BASIC

dispone de dos: el número e (2.7182818) se obtiene con print exp(1), y pi, el cual está incluido en el CPC y obtiene con print pi, cuyo valor es bien conocido por todos: 3.14... En el PCW se puede conseguir un valor muy aproximado de pi con la expresión print 4*atn(1) (casi perfecto).

Por último, existen cinco funciones «de ayuda» en los cálculos: INT(n) entrega el número redondeado al entero inferior más próximo: print int(−1.9999) devuelve −2.

FIX(n) elimina la parte decimal: print fix(−1.9999) da −1. ROUND(n) redondea al número entero más próximo:

print round (2.5001) da 3, pero
print round (2.5) da 2.

SGN(n) es un indicador de signo del argumento especificado. Si el argumento es mayor que 0, SGN devuelve 1, si el argumento es 0, SGN no lo modifica, si es negativo SGN da −1.

ABS(n) devuelve el valor absoluto del argumento. Por ejemplo, print abs(−34.5) dará 34.5. Esta función es útil cuando, por ejemplo, queremos calcular una raíz cuadrada cuyo argumento es negativo pero aceptamos como válido éste.

Con todas estas funciones de BASIC disponemos de una auténtica calculadora. Pero, como veremos en el capítulo 3, podemos completarla con las *variables*.

LÓGICA NUMÉRICA

El ordenador (aunque de una manera muy simple) sabe distinguir entre «verdad» y «mentira». A estos valores lógicos BASIC les asigna unos numéricos. En la mayoría de las versiones BASIC estos números son el 1 para «verdadero» y 0 para «falso». Pero en el BASIC Amstrad es un −1 (negativo) el «verdadero». Esto realmente no importa, el caso es que BASIC reconozca los valores que él mismo adopta, y eso, sin duda, lo hace perfectamente.

Las operaciones lógicas son de comparación: una cosa es igual a otra o no, mayor, menor, etc... Así, sabemos que 4 es igual a 4 (elemental, digamos) y para saber qué opina sobre esto el ordenador escribiremos:

```
print 4=4
−1
```

−1 es el valor numérico que indica «verdadero», luego el Amstrad también opina que 4 es igual a 4.

Las desigualdades matemáticas son: mayor estrictamente, mayor o igual, menor estrictamente, menor o igual y distinto. Todas estas funciones se representan en BASIC como >, >=, <, <=, <>, respectivamente. Además, los signos compuestos admiten variar el orden de los signos, de manera que BASIC entiende que:

Mayor o igual es >= o bien =>
Distinto es <> o bien ><
Menor o igual es <= o bien =<

Para comprobar todo esto lo mejor es teclear varios ejemplos:

print 7>=3 da −1, porque 7 es mayor que 3
print 4>5 da 0 porque 4 no es mayor que cinco
print (3=3)=(7>2) da −1 porque 3 es igual a 3, 7 es mayor que 2 y cierto es igual a cierto.

Los «otros» operadores lógicos de BASIC son AND, OR y XOR, cuyo funcionamiento se estudió detenidamente en el primer número de esta colección. Tan sólo diremos que en BASIC se pueden utilizar normalmente:

print 4>2 and 9=(18/2) da −1, puesto que las dos condiciones son verdaderas.
print 8=9 or 100=>2 da −1, porque la segunda condición es cierta.
print −1 xor −1 da 0 porque los dos operandos son «verdaderos».

El dominio de la lógica numérica será de gran importancia más adelante, ya que BASIC podrá basarse en ella para realizar determinadas operaciones.

EJERCICIOS

1. La única forma equivalente de escribir PRINT es:

A) Otro PRINT.
B) WRITE.
C) ?.

2. ¿Cómo se escribe el número 4E4 en notación normal?

A) 40000.
B) 4.4444.
C) 8.

3. ¿Y 2E−2?

A) 0.02.
B) 0.
C) 091.

4. Las operaciones (4*3)/2 y 4*(3/2), ¿dan el mismo resultado?

A) Sí.
B) No, están cambiados los paréntesis.
C) No, se provoca el error «Syntax error».

5. ¿Cuál es el resultado de 1+2↑2?

A) 5.
B) 9.
C) 0.

6. ¿Es lo mismo INT(−2.1) que FIX(−2.1)?

A) No, se diferencian en la F, la X, la T y la N.
B) No, INT redondeará.
C) Sí.

7. El resultado de escribir PRINT "Pepito"="Pepito" es:

A) Pepito.
B) 0.
C) Tener más letras en la pantalla.

8. Los valores que utilizan los Amstrad para denominar los estados falso y verdadero son, respectivamente:

A) FALSE y TRUE.
B) 0 y −1.
C) Syntax error y Ready.

9. La expresión PRINT 4="frambuesas", ¿qué error provoca?

A) Syntax error.
B) Type mismatch.
C) CP/M error on A: Invalid ice-cream flavour - Retry, Ignore or Cancel.

CADENAS

Toda la información almacenada en un ordenador puede clasificarse en numérica y alfanumérica. Hasta ahora hemos visto la primera. Los datos alfanuméricos tienen en BASIC un tratamiento especial: se indican entre comillas (") y suelen recibir el nombre de cadenas o *strings*.

Todo lo almacenado en una cadena —ya sean cifras o caracteres— no tienen significado especial para BASIC, es decir, no tienen ningún valor, al contrario que cualquier número. Pero esto no significa que no podamos operar con cadenas; existen muchas funciones para ello. Normalmente, todo lo que esté relacionado en BASIC con las cadenas incluye el signo $. Más adelante utilizaremos este signo.

Para empezar, vamos a ver varios ejemplos de cadenas:

"abcdefghijklmnopqrstuvwxyz"
"348"
"print 40"
"Esto es un ejemplo; Numaios, hermano mio"
""

Es necesario tener en cuenta que una cadena NO tiene valor numérico para BASIC. El segundo ejemplo NO VALE 348, sino que simplemente ES "348", sin más (un tres seguido de un cuatro y un ocho). Algo parecido ocurre con el tercer ejemplo. Esa cadena NO significa nada para BASIC, y si tecleamos en nuestro Amstrad print "print 40" obtendremos en la pantalla la cadena, que no ha sido interpretada: *print 40*. El último ejemplo es algo especial. Se trata de la «cadena vacía», es decir, aquella que no contiene ningún carácter. Se puede operar con ella normalmente.

Las comillas reciben un trato algo especial. ¿Cómo escribirlas? Por ejemplo, supongamos la cadena *Chus dijo "Hola"*. Debe ir entre comillas en el PRINT, de modo que la escribiríamos así:

print "Chus dijo "Hola""

El resultado en pantalla es:

Chus dijo 0

Todavía no sabemos a qué se debe esta respuesta tan «loca»; en cualquier caso, baste por ahora con saber que BASIC no ha aceptado la cadena completa, y eso es porque al llegar a las comillas de "Hola" ha entendido que allí acaba la cadena y el resto lo ha interpretado de otra manera.

La solución para que BASIC no se confunda con las comillas es tomar dos como una dentro de las cadenas. Así, nuestra cadena es ahora *Chus dijo ""Hola""*, y en un PRINT quedaría:

print "Chus dijo ""Hola"""

El resultado, esta vez sí, será:

Chus dijo "Hola"

Podemos operar con cadenas. Comenzaremos por algo tan sencillo como la suma. No con el mismo sentido que en los números, pero también se trata de una «suma». Probemos a sumar dos cadenas:

```
print ."estamos sum"+"ando cadenas"
print "123"+"7"
```

El último ejemplo da como resultado 1237. Las comillas quitan el significado numérico a cualquier cifra (es el viejo truco de "¿cuánto es uno más uno?: ¡11!).

Evidentemente, la resta no está definida en BASIC. Si intentamos escribir print "hola"-"Adios" BASIC mostrará el mensaje Type mismatch (error entre tipos: "hola" y "Adios" son operadores de tipo alfanumérico, y la operación resta solamente admite números). Tampoco aconsejamos multiplicar, dividir, elevar cadenas... puesto que obtendremos el mismo resultado (de paso, desafiamos a cualquiera a prever un resultado para tales operaciones).

RECORTAR, AÑADIR Y BUSCAR

Las funciones que permiten el fraccionamiento de cadenas son RIGHT$(c,v), LEFT$(c,v) y MID$(c,v,v2), donde *c* es una cadena y *v* es un valor entero no negativo.

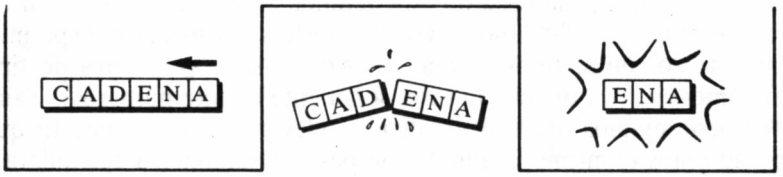
«Right» significa derecha y la función RIGHT$ fracciona las cadenas precisamente por el lado derecho. Un ejemplo: print right$("Eso es todo amigos",6) da el resultado *amigos*. El proceso de BASIC ha sido: primero, «coger» la cadena por la derecha. Luego, contar 6 caracteres y, por último, pegarle un tijeretazo por allí. Luego print ha tomado el resultado, *amigos,* para imprimirlo en la pantalla.

Podemos imaginar ya la operación que efectúa LEFT$ (lo cual significa izquierda):

print left$("Eso es todo, amigos",12)

Dará *Eso es todo,*. Para BASIC solamente cambia el extremo para empezar a contar.

8.2.1. La función LEFT$ fija la subcadena resultado a partir de la izquierda de la cadena operando.



8.2.2. La función RIGHT$ fija la subcadena resultado a partir de la derecha de la cadena operando.

La función MID$ (de MIDdle, mitad) es algo más complicada, pero nos hace falta en el caso de que el trozo de cadena que queramos obtener no esté en los extremos, sino en el centro. MID$ tiene tres argumentos. El primero es la cadena sobre la cual operaremos, el segundo indica el primer carácter de nuestra porción y el tercero el número de caracteres de la sub-cadena. Esto suena muy complicado, pero no es así. Vamos con un ejemplo:

```
print mid$("Quiero la mitad de la cadena",8,8)
```

Devuelve *la mitad.* Seguiremos el camino que ha tomado BASIC: primero, coger la cadena por el extremo izquierdo y contar ocho caracteres. Al llegar, pega un corte, tira el extremo izquierdo y sigue contando a la derecha, también ocho caracteres. De nuevo corta y tira el extremo derecho. print mid$("hola",2,1) da *o*.

La función de búsqueda es INSTR(c,c2). En esta operación tenemos una cadena (c2) en la cual se busca la existencia de un fragmento (c). Si se encuentra c dentro de c2, la función devuelve la posición en la que lo ha hallado. Si no lo consigue, devuelve 0.

Como ejemplo, busquemos el fragmento "el" dentro de la cadena "Ya veo el truco!". Lo escribiremos así:

```
print instr("el","Ya veo el truco!")
```

La función devuelve 8. Esto significa: primero, que INSTR ha encontrado la inclusión de la primera cadena; y segundo, que se encuentra en la posición ocho de la cadena sobre la cual se ha buscado. El resultado opuesto sería:

print instr("si","esto es inútil")

Que da 0 (INSTR no ha encontrado nada).

Una variante de INSTR permite incluir el carácter a partir del cual ha de iniciarse la búsqueda. La función se escribe entonces como INSTR(v,c,c2), siendo v dicho valor.

La última de las funciones más usuales es LEN. LEN devuelve la longitud de la cadena especificada, así:

print len("hola") da 4.
print len("") da, evidentemente, 0.

Los Auxiliares Para Operar

Las funciones que hemos visto hasta ahora bastan para arreglárselas con casi todo. Pero existen otras: UPPER$, LOWER$ y STRING$. UPPER$ devuelve su argumento en mayúsculas. LOWER$, al contrario, en minúsculas:

print lower$("Hola, CPC") da *hola, cpc.*
print upper$("Hola, CPC") da *HOLA, CPC.*

Con STRING$ podemos generar cadenas que contengan una colección del mismo carácter. Se escribe como STRING$(v,c), donce *c* es el carácter para repetir y *v* la cantidad de veces que se repite. Por ejemplo, queremos generar una cadena con 20 «aes»:

print string$(20,"a")

¿Qué ocurre si damos más de un carácter? Simplemente, se ignoran los siguientes:

print string$(3,"abc") da *aaa* y no *abcabcabc*

Podemos anidar operaciones de cadenas tal y como lo hacíamos en las numéricas. Esta vez no hay orden de prioridad, por lo que todo se efectuará por lo establecido en los paréntesis o simplemente operando de izquierda a derecha.

print string$(4,upper$(left$
("hola,"+"amigo",1))) da *HHHH*

DAR VALOR... O QUITARLO

¡Sorpresa! Una cadena puede VALER, es decir, unas cifras almacenadas en ella pueden utilizarse como valor numérico. También podemos operar en sentido contrario: tomar un valor y convertirlo en una cadena.

Vamos con lo primero. Tenemos una cadena, por ejemplo *"565"*. La función para convertirla a número es VAL(c), donde *c* es la cadena.

```
print val("565")
```

El resultado en la pantalla es *565*, lo cual no indica realmente si se trata de un valor. Para comprobarlo realmente, podemos OPERAR con ello:

```
print val("565")+30
```

Obtenemos un *595*, es decir, que funciona.

Existe algún problema, claro; si la cadena no empieza por una cifra, o no contiene ninguna, VAL devuelve 0:

```
print val("Esto no vale nada, Cocoliso")
```

Es 0.

También podemos convertir nuestro número (565) en una cadena. Esto se consigue con STR$(v), donde v es el valor a convertir.

```
print str$(450)
```

Ocurre algo parecido a lo que nos pasó antes; en la pantalla aparece un 450, pero no sabemos si es numérico o no. Para comprobarlo, vamos a sumarlo con otro valor:

```
print str$(450)+100
```

¿550? No. El ordenador nos indica un error entre tipos, *type mismatch*. Esto significa que el resultado de STR$ es una cadena.

Al contrario que VAL, STR$ admite cualquier clase de expresión; puesto que el destino es una cadena, y allí no importa lo que salga, porque no VALE, sino que ES. Así, la expresión:

```
print str$(1e20)
```

Produce una cadena cuyo contenido es *ie20.*

LA TABLA ASCII

«American Standard Code for Information Interchange» (código estándar americano para intercambio de información). Este código recoge los símbolos (alfabeto, signos, etc...) a los cuales asigna un número. Por ejemplo, el espacio (" ") tiene el código 32, y la "A" el 65. La tabla ASCII figura en todos los manuales de instrucciones de Amstrad.

DEC	OCTAL	HEX	ASCII characters
0	000	00	NUL ((CTRL)A)
1	001	01	SOH ((CTRL)B)
2	002	02	STX ((CTRL)C)
3	003	03	ETX ((CTRL)C)
4	004	04	EOT ((CTRL)D)
5	005	05	ENQ ((CTRL)E)
6	006	06	ACK ((CTRL)F)
7	007	07	BEL ((CTRL)G)
8	010	08	BS ((CTRL)H)
9	011	09	HT ((CTRL)I)
10	012	0A	LF ((CTRL)J)
11	013	0B	VT ((CTRL)K)
12	014	0C	FF ((CTRL)L)
13	015	0D	CR ((CTRL)M)
14	016	0E	SO ((CTRL)N)
15	017	0F	SI ((CTRL)O)
16	020	10	DLE ((CTRL)P)
17	021	11	DC1 ((CTRL)Q)
18	022	12	DC2 ((CTRL)R)
19	023	13	DC3 ((CTRL)S)
20	024	14	DC4 ((CTRL)T)
21	025	15	NAK ((CTRL)U)
22	026	16	SYN ((CTRL)V)
23	027	17	ETB ((CTRL)W)
24	030	18	CAN ((CTRL)X)
25	031	19	EM ((CTRL)Y)
26	032	1A	SUB ((CTRL)Z)
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑
95	137	5F	_
96	140	60	
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~

8.2.3. Tabla ASCII para CPC.

Decimal	Hexadecimal	Símbolo	Descripción	Segundo significado
0	#00	∞	infinito	control-@
1	#01	⊙	flecha saliente del papel	control-A
2	#02	Γ	gamma mayúscula	control-B
3	#03	Δ	delta mayúscula	control-C
4	#04	⊗	flecha entrante en el papel	control-D
5	#05	×	multiplicar	control-E
6	#06	÷	dividir	control-F
7	#07	∴	por consiguiente	control-G
8	#08	Π	pi mayúscula	control-H
9	#09	↓	flecha abajo	control-I
10	#0A	Σ	sigma mayúscula	control-J
11	#0B	←	flecha a la izquierda	control-K
12	#0C	→	flecha a la derecha	control-L
13	#0D	±	más menos	control-M
14	#0E	↔	flecha izquierda y derecha	control-N
15	#0F	Ω	omega mayúscula	control-O
16	#10	α	alfa minúscula	control-P
17	#11	β	beta minúscula	control-Q
18	#12	γ	gamma minúscula	control-R
19	#13	δ	delta minúscula	control-S
20	#14	ϵ	épsilon minúscula	control-T
21	#15	θ	teta minúscula	control-U
22	#16	λ	lambda minúscula	control-V
23	#17	μ	mu minúscula	control-W
24	#18	π	pi minúscula	control-X
25	#19	ϱ	rho minúscula	control-Y
26	#1A	σ	sigma minúscula	control-Z
27	#1B	τ	tau minúscula	control-[
28	#1C	ϕ	fi minúscula	control-\
29	#1D	χ	ji minúscula	control-]
30	#1E	ψ	psi	control-↑
31	#1F	ω	omega minúscula	control-_
32	#20		espacio	
33	#21	!	cerrar admiración	
34	#22	"	comillas	
35	#23	Pt	Peseta	
36	#24	$	dólar	
37	#25	%	por ciento	
38	#26	&	etcétera ('ampersand')	
39	#27	'	apóstrofo	
40	#28	(	abrir paréntesis	
41	#29	)	cerrar paréntesis	
42	#2A	*	asterisco	
43	#2B	+	más	

Decimal	Hexadecimal	Símbolo	Descripción
44	#2C	,	coma
45	#2D	–	menos
46	#2E	.	punto
47	#2F	/	barra
48	#30	0	cero
49	#31	1	uno
50	#32	2	dos
51	#33	3	tres
52	#34	4	cuatro
53	#35	5	cinco
54	#36	6	seis
55	#37	7	siete
56	#38	8	ocho
57	#39	9	nueve
58	#3A	:	dos puntos
59	#3B	;	punto y coma
60	#3C	<	menor
61	#3D	=	igual
62	#3E	>	mayor
63	#3F	?	cerrar interrogación
64	#40	@	arroba ('at')
65	#41	A	A mayúscula
66	#42	B	B mayúscula
67	#43	C	C mayúscula
68	#44	D	D mayúscula
69	#45	E	E mayúscula
70	#46	F	F mayúscula
71	#47	G	G mayúscula
72	#48	H	H mayúscula
73	#49	I	I mayúscula
74	#4A	J	J mayúscula
75	#4B	K	K mayúscula
76	#4C	L	L mayúscula
77	#4D	M	M mayúscula
78	#4E	N	N mayúscula
79	#4F	O	O mayúscula
80	#50	P	P mayúscula
81	#51	Q	Q mayúscula
82	#52	R	R mayúscula
83	#53	S	S mayúscula
84	#54	T	T mayúscula
85	#55	U	U mayúscula
86	#56	V	V mayúscula
87	#57	W	W mayúscula
88	#58	X	X mayúscula
89	#59	Y	Y mayúscula
90	#5A	Z	Z mayúscula

Decimal	Hexadecimal	Símbolo	Descripción
91	#5B	¡	abrir admiración
92	#5C	Ñ	Ñ mayúscula
93	#5D	¿	abrir interrogación
94	#5E	↑	flecha arriba
95	#5F	_	subrayado
96	#60	`	acento grave
97	#61	a	A minúscula
98	#62	b	B minúscula
99	#63	c	C minúscula
100	#64	d	D minúscula
101	#65	e	E minúscula
102	#66	f	F minúscula
103	#67	g	G minúscula
104	#68	h	H minúscula
105	#69	i	I minúscula
106	#6A	j	J minúscula
107	#6B	k	K minúscula
108	#6C	l	L minúscula
109	#6D	m	M minúscula
110	#6E	n	N minúscula
111	#6F	o	O minúscula
112	#70	p	P minúscula
113	#71	q	Q minúscula
114	#72	r	R minúscula
115	#73	s	S minúscula
116	#74	t	T minúscula
117	#75	u	U minúscula
118	#76	v	V minúscula
119	#77	w	W minúscula
120	#78	x	X minúscula
121	#79	y	Y minúscula
122	#7A	z	Z minúscula
123	#7B	¨	diéresis
124	#7C	ñ	Ñ minúscula
125	#7D	}	cerrar llaves
126	#7E	~	tilde
127	#7F	0	cero sin barra
128-159	#80-#9F		códigos expansibles
160	#A0	ª	a voladita
161	#A1	º	o voladita
162	#A2	°	grados
163	#A3	£	libra
164	#A4	©	copyright
165	#A5	¶	calderón
166	#A6	§	párrafo
167	#A7	†	cruz

Decimal	Hexadecimal	Símbolo	Descripción
168	#A8	¼	un cuarto
169	#A9	½	un medio
170	#AA	¾	tres cuartos
171	#AB	«	abrir comillas
172	#AC	»	cerrar comillas
173	#AD	#	n.° ('hashmark')
174	#AE	]	cerrar corchetes
175	#AF	[	abrir corchetes
176	#B0	*f*	florín
177	#B1	¢	centavo
178	#B2	{	abrir llaves
179	#B3	´	acento agudo
180	#B4	^	acento circunflejo
181	#B5	‰	por mil
182	#B6	⅛	un octavo
183	#B7	⅜	tres octavos
184	#B8	⅝	cinco octavos
185	#B9	⅞	siete octavos
186	#BA	β	doble S alemana
187	#BB	○	círculo
188	#BC	●	topo
189	#BD	¥	yen
190	#BE	®	marca registrada
191	#BF	™	marca comercial
192	#C0	Á	A mayúscula agudo
193	#C1	É	E mayúscula agudo
194	#C2	Í	I mayúscula agudo
195	#C3	Ó	O mayúscula agudo
196	#C4	Ú	U mayúscula agudo
197	#C5	Â	A mayúscula circunflejo
198	#C6	Ê	E mayúscula circunflejo
199	#C7	Î	I mayúscula circunflejo
200	#C8	Ô	O mayúscula circunflejo
201	#C9	Û	U mayúscula circunflejo
202	#CA	À	A mayúscula grave
203	#CB	È	E mayúscula grave
204	#CC	Ì	I mayúscula grave
205	#CD	Ò	O mayúscula grave
206	#CE	Ù	U mayúscula grave
207	#CF	Ÿ	Y mayúscula diéresis
208	#D0	Ä	A mayúscula diéresis
209	#D1	Ë	E mayúscula diéresis
210	#D2	Ï	I mayúscula diéresis
211	#D3	Ö	O mayúscula diéresis
212	#D4	Ü	U mayúscula diéresis
213	#D5	Ç	C mayúscula con cedilla
214	#D6	Æ	diptongo AE mayúscula

Decimal	Hexadecimal	Símbolo	Descripción
215	#D7	Å	A mayúscula círculo
216	#D8	Ø	O mayúscula barra
217	#D9	\	barra a la izquierda
218	#DA	Ã	A mayúscula tilde
219	#DB	Õ	O mayúscula tilde
220	#DC	⩾	mayor o igual
221	#DD	⩽	menor o igual
222	#DE	≠	distinto
223	#DF	≃	aproximadamente igual
224	#E0	á	A minúscula agudo
225	#E1	é	E minúscula agudo
226	#E2	í	I minúscula agudo
227	#E3	ó	O minúscula agudo
228	#E4	ú	U minúscula agudo
229	#E5	â	A minúscula circunflejo
230	#E6	ê	E minúscula circunflejo
231	#E7	î	I minúscula circunflejo
232	#E8	ô	O minúscula circunflejo
233	#E9	û	U minúscula circunflejo
234	#EA	à	A minúscula grave
235	#EB	è	E minúscula grave
236	#EC	ì	I minúscula grave
237	#ED	ò	O minúscula grave
238	#EE	ù	U minúscula grave
239	#EF	ÿ	Y minúscula diéresis
240	#F0	ä	A minúscula diéresis
241	#F1	ë	E minúscula diéresis
242	#F2	ï	I minúscula diéresis
243	#F3	ö	O minúscula diéresis
244	#F4	ü	U minúscula diéresis
245	#F5	ç	C minúscula con cedilla
246	#F6	æ	diptongo AE minúscula
247	#F7	å	A minúscula círculo
248	#F8	ø	O minúscula barra
249	#F9	|	barra vertical
250	#FA	ã	A minúscula tilde
251	#FB	õ	O minúscula tilde
252	#FC	⇒	flecha doble a la derecha
253	#FD	⇐	flecha doble a la izquierda
254	#FE	⇔	flecha doble izquierda/derecha
255	#FF	≡	equivalente

BASIC incluye dos funciones para obtener un símbolo a partir de su número, o el número al cual pertenece el símbolo. La primera operación se consigue con CHR$(n), donde *n* es el número de código, comprendido entre 0 y 255. Si queremos obtener el carácter correspondiente al número 45, teclearemos:

```
print chr$(45)
```

Que dará el signo menos (−).

Para obtener el código de un carácter se utiliza ASC("c"), donde c es un carácter. Con...

```
print asc("S")
```

Obtendremos 83. Con...

```
print chr$(83)
```

Obtendremos, precisamente, la «S». Como los códigos están comprendidos entre 0 y 255, BASIC rechazará cualquier otro valor emitiendo el mensaje "Improper argument" (argumento inapropiado).

Lógica Con Cadenas

Para BASIC también es cierto que una cadena es igual a otra o no, y también admite operar lógicamente con "verdades" o "falsedades" derivadas de la operación con ellas a través de los operadores AND, OR, NOT y XOR.

Pero respecto a las expresiones mayor y menor (estrictamente o no) hay una gran diferencia. Para BASIC, una cadena no es mayor que otra por su longitud, sino por su clasificación dentro de la tabla ASCII, es decir, por el número que les corresponda en la tabla.

Así, "Hola" es mayor que "Adios", porque la "A" va en el código ASCII antes que la "H". "Hola" es mayor que "Hadios", porque la «o» va después que la «a».

Conviene tener esto muy en cuenta; nos hará falta para cuando debamos ordenar alfabéticamente una lista de cadenas.

EJERCICIOS

10. ¿Cuál de las siguientes cadenas es correcta?

A) "Huyamos" —dijo— "Esto se pone feo".
B) """"Huyamos"" —dijo— ""esto se pone feo""".
C) Tranquilo, estás bien aquí.

11. Al escribir PRINT "abc"-"AB", en la pantalla obtendremos:

A) Type mismatch.
B) "C".
C) C.

12. ¿Qué obtendremos con PRINT RIGHT$("mi mama me ama",7)?

A) Una madre cariñosa.
B) mi mama.
C) me ama.

13. Para buscar el carácter "o" en la cadena "Biblioteca Amstrad", teclearemos:

A) print instr("Biblioteca Amstrad","o").
B) print instr("o","Biblioteca Amstrad").
C) print chr$(o),str$(Biblioteca Amstrad).

14. ¿Qué es la cadena vacía?

A) La que no contiene ningún carácter.
B) La que no tiene eslabones.
C) La que está compuesta sólo por un par de comillas.

15. ¿Cuál de las siguientes afirmaciones es cierta?

A) Las cadenas no tienen valor numérico.
B) Las cadenas siempre tienen valor numérico implícito, pero para utilizarlo hace falta sacarlo con VAL.
C) Las cadenas siempre tienen valor, a no ser que lo inutilicemos con STR$.

16. ¿Cuál de las siguientes expresiones dará un número distinto de 0?

A) print val("Busque, compare, y si encuentra algo mejor usted se lo pierde").
B) print val("123 responda otra vez").
C) print str$(512−2 ↑ 9).

LAS VARIABLES

Operar no siempre es tan sencillo como entregar dos valores para obtener un resultado. Además, es necesario almacenar tanto datos como soluciones, y ese es el trabajo de las variables.

No es fácil explicar el funcionamiento de las variables; incluso es difícil decir qué es una variable. Vamos a basarnos en un ejemplo simple, pero que nos parece eficaz.

Nuestro ordenador es un gran criadero de loros. No son loros normales; para empezar caben en un ordenador. Todos ellos saben aprender y repetir una sóla cosa. Bien, pues ahora queremos que un loro nos recuerde el valor 32. Cogemos a uno de ellos, y para distinguirlo de los demás le llamamos Pirulo. ¿Por qué? Pues porque sí, porque queremos llamarlo así. Una vez asignado su nombre, le decimos: *32*. En ese momento el loro memoriza el 32 y cuando le llamemos el plumífero nos lo dirá.

Sencillo, ¿no? Este loro es, simplemente, una variable. Evidentemente, yo puedo hacer que Pirulo aprenda cualquier otra cosa. Por

ejemplo, una cadena. También puedo coger otros loros y, tras darles nombre, enseñarles lo que quiera.

Hay una condición evidente: si ya tengo a Pirulo con su 32, y cojo a otro loro e intento llamarle también Pirulo vendrá el antiguo para aprender lo que diga y no el nuevo. Además, puedo dar un gran golpe dentro del criadero, con lo que todos los loros olvidarán su nombre y lo que habían aprendido debido al susto (más adelante veremos cómo se da el *gran golpe).*

Traducido a lo que ya conocemos del BASIC, podremos distinguir dos clases de variables: numéricas y alfanuméricas:

```
pirulo=32
```

No ocurre nada. Bien, acabamos de elegir entre miles de loros sin nombre a Pirulo. El recuerda ahora su valor, 32. Comprobémoslo con esta línea:

```
print pirulo da 32.
```

Al escribir *pirulo,* no queremos, evidentemente, que salga el loro. Lo que nos interesa es lo que le hemos enseñado, o sea, el 32.

Si llamamos a un loro numérico al cual no habíamos enseñado ningún número, nos dirá *cero*. De la misma manera, si llamamos a un loro alfanumérico que no ha aprendido nada, no dirá nada, es decir, dará la cadena vacía ("").

Con lo explicado hasta ahora, podemos almacenar los datos de una operación. Por ejemplo:

```
suma=45+323
resultado=suma/2
print resultado
```

Como comprobaremos, la cantidad de loros disponible es inmensa, por lo que nunca nos veremos apurados. Pero, eso sí, existen ciertas normas para llamar a los loros (empezaremos a partir de ahora a llamarlos variables). Para empezar, BASIC debe distinguir entre instrucciones y variables, por lo que sería imposible una asignación del tipo:

```
print=40
```

Porque PRINT es un comando de BASIC, y para evitar confusión

se impide dicha posibilidad: el ordenador imprime el error «Syntax error»: un loro no se puede llamar Print.

Otras condiciones son:

— El nombre de la variable puede contener cifras, pero no como primer carácter; allí debe haber una letra.

— No se puede incluir signos de ningún tipo, excepto puntos, y tampoco como primer carácter del nombre.

— No puede haber espacios.

— Mayúsculas y minúsculas tienen el mismo significado; PIruLO y pirulo son el mismo loro.

— BASIC distingue perfectamente una variable numérica de otra alfanumérica, con el mismo nombre: *var* y *var$* son cosas distintas.

Dadas estas condiciones, aunque no podamos asignar a una variable el nombre *print,* sí podremos al menos darle uno muy parecido, por ejemplo p.rint, pr.int. etc.

Podemos operar con variables alfanuméricas:

```
recuerda$="Hoy hace un tiempo magnífico, María Luisa".
otro$="¿No crees?"
print recuerda$+otro$
```

```
puntero=10
frase$="Esta tecla no va"
frase2$=left$(frase$,puntero)
print frase2$
```

Si, dentro de una función como left$ especificamos una variable en lugar de una cadena no pondremos las comillas: en ese caso, BASIC entenderá que la cadena es precisamente lo entrecomillado; en el ejemplo, left$("frase$",puntero) opera sobre la cadena *frase$*, mientras que left$(frase$,puntero) opera sobre *Esta tecla no va,* es decir, el contenido de *frase$.*

Una característica curiosa de la asignación de variables es que podemos asignar a una variable el resultado de haber operado con ella misma. Por ejemplo, si escribimos:

```
a=40
```

Y después:

```
a=a+1
```

BASIC toma el contenido de a, le suma 1 y lo almacena en a. Resultado: añadimos 1 a a. Esto se puede hacer también con variables alfanuméricas, por ejemplo:

```
a$=upper$(a$)
```

LAS LIMITACIONES DE LAS VARIABLES

Existen limitaciones, claro. Respecto a las numéricas, podemos suponer que su contenido se limita al rango de números manejable por BASIC. Este depende de la estructura interna del ordenador, pero generalmente está entre 1e38 y le−38.

```
print 6↑6↑6↑6↑6 (seis elevado a seis elevado a...)
```

Provoca el error *overflow* o «rebosamiento».

Se pueden definir unas características concretas a las variables, pero tales procedimientos sobrepasan el contenido de este libro; los analizaremos en el siguiente número dedicado a BASIC.

Las variables alfanuméricas tienen otro pequeño defecto; volviendo a los loros, éstos no pueden memorizar letras y más letras. Sólo pueden con 255 (aceptable de todas formas para un simple loro). El caso es que, en muchas ocasiones, 255 caracteres se quedan cortos para programar ciertas aplicaciones.

BASIC indica que se ha sobrepasado el límite máximo con el error *string too long* (cadena demasiado larga). Es muy fácil provocarlo:

```
q$=string$(200,"e")+string$(200,"p")
```

Genera una cadena de 400 caracteres y, por tanto, el error.

Antes hablábamos de un «gran golpe». La instrucción que provoca la amnesia de los loros es CLEAR. Comprobar sus efectos es muy fácil:

```
a=23
b=24
print a+b
clear
print a+b
```

El segundo PRINT devuelve 0, y eso es porque las variables *a* y *b* están vacías, debido a la acción de CLEAR. No olvidemos que esta orden borra TODAS las variables, y su efecto es irreversible.

MÁS SOBRE PRINT

El comando PRINT permite combinar varios elementos a la vez para imprimir, tanto numéricos como alfanuméricos. Pero entre uno y otro se debe incluir un separador. Normalmente, éste separador es el signo ";" (punto y coma). Esto significa para PRINT que el elemento que viene a continuación debe escribirse en la misma línea y, además, pegado a lo anterior. Por ejemplo:

```
print "cosas";"pegadas"
```

Da como resultado *cosaspegadas.*

Esto no es una suma de cadenas, puesto que no se ha efectuado ninguna operación. Es, simplemente, el resultado impreso de unos elementos dispuestos tal y como se ha ordenado.

Si queremos separar las palabras, tendremos que incluir un espacio en una de las cadenas:

```
print "cosas ";"separadas" o bien
print "cosas";" separadas"
```

Con los números, PRINT será algo menos laborioso: si incluimos varias cifras, separadas por punto y coma, BASIC se encargará de separarlas lo suficiente como para evitar confusiones. Así:

```
print 32;45
```

Da *32 45,* y no *3245,* lo cual sería seguir el comportamiento propio de una cadena.

Existe otro separador para PRINT, y su efecto es similar al tabulador de una máquina de escribir. PRINT considera que la pantalla puede dividirse en zonas de ocho columnas y cuando se le indica pasa a la siguiente zona. El signo de tabulación es la coma (,).

```
print "cosas","mas separadas"
print 123,3445
print valor,valor2,nota$
```

Atención al segundo ejemplo: se trata de dos números; ya dijimos que el decimal es en BASIC un punto y no la coma. Lo que hacemos aquí es separar 123 de 3445.

Lo mejor para conocer a fondo el funcionamiento de PRINT es

probar y observar el resultado en la pantalla. Por ejemplo, si escribimos:

```
print "comamos";;;;;;;;;;;;;;;;;"cereales"
```

Obtendremos *comamoscereales,* a pesar de todos los signos introducidos. Para BASIC eso significa: "escribe a continuación, escribe a continuación, escribe..." por mucho que lo repitamos significa lo mismo.

Por el contrario, la expresión:

```
print "comamos",,,"cereales"
```

Separará ambas palabras a una considerable distancia (incluso puede que "cereales" baje a la siguiente línea). En esta ocasión BASIC entiende "pasa a la siguiente columna, pasa a la siguiente, a la siguiente" y no significa lo mismo que si lo decimos una sola vez. Es más, cuando llegue a la última columna de la línea se verá obligado a pasar a la siguiente línea y comenzar allí por la primera columna.

Al teclear habremos comprobado que cuando ya no queda pantalla para escribir, el ordenador desplaza la imagen hacia arriba para dejar sitio. Lo que se escape por arriba se pierde. En realidad, todo lo que hay en la pantalla está «perdido»; es decir, no resulta útil al ordenador puesto que no memoriza la pantalla; tan sólo lo que hayamos metido en las variables.

Este desplazamiento de la pantalla hacia arriba se denomina *scroll* (en inglés: rollo, persiana...). Tendremos que hablar del *scroll* más adelante, con los *listados.*

EJERCICIOS

17. Uno de estos tres nombres no es correcto para una variable, ¿cuál?

A) Ingresos mensuales.
B) Contador/1.
C) Mayday.mayday.estoy.cayendo.321.corto.

18. Uno de estos tres nombres es correcto para una variable, ¿cuál?

A) Estamo$comiendo.nomole$te$.
B) mensaje.recibido$.
C) $.

19. Una de estas tres asignaciones no es correcta, ¿cuál?

A) Antonio$=chr$(asc("G"))+"enial!!!".
B) clave=string$(20,"a").
C) helado.guay$="Trufa+chocolate+vainilla".

20. Una de estas tres asignaciones es correcta, ¿cuál?

A) ingresos=beneficio$.
B) incorrecto=len("incorrecto").
C) imposible$=incompatible.

21. ¿Se limita de alguna forma la longitud de una variable alfanumérica?

A) Su nombre sí, pero su contenido no.
B) Su contenido sí, pero no su nombre.
C) Su nombre sí, pero su contenido también.

22. ¿Cuál es la longitud máxima de una variable alfanumérica?

A) Es variable.
B) 255 caracteres.
C) 255 cifras.

23. Para eliminar el contenido de la variable a$ escribiremos:

A) a$="".
B) clear.
C) clear a$.

24. Para borrar el contenido de todas las variables, el procedimiento correcto es:

A) Lo mejor es apagar el ordenador y volver a encenderlo.
B) Dar una gran palmada.
C) Teclear CLEAR.

25. ¿Pueden existir a la vez las variables *bien* y *bien$?*

A) No, porque BASIC las confundiría.
B) Es su problema.
C) Sí, porque son de distinto tipo.

26. Es correcta la expresión PRINT total;,;subtotal:

A) No, porque hay tres separadores.
B) Sí.
C) No, porque el subtotal se escribe siempre antes.

27. Si tecleamos *PRINT "a",,,,,,,,,,,,,,,,,,,,"a".*

A) La segunda "a" aparecerá en la pantalla varias columnas más abajo que la primera.
B) La segunda "a" aparecerá a continuación de la primera.
C) No aparecerá la segunda "a" porque hay más de un separador.

EL PROGRAMA

No hemos hecho nada más que empezar; lo visto hasta ahora es tan sólo una pequeña parte de lo que puede hacer el intérprete BASIC. Los ordenadores trabajan «solos»; es decir, no debemos estar tecleando continuamente lo que deben hacer a continuación (aunque eso es lo que hemos hecho hasta ahora en este libro).

Lo normal es que todas las instrucciones que han de ejecutarse se indiquen a la vez y después se señale al ordenador que empiece a seguirlas. Esta sucesión de instrucciones es lo que llamamos programa, y se introduce en la memoria del ordenador para que dicha sucesión pueda ejecutarse las veces que haga falta.

Hay distintas formas de almacenar el programa en la memoria del ordenador, dependiendo del lenguaje y de las características de la máquina. En el caso de BASIC las instrucciones se numeran; BASIC las memoriza por orden creciente de dichos números (que deben estar en el margen 1-65534).

Así, un programa BASIC tiene un aspecto parecido a:

```
1 instrucción
2 instrucción
3 instrucción
...
```

Es lo que llamamos listado de un programa.

Pero esto es pura teoría. Para empezar, podemos colocar varias instrucciones en una misma línea. Esto se consigue separándolas con el signo «:» (dos puntos). Ahora, un programa puede ser:

```
1 instrucción:instrucción
2 instrucción
3 instrucción:instrucción:instrucción
...
```

No debemos recargar las líneas del programa. Así evitaremos un aspecto confuso y una gran dificultad para corregir o depurar programas. Lo aconsejable es, en principio, una sola instrucción por línea.

Otro detalle es que, normalmente, las líneas no se numeran de uno en uno; esto se debe a la posibilidad de omisiones en la codificación del programa. Imaginemos que se nos han olvidado un par de instrucciones entre las líneas 4 y 5. Si quisiéramos entonces añadir más líneas tendríamos que reescribir todo a partir de la inserción en la 5.

Si, por ejemplo, numeramos de 10 en 10 podemos incluir 9 líneas adicionales; BASIC se encargará, como siempre, de clasificarlas por orden creciente. Está claro entonces que el intervalo de separación entre números es directamente proporcional a la complejidad del programa.

Vamos ya con la práctica. Tenemos una secuencia de instrucciones para el cálculo del área de un triángulo de base 10 u.l. y altura 8 u.l.:

```
altura=8
base=10
área=(base*altura)/2
?"El área es ";área
```

Para que el ordenador almacene todo esto en la memoria debemos ponerle un número a cada orden. Además, debe ser un número creciente. Vamos a teclear lo siguiente:

```
5 altura=8
7 base=10
```

```
9 área=(base*altura)/2
11 ?"el área es ";área
```

¿Qué ha pasado? El PRINT no funciona, y si comprobamos el contenido de *área, base* y *altura,* veremos que son 0, como si no se hubiese asignado valor alguno. Simplemente se ha almacenado el programa, pero no se ejecuta hasta que se lo digamos.

El Manejo De List

Se puede comprobar en cualquier momento que el programa está en la memoria. Basta con teclear LIST. Aparecerá entonces un listado completo.

El que aparezca un «listado completo» puede resultar molesto si éste ocupa más de la longitud de la pantalla (cosa que ocurre en el 99 % de los programas). En este caso comenzará el desplazamiento (que hemos llamado scroll) y, por tanto, se nos escaparán las primeras líneas sin que nos haya dado tiempo a verlas.

Para evitar esto nuestro Amstrad dispone de un sistema para detener temporalmente el listado. En los CPC hay que pulsar la tecla ESC una vez, con lo que parará el scroll, y cualquier OTRA tecla para seguir. Si volvemos a pulsar ESC se detendrá el listado.

En el PCW debemos pulsar ALT y S. Para continuar el listado cualquier pulsación. Si pulsamos STOP se detendrá.

Pero LIST es una instrucción mucho más completa. Si, por ejemplo, queremos comprobar el contenido de una sola línea de programa basta con indicarlo:

```
list 9
```

El resultado es:

```
9 área=(base*altura)/2
```

Si la línea que hemos escrito no existe BASIC no mostrará nada (el PCW dará el error *line does not exist*, esa línea no existe) —es un intérprete algo exigente..

Igualmente, podemos indicar a BASIC que nos muestre un fragmento del listado; es decir, un grupo de líneas. Esto se consigue con:

```
list 5-9
```

Que muestra todas las instrucciones comprendidas entre esos números (incluidas 5 y 9).

Otra opción de LIST muy interesante permite mostrar fragmentos del tipo "principio hasta aquí" o bien "desde aquí hasta el final". Esto se expresa con list −n y list n−, respectivamente, donde n es el número de línea. Por ejemplo:

```
list −7
```

Muestra desde el principio hasta la 7 incluida, y:

```
list 7−
```

Muestra desde la 7 hasta el final.

Hay algo que destaca especialmente en los listados: aquellas instrucciones que conoce BASIC están en mayúsculas, y el signo ? (ya dijimos que equivale a PRINT) figura en el listado, precisamente, como un PRINT. Esto ocurre porque BASIC interpreta lo escrito y lo codifica internamente para ahorrar memoria. Al listar se indica todo lo interpretado en mayúsculas, de manera que con un rápido vistazo podemos detectar un error. Por ejemplo, si escribimos list 11 obtendremos:

```
11 PRINT "El área es ";área
```

Si por error, lo que escribimos al introducir la línea fue "PRNIT" en lugar de PRINT, CON list 11 obtendríamos:

```
11 prnit "El área es ";área
```

lo cual significa que BASIC no ha conseguido codificar ese *prnit.*

EJECUTANDO UN PROGRAMA

El comando para comenzar la ejecución de un programa es RUN. Si una vez introducido el programa tal como indicábamos anteriormente escribimos run, en la pantalla aparecerá:

```
El área es 40
```

Esto significa, simplemente, que la secuencia de instrucciones que habíamos introducido ha sido ejecutada. Podemos comprobar el contenido de las variables, así:

```
print altura da 8 y
print base da 10.
```

Y ejecutar el programa tantas veces como queramos. Basta con escribir de nuevo run.

RUN admite un parámetro, aunque normalmente no tienen utilidad (repetimos: normalmente). Podemos ejecutar programa empezando por una línea determinada, no necesariamente la primera. Por ejemplo, run 7 empezará a ejecutar el programa a partir de la línea 7. Esto sólo suele ser útil durante la corrección o depuración de programas. En estas operaciones podemos comprobar el funcionamiento de un sector del programa con run n, donde *n* es la línea de comienzo de dicho sector.

Hay un detalle que debemos tener muy en cuenta. RUN es un comando, por decirlo de alguna manera, «múltiple». Esto significa que el intérprete BASIC no sólo ejecuta el programa al recibir esta instrucción, sino que, de paso, realiza otras pequeñas tareas. La que nos interesa en este momento es CLEAR. O sea, que cuando tecleamos run en realidad es como si hubiéramos escrito *borra las variables* y *luego ejecuta el programa.*

La razón de este "añadido" de CLEAR es bien sencilla: nuestro programa utilizará ciertas variables, pero pueden coincidir con algunas ya empleadas. En ese caso el programa podría contener errores, derivados de los valores pertenecientes a esas variables ajenas al programa, pero cuyos nombres coinciden. La solución es simple; y el comando run la contiene: borrar las variables.

BORRAR, CORREGIR... Y EMPEZAR DE NUEVO

Lo más seguro es que queramos introducir otro programa distinto para comprobar cómo funciona. Pero tenemos uno en memoria y hay que borrarlo. La instrucción para borrar programas es NEW, y podríamos decir que es también de tipo «múltiple», ya que borra el programa y además las variables; tras su actuación no queda nada en el ordenador de lo que hayamos hecho antes. No hace falta advertir sobre la precaución a la hora de utilizar esta instrucción. Los errores, en este caso, se pagan, puesto que el efecto de NEW es irreversible. Probémoslo con el programa recién introducido:

```
list
t altura=8
```

```
7 base=10
9 área=(base*altura)/2
11 PRINT "El área es ";área
Ready
new
Ready
list
Ready
```

Otra forma de borrar es introducir las mismas líneas, pero vacías. Por ejemplo, para borrar la línea 11:

```
list
5 altura=8
7 base=10
9 área=(base*altura)/2
11 PRINT "El área es ";área
Ready
11 (línea vacía)
Ready
list
5 altura=8
7 base=10
9 área=(base*altura)/2
Ready
```

La línea 11 ha desaparecido.

De similar manera podemos modificar una línea. Por ejemplo, si queremos que el anterior programa calcule el área de un rectángulo, sobra la división entre dos. Bastaría con reescribirla de nuevo:

```
list
5 altura=8
7 base=10
9 área=(base*altura)/2
11 PRINT "El área es ";área
Ready
9 área=base*altura
Ready
list 9
9 área=base*altura
Ready
```

De todas formas, hay un modo de corregir líneas mucho más cómodo, con la instrucción EDIT (editar). Si quisiéramos corregir esa misma línea 9 teclearíamos:

```
edit 9
```

Y la línea aparecería a la altura del cursor tal y como si la acabáramos de escribir. Ya sabemos que llevando el cursor a un lado y a otro, y con las teclas de borrar, podemos corregir cualquier cosa. Para terminar, se pulsa RETURN y la línea se almacena en memoria corregida. (Los CPC tienen además otro sistema de corrección con COPIA más cómodo, pero los detalles específicos de una y otra máquina escapan del objetivo de este libro).

Existe la instrucción DELETE para borrar muchos números de línea a la vez. Resulta muy útil cuando tenemos que borrar grandes fragmentos de programa. Por ejemplo, supongamos que nos sobran las líneas 30 a 100; sería muy molesto escribir todos esos números para que se borraran.

Con DELETE las borraríamos así:

```
DELETE 30-100
```

El modo de mencionar las líneas es el mismo que para LIST, de manera que DELETE −40 borrará hasta la línea 40, inclusive; DELETE 200− borrará a partir de la 200... Es muy sencillo borrar líneas de más por error, y el efecto de DELETE es irreversible (¡cuidado!).

EJERCICIOS

28. ¿Qué es un programa?

A) Un conjunto ordenado de instrucciones almacenado en la memoria del ordenador.
B) Un listado.
C) Emisión de TV.

29. En BASIC el separador de instrucciones es:

A) Punto y coma.
B) Dos puntos.
C) Ladrillo de hueco doble y mortero de cemento EP-703.

30. Muchos programadores tienen la costumbre de numerar las líneas de 10 en 10. ¿A qué se debe esto?

A) Se hace sólo para redondear.
B) Para poder intercalar otras líneas si fuera necesario.
C) Para facilitar a BASIC la ordenación de números de línea.

31. Si tras introducir en la memoria una línea 10 y otra 20 tratamos de introducir una 15:

A) BASIC la rechazará.
B) Será incluida en el programa, después de la 20.
C) Será incluida entre la 10 y la 20, puesto que se respeta el orden de menor a mayor.

32. Para detener un listado muy largo en el CPC el procedimiento correcto es:

A) Lo mejor es apagar y volver a encender el ordenador.
B) Se pulsa CTRL y C a la vez.
C) Se pulsa ESC.

33. El mismo caso en un PCW será resuelto...

A) Pulsando CTRL y S a la vez.
B) Pulsando MAYS, EXTRA y SAL a la vez.
C) Bajando el brillo del monitor.

34. Si escribimos el programa en minúsculas... ¿Por qué al listarlo BASIC convierte las instrucciones a mayúsculas?

A) Porque indica así que han sido convenientemente codificadas.
B) Porque las instrucciones son así de importantes.
C) Cosas de BASIC...

35. Además de ejecutar el programa almacenado en memoria (si lo hay), ¿qué otra cosa hace RUN?

A) Borrar la pantalla.
B) Podemos estar contentos de que sólo le dé por ejecutar al programa, y no...
C) Borrar el contenido de todas las variables.

36. ¿Qué medios tenemos para corregir una línea de programa?

A) Teclear de nuevo el programa.
B) Reescribirla o corregirla con EDIT.
C) Lo mejor es apagar y volver a encender el ordenador.

37. Si queremos borrar la línea 120, la forma más rápida y cómoda de hacerlo es:

A) Con DELETE 120−.
B) Teclear 120.
C) Con DELETE −120.

38. ¿Cómo se borra un conjunto muy extenso de líneas rápidamente?

A) Tecleando todos y cada uno de los números de dichas líneas.
B) Con DELETE, indicando primera y última línea del conjunto.
C) Lo mejor es emplear explosivo plástico de media potencia.

MÁS SOBRE PROGRAMACIÓN. AYUDAS

Interpretar la codificación de un programa no es siempre sencillo. El ejemplo que hemos utilizado hasta ahora no es un verdadero programa, ya veremos porqué. Los programas extensos (la mayoría lo son) necesitan un esquema previo llamado organigrama. Este esquema se representa tradicionalmente con unos determinados símbolos que varían según la operación que se realiza.

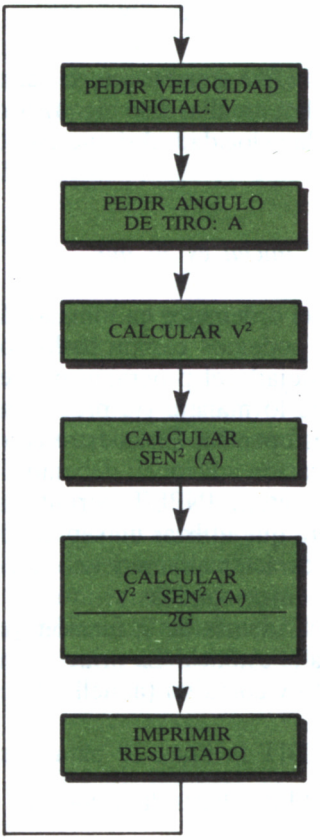
En realidad, el conocer los símbolos no importa demasiado. El objetivo del organigrama es tener un «esqueleto» del programa en el cual se reflejen las operaciones más importantes. El esqueleto, por ejemplo, para hacer una tortilla es el que representamos en la figura.

Algo más al alcance de un ordenador es calcular, por ejemplo, la altura máxima alcanzada por un proyectil lanzado a una cierta velocidad e inclinación. Como vemos en el organigrama, casi todos los pasos para realizar ya los conocemos.

8.5.1. Organigrama del algoritmo de preparación de una tortilla.

EL MANEJO DE INPUT

Podemos empezar a escribir el programa. Lo primero es pedir el valor de la velocidad inicial en m/s. ¿Pedir? Todavía no sabemos cómo se hace eso. De todos modos está claro que un programa no lo es verdaderamente si no resuelve cualquier problema con distintos datos. Hemos visto un programa que calcula el área de un triángulo, pero sólo de UNO. Esto no es nada útil, puesto que para calcularlo con otras dimensiones hay que reescribir el programa.



8.5.2. Organigrama del algoritmo de cálculo del alcance vertical de un proyectil.

Sin embargo, si conseguimos que el programa PIDA la variable durante su ejecución podemos escribir todas las dimensiones que queramos. Eso sí es un programa, puesto que resuelve todos los casos posibles dentro de su capacidad (es decir, todos los triángulos posibles).

Pues bien, la instrucción para pedir el contenido de una variable es INPUT (comentario)(separador) variable. Esto parece muy complicado, pero no lo es. Escribamos este «mini programa»:

```
10 print "CALCULO DEL ALCANCE VERTICAL DE UN
PROYECTIL
20 input v
30 print "La velocidad inicial es ";v;" m/s"
```

Al ejecutarlo con RUN veremos que no ocurre nada; simplemente el cursor aparece en la pantalla. Eso significa que el programa pide algo, sabemos que es la velocidad. Así que escribamos una, por ejemplo 30:

```
30
La velocidad inicial es 30 m/s
```

Esto significa que el ordenador ha tomado el 30 y lo ha almacenado en la variable v (podemos comprobarlo con print v).

Tal como hemos dejado el programa puede resultar un misterio para otra persona que lo maneje (y, precisamente, el programador trabaja para que otros operen). En nuestro caso el cursor aparecerá, pero sin ninguna indicación adicional: debemos introducir un dato numérico, alfanumérico, etc... INPUT permite añadir un comentario aclaratorio, pero habrá que utilizar un separador para distinguirlo de la variable. Dado el carácter alfanumérico del comentario (sea cual sea) ha de indicarse entre comillas.

El separador tiene, además de la función que le da nombre, otra dedicada totalmente a la estética. En principio, utilizaremos como separador el signo punto y coma (;) (aquello de... "escribe a continuación").

Corrijamos con EDIT la línea 20 para que tenga este aspecto:

```
20 input "Introduzca la velocidad inicial";v
```

Y ejecutemos el programa. Nada más empezar INPUT se encontrará con un comentario, o sea, que lo imprimirá. Después, pide el valor de una variable v. El resultado interno es el mismo, pero resulta

mucho más comprensible para una persona ajena al proceso de programación, es decir, la que lo utiliza.

¿Qué ocurre si introducimos un dato incongruente?; por ejemplo, una cadena. INPUT se encontrará con que la variable para almacenar dicha cadena es numérica y, por tanto, repetirá el proceso emitiendo el mensaje *?Redo from start* (algo así como "empezamos otra vez").

Seguimos con el programa: ahora tenemos que pedir el ángulo de tiro del proyectil respecto a la horizontal, o sea, que ya sabemos las líneas que debemos añadir:

```
40 input "Introduzca ángulo de tiro respecto al horizontal";a
50 ángulo=a/180*pi
```

Como vamos a utilizar la función SEN hay que pasar el ángulo a radianes. Ya explicamos el proceso que utilizamos ahora en la línea 50. Dijimos también cómo obtener PI en el PCW:

```
50 ángulo=a/180*(4*atn(1))
```

Sigamos añadiendo:

```
60 print "El ángulo de tiro en radianes es ";ángulo
70 v cuadrado=v ↑ 2
80 sen.cuadrado=sen(ángulo) ↑ 2
90 g=9.8
100 resultado=(v.cuadrado*sen.cuadrado)/ (2*g)
110 print "La altura máxima alcanzada es ";resultado
```

Ya tenemos terminado el programa. Podemos decir además que es un "auténtico" programa, ya que resuelve todos los problemas de este tipo que le sean planteados (varias velocidades, varios ángulos).

Podemos, eso sí, mejorarlo. Por ejemplo, utilizamos demasiadas variables y, por tanto, demasiada memoria. Es posible, como dijimos en el capítulo 3, asignar a una variable el resultado de operar con ella misma, así que podemos cambiar la última parte del programa por:

```
50 a=a/180*pi
60 print "El ángulo de tiro en radianes es ";a
70 v= v ↑ 2
80 a=sen(a) ↑ 2
90 g=9.8
100 resultado=(v*a)/(2*g)
110 print "La altura máxima alcanzada es "; resultado
```

El ahorro de memoria no es siempre aconsejable. Si nos damos cuenta, el listado ofrece ahora un aspecto mucho más confuso. El grado de simplificación en el uso de variables depende en gran parte del gusto del programador. En principio conviene tomar variables con distintos nombres. Con el tiempo veremos lo que nos conviene reducir a una misma variable con soltura. Es, como todo, cuestión de práctica.

UNA GRAN AYUDA: AUTO

Es una costumbre (y bastante extendida) tomar como intervalo entre líneas de programa 10. También se utiliza el 100, aunque es un poco exagerado. Pero sea cual sea el modo en que vayamos a escribir los números hay algo que no varía: tenemos que teclearlos.

Para evitar este «trabajito» nuestro BASIC Amstrad dispone de la orden AUTO. Su función es simplemente escribir los números de línea para que nosotros no tengamos que preocuparnos de ese detalle. Vamos a comprobarlo:

```
new (para borrar el otro programa)
Ok
auto
10-
```

Aparece en la pantalla un 10 y a continuación el cursor. Escribamos algo, por ejemplo:

```
10 print "Esto es otro programa"
```

Al pulsar RETURN aparece un 20 y de nuevo el cursor a su derecha. Este proceso es continuo y para interrumpirlo debemos utilizar la tecla STOP del PCW o ESC del CPC. El ordenador emite el mensaje *Break* (detenido) y todo sigue como antes, excepto, eso sí, unas líneas en memoria cuyos números no hemos tenido que teclear.

Al hacer esta prueba AUTO nos ha proporcionado líneas de 10 en 10 empezando por la 10. Pero esto de la numeración es algo bastante subjetivo y personal, de manera que si queremos nuestro propio incremento y comienzo se lo podemos decir a AUTO así:

```
auto comienzo, incremento
```

Por ejemplo, si queremos líneas de 5 en 5 empezando por la 20 escribiremos:

```
auto 20,5
```

Si simplemente queremos que vayan de 10 en 10 pero empiecen en la 100:

```
auto 100
```

Y si queremos que el intervalo sea de 2, pero empezando en 10:

```
auto ,2
```

Otra característica de AUTO es que si encuentra una línea ya escrita no la borra, sino que permite su modificación. Si tenemos el programa:

```
10 print "2 más 2 son ";2+2
20 print "y 1 más 1 son ";"1"+"1"
```

Y escribimos AUTO, aparecerá la línea 10 tal y como si hubiéramos escrito EDIT 10, con la diferencia de que al pulsar RETURN no terminará el proceso, sino que aparecerá la 20. Después de ésta aparecerán la 30, 40... pero en blanco.

COMENTARIO...

Cuando hablábamos de INPUT decíamos que las cosas debían quedar bien claras de cara a la persona que utiliza el programa. Pero también tiene que quedar claro el aspecto de un programa para el propio programador. Es por eso que BASIC permite incluir comentarios (en inglés *REMarks)* en los números de línea. Esto lo conseguiremos con REM, y BASIC ignorará lo que haya escrito detrás de él, puesto que se trata de un comentario de interés exclusivamente para el programador. Si, por ejemplo, escribimos:

```
10 rem Este trabajo está dedicado a Paz
```

Y a continuación lo ejecutamos, no ocurrirá absolutamente nada. Aún más, si escribimos:

```
10 rem print 40
```

Y lo ejecutamos, tampoco ocurrirá nada, y es que BASIC ignora todo (repetimos: TODO) lo que haya después de un REM.

Cómo Cambiar El Sentido Del Programa: La Bifurcación

Tras ejecutar varias veces el programa nos daremos cuenta de lo molesto que es estar escribiendo RUN todo el tiempo. ¿Qué tal si el programa se repitiera continuamente? Para ello necesitamos que el curso normal del programa cambie a dirigirse a otra línea que no sea la siguiente. El comando para ello es GOTO n (go to = ir hacia), donde n es el número de línea hacia el cual ha de dirigirse el intérprete para seguir ejecutando. Vamos a añadir la línea 120:

```
120 goto 20
run
```

Veremos que, una vez calculado el alcance de un proyectil, nos pedirá los datos de otro. Esto ocurre porque BASIC encuentra la instrucción GOTO y deja todo para dirigirse a la línea especificada. Ya tenemos un programa continuo que calcula las veces que queramos nuestro problema. Mejor dicho... no «las veces que queramos», sino infinitas veces, porque este programa no se acaba nunca: al llegar al final, GOTO le hace empezar de nuevo.

El ordenador no se cansará, pero nosotros sí. La forma de detener el ordenador es pulsar ESC dos veces en el CPC, y STOP una vez en el PCW. El ordenador emitirá el mensaje *Break in n,* donde n indica el número de línea en el que se le «pilló». Con lo que sabemos hasta ahora, la utilización de GOTO resulta bastante pobre, y debemos acabar así.

Y aprovechamos para saber cómo reanudar la ejecución del programa: el comando CONT (de *CONTinue,* continuar) hace que BASIC siga por donde se había quedado. CONT es algo exigente: no permite que se modifique el programa. Es decir, si interrumpimos la ejecución y corregimos una línea, al escribir CONT recibiremos el mensaje: *Can't CONTinue* (no se puede continuar). Esto ocurre porque al modificar algo a BASIC se le rompen los esquemas; es necesario volver a empezar con RUN.

Podemos ver el contenido de las variables interrumpiendo el programa para ello. Esto no interfiere para nada a CONT. Además,

comprobar el contenido de las variables es muy útil cuando se está corrigiendo un programa cuyo error no conocemos demasiado bien.

Como veremos a continuación, GOTO no es por sí sólo una gran herramienta de programación. Existen otras instrucciones llamadas «de control», cuya función es controlar el curso de la ejecución del programa y mucho más útiles.

LISTADOS

LISTADO 1

```
10 input "Cadena 1 ";a$
20 input "Cadena 2 ";b$
30 input "Cadena 3 ";c$
40 print "La cadena 1 es """;a$;""""
50 print "La cadena 2 es """;b$;""""
60 print "La cadena 3 es """;c$;""""
70 print
80 print a$+b$+c$
90 print a$+c$+b$
100 print b$+a$+c$
110 print b$+c$+a$
120 print c$+a$+b$
130 print c$+b$+a$
```

LISTADO 2

```
10 rem Programa para calcular numero de patas
20 rem en una granja
30 input "Numero de patos ";patos
40 input "Numero de gallinas ";gallinas
50 input "Numero de cerdos ";cerdos
60 input "Numero de conejos ";conejos
70 input "Numero de vacas ";vacas
80 print "Patos..... ";patos
90 print "Gallinas..... ";gallinas
100 print "Cerdos..... ";cerdos
110 print "Conejos..... ";conejos
120 print "Vacas..... ";vacas
130 rem comenzamos el calculo de patas
140 ppatos=patos*2
150 pgallinas=gallinas*2
160 pcerdos=cerdos*4
170 pconejos=conejos*4
180 pvacas=vacas*4
190 print "Numero total de patas: ";ppatos+pgallinas+pcerdos+
pconejos+pvacas
```

LISTADO 3

```
10 rem ecuacion de segundo grado
20 print "La ecuacion de segundo grado tiene la forma"
30 print "          a*x^2+b*x+c=0"
40 print "Escriba los datos:"
50 input "Valor para 'a' ";a
60 input "Valor para 'b' ";b
70 input "Valor para 'c' ";c
80 print
90 print "La ecuacion es"
100 print a;"x^2+";b;"x+";c
110 raiz=sqr(b*b-(4*a*c))
120 divisor=2*a
130 solucion1=(-b+raiz)/divisor
140 solucion2=(-b-raiz)/divisor
150 print "Primera raiz: ";solucion1
160 print "Segunda raiz: ";solucion2
```

EJERCICIOS

39. ¿Para qué nos servirá un organigrama?

A) Para estructurar nuestro programa.
B) Para planear cuáles deben ser los números de línea apropiados.
C) En tamaño folio, para el cucurucho de las castañas.

40. ¿Son los organigramas estrictamente necesarios?

A) No, pero sirven de gran ayuda, sobre todo en programas grandes.
B) La verdad es que sólo sirven para liar a la gente.
C) Sólo serán útiles en pequeños programas.

41. La función de INPUT es...

A) Cambiar el contenido de una variable por el teclado.
B) Bajar el brillo del monitor con el contenido de la variable.
C) Obtener un valor por el teclado para asignarlo a una variable.

42. Para separar el comentario de la variable, dentro del INPUT, se utiliza:

A) Punto y coma.
B) Dos puntos.
C) Un tupido velo.

43. Cuando los datos introducidos a través de un INPUT no son concordantes con la variable que debe contenerlos se obtiene el mensaje:

A) Type mismatch.
B) Redo from start.
C) This program will not run in this environment.

44. Para evitar escribir los números de línea que preceden a las instrucciones de un programa:

A) Se hace sin números.
B) Con AUTO.
C) Lo mejor es apagar y no volver a encender el ordenador.

45. AUTO nos permite omitir los números de línea y además:

A) Permite que el comienzo y el incremento sean aquellos que se indiquen.
B) Permite que el comienzo se indique, pero el incremento es automático.
C) No permite nada más.

46. La orden auto 100,5.

A) Genera líneas automáticamente, empezando por la 100 y de 5 en 5.
B) Genera líneas automáticamente, empezando por la 5 y de 100 en 100.
C) Generalmente, provoca el error Line does not exist.

47. Si AUTO encuentra un número de línea que ya contiene instrucciones:

A) Mostrará dichas instrucciones como si se tratara de un EDIT.
B) Borrará las líneas para permitir la introducción de las nuevas.
C) Emite el error Line already exists.

48. La única manera de detener el efecto de AUTO es:

A) Teclear STOP.
B) Pulsar la tecla STOP (PCW) o ESC (CPC).
C) Lo mejor es apagar y volver a encender el ordenador.

49. REM nos permite:

A) Almacenar los datos.
B) Imprimir comentarios.
C) Incluir comentarios en los listados.

50. Todo lo escrito detrás de REM se ignora, pero si escribimos tras él el separador de instrucciones (dos puntos):

A) No habrá ningún efecto especial por ello.
B) Podremos escribir otra instrucción.
C) Podremos escribir un comentario totalmente separado del anterior.

51. El efecto de GOTO en un programa es:

A) Cambiar el curso de la ejecución hacia otra línea, especificada detrás de GOTO
B) Generalmente, provoca el error Line does not exist.
C) Cambia el curso de la ejecución a la línea anterior.

52. Si GOTO hace referencia a una línea inexistente:

A) BASIC emite el error Line does not exist.
B) Se sigue por la línea superior más próxima.
C) Se sigue por la línea inferior más próxima.

LOS BUCLES

btengamos ahora un programa en BASIC cuya función es saludar a cinco personas después de haber preguntado su nombre. Si lo escribimos con lo que hasta ahora sabemos el listado sería más o menos así:

```
10 rem Programa que saluda 5 veces
20 input "Cómo te llamas";a$
30 print "Hola, ";a$
40 input "Cómo te llamas";a$
50 print "Hola, ";a$
60 input "Cómo te llamas";a$
70...
...
```

Como vemos, esto resulta pesado (imaginemos que se debe saludar a cien personas). Una solución podía ser:

```
10 input "Cómo te llamas";a$
20 print "Hola, ";a$
30 goto 10
```

Pero ya vimos que este tipo de programas no se "acaba" nunca.

LOS BUCLES FOR-NEXT

BASIC nos ofrece para hacer más fácil el trabajo del programador el control del tipo FOR - TO - NEXT. Para comprender su funcionamiento vamos a recurrir a un pequeño ejemplo.

Tenemos a un señor dedicado a llenar depósitos de agua. Su proceso normal (lo que repite) es llenar el cubo y echarlo en el depósito. Se detiene cuando ve que el depósito está lleno, aunque es un poco estúpido: sólo se da cuenta de que está lleno cuando ya tiene otro cubo listo para echar.

Cuanto más grande sea el depósito más veces debe repetir su proceso. Y cuanto más grande sea el cubo que le demos para transportar el agua menos veces repetirá ese proceso.

En principio, vamos a darle un cubo de un litro y un depósito de 5 litros. El proceso de carga lo repetirá cinco veces y habrá llenado un cubo que el final no ha echado, porque vio el depósito lleno. El total de cubos llenos es, por tanto, de 6.

Si trasladamos esto a BASIC lo escribiremos: FOR var=1 TO ld, donde *var* es una variable numérica donde se almacena el número de cubos que han sido llenados hasta el momento y *ld* es el número de litros del depósito. La forma de que el señor vaya a por otro cubo es escribir NEXT var.

Si escribimos este programa:

```
10 for a=1 to 5
20 print a
30 next a
```

La línea 20 se encarga de mostrar el número de cubos que ha llenado el señor. El conjunto de líneas 10-30 es lo que llamamos un *bucle.*

Aplicado al problema que antes nos planteábamos, el listado sería:

```
10 print "Programa que saluda"
```

```
20 for a=1 to 5
30 input "Cómo te llamas ";a$
40 print "Hola, ";a$
50 next a
60 print "Ya no saludo más."
```

Cuando BASIC (que, por supuesto, asume el papel de acarreador de cubos de nuestro ejemplo) encuentra la línea 20, coge un cubo y se dispone a llenar un depósito de 5 unidades. Por supuesto, el cubo tiene aquí forma de variable y su contenido indica las veces que ha sido llenado.

BASIC ejecuta lo que encuentre (líneas 30 y 40) como lo haría en cualquier otro caso. Asumiremos que dichas líneas son el proceso de llenado del depósito. Pero cuando llega a la 50 se encuentra con que debe llenar otro cubo. Para ello debe volver de nuevo a la línea 20. Ahora, la variable *a* ya vale 2.

Cuando BASIC llegue a la línea 20 por quinta vez (*a* será cinco) y tras llenar el depósito (saludar) llegue al NEXT a llenará el cubo, pero al subir arriba (ya dijimos que era algo estúpido) verá que ya está lleno, con lo cual olvida el NEXT y pasa a la línea 60. *a* vale 6 porque se ha llenado el cubo una vez inútilmente.

Y hablando del cubo, hemos dejado un pequeño detalle atrás, su tamaño. Efectivamente, FOR-TO-NEXT admite un auxiliar que determina el incremento de la variable de control (la cual acompaña al FOR y al NEXT). Se trata de STEP. Por ejemplo, el programa:

```
10 for puaf=1 to 9 step 2
print puaf
next puaf
```

Hace que el cubo tenga dos litros de capacidad, luego el incremento de la variable irá de dos en dos (litros). El resultado de este programa es la impresión de los 5 primeros números impares. *puaf* valdrá al salir del bucle 11.

Igualmente, si hacemos un STEP más pequeño que uno, el incremento de la variable será precisamente ése:

```
10 for a=1 to 2 step 0.25
20 print a
30 next a
```

Este programa muestra los números 1, 1.25, 1.5, 1.75 y 2. Cuando BASIC salga del bucle *a* valdrá 2.25.

Siempre hemos empleado el 1 como valor de partida para la variable de control. Esto se debe sólo a que fuera coherente con el ejemplo, pero en realidad podemos tomar cualquier valor inicial a condición de que sea mayor que el final, por ejemplo:

```
10 for yeah=15 to 25
20 print "Estamos en el ""yeah"" número ";yeah
30 next yeah
```

Otro caso extremo que nos hemos planteado son los valores negativos (puesto que no hay capacidades negativas para nuestro depósito de ejemplo):

```
10 for q=-20 to -10 step 5
20 print q
30 next q
```

Y también hemos omitido en principio los cubos con capacidad negativa, es decir, que se van «llenando al vaciarlos»:

```
10 for a=10 to 0 step -1
20 print "Esto baja a ";a
30 next a
```

En este ejemplo *a* toma el valor 10 y cada vez que se encuentra con el NEXT le suma −1 (o bien le resta 1). El caso es que el último PRINT da 0 y al terminar *a* vale −1 (porque, eso sí, BASIC sigue siendo algo estúpido).

Si omitimos el STEP −1 en el ejemplo anterior BASIC comprobará que no es posible ir añadiendo 1 a 10 para llegar a 0, con lo cual salta ese bucle y no lo ejecuta.

Igualmente, si intentamos hacer un bucle creciente con un STEP negativo BASIC comprobará que no es posible llegar de la cifra pequeña a la mayor añadiendo un número negativo y, por tanto, no lo ejecutará.

Como en todos los casos, ya lo hemos dicho, lo mejor es experimentar. El funcionamiento de estos bucles quedará totalmente claro cuando se hayan planteado varios problemas «repetitivos».

El siguiente programa calcula el factorial de un número. El factorial es el producto de los números enteros comprendidos entre 1 y él incluido; por ejemplo, factorial de 7 (escrito 7!) es 1*2*3*4*5*6*7.

```
10 input "Dame el número";n
```

```
20 total=1
30 for a=1 to n
40 total=total*a
50 next a
60 ?n;"! es ";total
```

Este programa cumple el convenio *0!=1.*

WHILE - WEND

Podemos imaginar que no termina aquí la capacidad de BASIC para repetir secuencias. Es fácil conseguir que un bucle se repita hasta que se cumpla una determinada condición. Siguiendo el ejemplo de los saludos, imaginemos un programa que debe saludar hasta encontrarse con Fabián, a quien no saludará.

La condición entonces para que se salude es que el nombre sea distinto (<>) de "Fabián". El bucle condicional se construye con WHILE condición y para repetir el proceso, WEND, el cual asume un papel similar a NEXT.

Nuestro programa quedaría así:

```
10 input "Dime tu nombre";a$
20 while a$<>"Fabián"
30 print "Hola, ";a$
40 input "Dime tu nombre";a$
50 wend
60 print "A ti no te saludo, Fabián."
```

Al encontrar el WHILE (en inglés, mientras) BASIC condiciona la ejecución de las líneas anteriores al WEND (del inglés *While END,* fin de mientras) al contenido de a$, en este caso debe ser distinto de "Fabián".

La línea 10 «funciona» sólo una vez, pero es necesaria: en caso contrario el programa comenzaría diciendo: *Hola,*. Si previamente le hemos dado un nombre entrará en el bucle saludando a la primera persona. En el caso extremo de que esa primera persona fuera Fabián el bucle simplemente no se ejecutaría, puesto que la condición para ello falla: produce, por tanto, el mismo resultado que si Fabián no es el primero.

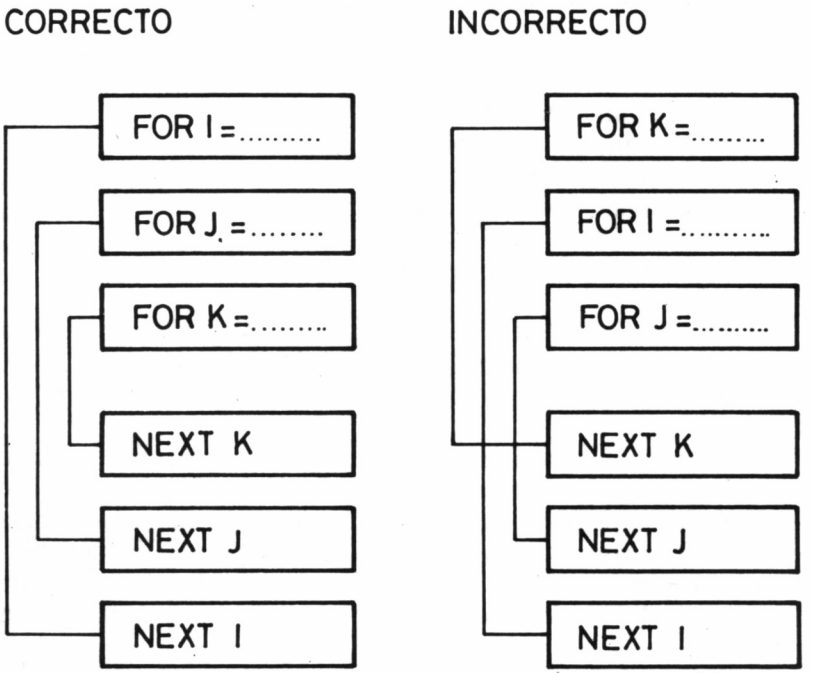
También podemos conseguir bucles «eternos» con WHILE. Si ponemos una condición que se cumple siempre el bucle será infinito:

```
10 rem programa kontestatario
20 while 2=2
30 print "Deten el programa o pudrete"
40 wend
```

Evidentemente, 2 jamás dejará de ser 2 (al menos así ha ocurrido hasta ahora), luego este programa no acaba nunca.

ANIDAR BUCLES: ¡CUIDADO!

Anidar consiste en meter un bucle dentro de otro bucle. Esto se debe hacer con precaución, ya que no deben estar «enganchados» sino incluido uno en otro.

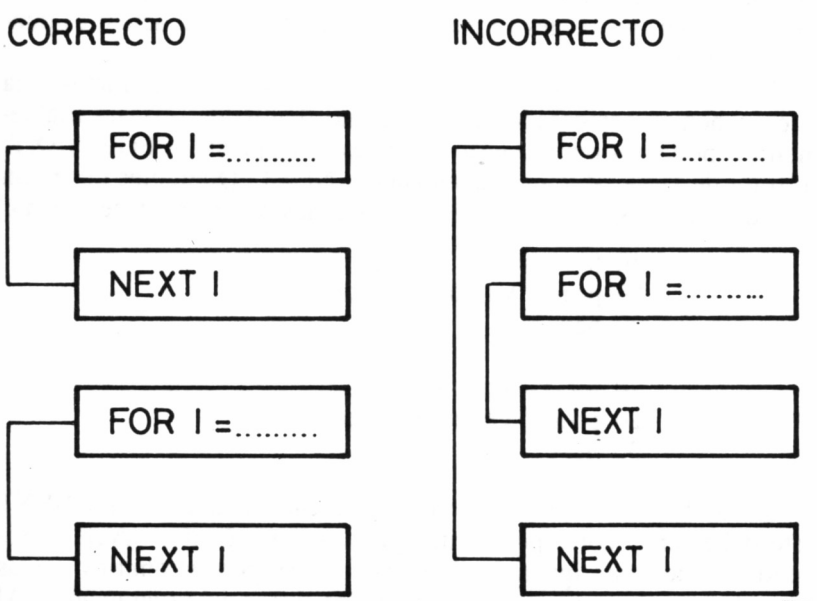


8.6.1. Esquemas de anidamiento de bucles.

Como ejemplo, vamos a teclear un programa que confecciona las tablas de multiplicar del 1 al 10.

```
10 for a=1 to 10
20 for b=1 to 10
30 print a;" por ";b;" es ";a*b
40 next b
50 next a
```

Vamos a seguir el camino que traza BASIC al ejecutar el programa: tomo a=1 y paso al siguiente bucle, donde tomo b=1. Hago la multiplicación, veo el NEXT de b y tomo b=2. Así hasta que b es 10, multiplico, llego a NEXT b, b es 11, o sea, que ignoro el bucle 20-40 y sigo; veo el NEXT de a, o sea, que tomo a=2 y vuelvo arriba: b=1, etc...



8.6.2. Esquemas de anidamiento de bucles.

En resumen, cuando a valga 1, b valdrá de uno a 10. Cuando a valga 2, b valdrá de uno a 10. Hasta a=10. Cuando se termine el programa ambos valdrán 11.

También podemos anidar un bucle FOR NEXT en un WHILE WEND:

```
10 input "Escribe algo ";a$
20 while a$<>""
30 for cuenta=1 to len(a$)
40 print mid$(a$,cuenta,1)
50 next cuenta
60 input "Escribe algo";a$
70 wend
```

Pongámonos como antes en el puesto de BASIC: Pido a$. Ahora ejecutaré las líneas 20-70 si a$ no es la cadena vacía. Tomo cuenta=1, y escribo la primera letra de a$. Veo el NEXT, o sea, que cuenta es 2, y vúelvo a escribir la segunda letra de a$. Así hasta que termine con la cadena. Luego termino el bucle 30-50 y pido otra vez a$; si es la cadena vacía puedo parar. Si no, vuelvo a empezar en 30.

El resultado del programa es desmenuzar aquello que tecleemos escribiéndolo carácter por carácter. Lo que se respeta en cualquier caso al anidar dos bucles (o más) es que el primero en actuar es también el último.

Dado que el proceso de anidación de bucles tiene la estructura fija que ya hemos comentado (es decir, uno dentro de otro invariablemente), podemos omitir cierto «detalle» al emplear los FOR-NEXT: se trata de la variable que acompaña al NEXT. Queremos decir con esto que los dos programas siguientes funcionarán igual de «bien»:

```
10 for a=0 to 10
20 print "Vamos por ";a
30 next a

10 for a=0 to 10
20 print "Vamos por ";a
30 next
```

En ambos casos BASIC «sabe» que cuando llegue a un NEXT éste debe ser el correspondiente a *a*. Los dos métodos sirven (el segundo se ejecutará quizá con mayor rapidez), pero el omitir o no las variables es, cómo no, dependiente de los gustos del programador. Al principio, conviene poner todos los nombres con los NEXT.

LISTADOS

LISTADO 1

```
10 rem programa que devuelve el codigo de los
20 rem caracteres introducidos previamente en
30 rem una cadena
40 input "Dame una cadena ('FIN' para terminar) ";cad$
50 while cad$<>"FIN"
60 for a=1 to len(cad$)
70 trozo$=mid$(cad$,a,1)
80 print trozo$,asc(trozo$)
90 next a
100 input "Dame otra cadena ('FIN' para terminar)" ;cad$
110 wend
120 print "Fin del programa"
```

LISTADO 2

```
10 rem Calculo de la media aritmetica
20 rem de n numeros introducidos
30 div=0
40 input "Dame un numero (0 para terminar) ";num
50 while num<>0
60 div=div+1
70 sum=sum+num
80 input "Dame otro numero (0 para terminar) ";num
90 wend
100 media=sum/div
110 print "La media de todos los numeros"
120 print "tecleados es";media
```

LISTADO 3

```
5 rem programa para demostrar
10 rem el comportamiento de
15 rem las sentencias FOR-TO
20 rem y STEP
25 input "Dame el comienzo del bucle ";com
30 input "Dame el final del bucle ";fin
35 print "Dame el salto entre incrementos"
40 print "Debe ser negativo si el comienzo es mayor"
45 input "que el bucle: ";salto
50 print "Empiezo en ";com
55 print "y saltando de ";sal;" en ";sal
60 print "debo llegar a ";fin
63 contador=0
65 for bucle=com to fin step sal
70 print bucle
75 contador=contador+1
80 next
85 print "El proceso se ha repetido ";contador;" veces."
90 goto 25
```

LISTADO 4

```
10 rem anidar bucles es sencillo si
20 rem se respetan los ordenes
30 for curso=1 to 3: rem cursos primero a tercero
40 for clase=1 to 6: rem seis clases en cada curso
50 for alumno=1 to 40: rem 40 alumnos en cada clase
60 print "Alumno numero ";alumno;" de la  clase  numero  ";clase
;" de el curso ";curso
70 next alumno,clase,curso
```

EJERCICIOS

53. ¿Qué estructura de bucle se ejecuta mientras se cumpla una condición?

A) La que ve que no tiene otro remedio.
B) WHILE-WEND.
C) FOR-NEXT.

54. ¿Cuántos parámetros se pueden expresar en una línea FOR?

A) 3.
B) 2.
C) Ninguno.

55. ¿Y cuántos de ellos son estrictamente necesarios?

A) Todos.
B) Ninguno.
C) Dos: el valor inicial y el final.

56. ¿Es posible emplear un valor negativo para el parámetro STEP?

A) STEP siempre es negativo.
B) Sí.
C) No.

57. ¿Y un valor decimal?

A) STEP es siempre decimal.
B) Sí.
C) No.

58. ¿Se ejecutará un bucle STEP 1 cuyo valor inicial sea superior al final?

A) Sí, pero de atrás a delante.
B) No.
C) Sí.

59. ¿Es posible la anidación combinada de bucles WHILE-WEND y FOR-NEXT?

A) Sí, siempre y cuando no se entrecrucen.
B) Depende de que se lleven bien entre ellos.
C) Sí, incluso WHILE-NEXT y FOR-WEND.

60. ¿Es correcta la siguiente estructura?

```
10 FOR I=1 TO 10
20 FOR J=3 TO 5
30 ...
40 NEXT I
50 NEXT J
```

A) No. Está incorrectamente anidada.
B) Sí. Está correctamente anidada.
C) El caso es que es un poco rara.

BASIC DECIDE

racias a los bucles WHILE-WEND nuestro ordenador puede decidir cuándo debe dejar de repetir un proceso (exactamente, cuando no se cumpla la condición expresada). Pero no termina aquí el «poder de decisión» que BASIC proporciona.

Para plantear una condición al ordenador tenemos el conjunto de instrucciones IF...THEN (en inglés, si... entonces). Entre IF y THEN debe estar la condición que queremos probar, y es siempre una expresión de tipo lógico; ya vimos lo que esto significaba en los primeros capítulos. Cuando la expresión vale −1 (verdad) entonces diremos que se cumple la condición y, por tanto, se ejecutará la instrucción pendiente de ella. Por ejemplo, escribamos:

```
a=5
Ready
if a>2 then print "A es mayor que 2"
A es mayor que 2
```

```
Ready
if a>10 then print "A es mayor que 10"
Ready
```

Con esto comprobamos que si la condición se cumple entonces se ejecuta lo que se dice a continuación. En caso contrario, se pasa a la siguiente instrucción. Todo queda más claro si traducimos lo que dice: *si a>2 entonces print "A es mayor que 2".*

Recordemos nuestro programa para saludar a todos menos a Fabián. Aquel programa terminaba cuando Fabián se presentaba al ordenador y éste no le saludaba. Podemos modificarlo para que no salude a Fabián, pero siga saludando después, es decir, que el programa no termine a pesar de eso:

```
10 input "Dime tu nombre";a$
20 if a$="Fabián" then goto 50
30 print "Hola, ";a$
40 goto 10
50 print "A ti no te saludo, Fabián"
60 goto 10
```

Introduciremos al ejecutarlo los nombres "Pirulo" y "Fabián".

Vamos a pensar como BASIC: primero pido el nombre. Me dan "Pirulo". Compruebo si es "Fabián", y como no lo es no sigo mirando detrás del IF y paso a la línea 30, donde saludo a Pirulo. Luego me obligan a ir a la línea 10, o sea, que pido otro nombre. Me han dado "Fabián". Eso es igual a "Fabián". Según lo que dice después de THEN, debo ir a la línea 50 y después me obligan a seguir en la 10...

Es fácil comprobar que este programa no se acaba nunca. Pero podemos utilizar también IF-THEN para determinar si un programa debe acabar o no:

```
10 input "Escribe algo ";a$
20 print a$
30 if a$<>"fin" then goto 10
```

Este programa sigue funcionando mientras a$ no contenga la cadena "fin". Si lo contiene, el programa termina.

Pero, por supuesto, BASIC admite que detrás del THEN se ponga cualquier instrucción; no sólo GOTO. Unos ejemplos «sueltos» pueden ser:

```
if 4=4 then w$="4 es igual a 4"
```

```
if beneficios<gastos then print
"Estamos perdiendo dinero"

if a$=upper$(a$) then if len(a$)=10
then b$="ADIOS"
```

El último ejemplo es un poquito complicado. Como ya sabemos, UPPER$ convierte a mayúsculas una cadena. Al principio BASIC lee: si a$ y a$ en mayúsculas son lo mismo (es decir, a$ está escrito en mayúsculas), entonces otra condición. Si no es así no ocurre nada. La otra condición es: si a$ mide 10 caracteres entonces hacer b$ igual a "ADIOS".

Hemos dicho que GOTO no siempre sigue a un THEN. Sin embargo, se trata de una construcción tan usual que BASIC admite «colarse» el GOTO. Por tanto, obtendremos el mismo resultado con:

```
20 if a$="Fabián" then goto 50
```

Y...

```
20 if a$="Fabián" then 50
```

Cuando hayamos probado con varios programas veremos que IF-THEN se queda corto. Efectivamente, a veces necesitamos que si no se cumple una condición se ejecute algo distinto. La palabra que falta es ELSE (traducida es «de lo contrario...»). Lo que viene a continuación de ELSE se ejecutará sólo si la condición expresada en el IF no se cumple. Por ejemplo, aquí tenemos un modo elegante de codificar nuestro programa de saludo:

```
10 input "Dame tu nombre ";a$
20 if a$<>"Fabián" then print "Hola, ";a$ else print "A ti no te
saludo, Fabián."
30 if a$<>"fin" then 10
40 print "Fin de programa"
```

Lo que pasa en la línea 20 es muy sencillo. Basta con leerlo en nuestro idioma: *si a$<>"Fabián" entonces print "Hola, ";a$ en caso contrario, print "A ti no te saludo, Fabián."*. Ahora tenemos un programa que saluda indefinidamente (excepto a Fabián) y que sólo termina cuando escribamos "fin".

La utilización de IF-THEN es la clave del funcionamiento de la mayoría de los programas. Su equivalente, aunque un poco más pobre, es ON-GOTO.

DE LA VARIABLE DEPENDE...

Dado que los programas tienen una cierta estructura interior, podemos diferenciar distintos sectores del mismo delimitando un determinado conjunto de líneas. Si tenemos, por ejemplo, un programa que guarda una información, habrá una parte del programa dedicada a consultas, otra a renovación, etc.

Si numeramos cada uno de los bloques que componen las diversas funciones del programa, independientemente de las líneas que ocupan, tendremos el bloque 1, el 2... Lo que BASIC no ofrece con ON-GOTO es precisamente llevar la ejecución del programa a cada uno de esos bloques.

ON asocia el valor de una variable (mencionada tras él) con un número de línea (escrito tras el GOTO). La expresión completa se escribe: ON v GOTO n1, n2, n3, n4, ... (v es una variable numérica y las "enes" son números de línea). Cuando BASIC se encuentra con esto tomará el valor de la variable v. Si v=1, la instrucción equivale a GOTO n1. Si v=2, la instrucción equivale a GOTO n2. Si v supera en valor el número de líneas mencionadas se ignora la instrucción.

Recordaremos lo que hace GOTO si encuentra una referencia a una línea inexistente: emite el error *Line does not exist.*

```
10 print "1-decir hola"
20 print "2-decir adiós"
30 print "3-no decir nada"
40 input "Elige opción ";opción
50 on opción goto 70,80,90
60 goto 10
70 print "hola":goto 10
80 print "adiós":goto 10
90 print "algo":goto 10
```

La línea 60 parece que tiene poco sentido, pero, como ya hemos dicho, ON-GOTO ignora cualquier número si el valor indicado supera las referencias. Si escribimos 4 la línea 50 no se ejecutará, y si no hubiéramos puesto la 60 el programa produciría el mismo resultado que el 1.

La práctica, como siempre, es el mejor método para dominar estas instrucciones.

TERMINAR... POR LAS BUENAS

Cuando hemos introducido la utilización de GOTO, IF-THEN y ON-GOTO ha aparecido un pequeño problema: terminar el programa. Ya sabemos que BASIC termina la ejecución del mismo cuando no encuentra más líneas. Esto es lo normal, pero en muchas ocasiones comprobaremos que resultaría muy cómodo terminar el programa en otra línea, no la última.

BASIC, cómo no, ofrece un medio para detener el programa: END (en inglés, extremo, fin). Cuando durante la ejecución del programa se llegue a la instrucción END el programa se detendrá sin más, así de sencillo. De manera que el siguiente programa termina en la línea 20 y el PRINT de la 30 se ignora:

```
10 print "Querida Matilde: llámame por teléfono."
20 end
30 print "Enviad cestita con bombones."
```

Podemos utilizar END junto con IF para determinar si un programa debe continuar o no su ejecución. Está claro que END puede estar al final del programa, pero también en cualquier otro lugar.

Algunos libros aconsejan escribir END al final del programa aunque termine precisamente en el final. La razón es, además de un intento por hacer las cosas bien, una costumbre para quien maneja ordenadores que comparten varias tareas al mismo tiempo (grandes ordenadores, claro). En estos casos, es necesario indicar cuándo se ha terminado un proceso claramente, es decir, con END. En el Amstrad no resulta necesario, y si se pone será simplemente por «hacer las cosas bien».

Existe otra forma de detener la ejecución de un programa, pero es «brusca» y sólo se utiliza durante la corrección de programas, cuando queremos que se detengan en un punto determinado. Cuando BASIC la encuentra durante la ejecución tiene el mismo resultado que cuando se pulsa la tecla STOP o ESC. La instrucción de detención es STOP. El mensaje que BASIC emite es «Break in n», donde *n* es el número de línea en el cual se encuentra el STOP. No es ésta la forma adecuada de dar por terminado un programa; repetimos que se utiliza normalmente en la depuración de los mismos.

LISTADO 1

```
10 input "Escribe una cadena ";a$
20 for a=1 to len(a$)
30 print left$(a$,a)
40 next
50 for a=1 to len(a$)
60 print right$(a$,a)
70 next
80 input "Otra cadena (s/n) ";x$
90 if x$="s" then goto 10
100 if x$<>"n" then goto 80
110 print "Se acabo"
120 end
```

LISTADO 2

```
1 rem calculo de la altura a la que asciende un liquido
2 rem por un tubo capilar, debido a su tension superficial
3 rem mediante la ley de difusion.
10 print "Radio del capilar";
20 input r
30 if r>0 then 60
40 print "Fin."
50 end
60 print "Masa especifica ";
70 input m
80 print "Tension superficial ";
90 input t
100 print "Angulo del menisco en radianes ";
110 input a
120 h=2*t*cos(a)/(r*m*9.8)
130 print "Altura: ";h
140 print
150 goto 10
```

LISTADO 3

```
1 rem calculo de la potencia electrica de una presa
2 rem datos:
3 rem       caudal de agua
4 rem       altura del salto en metros
5 rem       rendimiento del generador en %
10 input "Caudal ";c
30 if c>0 then 60
40 print "Fin."
50 end
60 input "Altura ";h
80 input "Rendimiento ";r
100 p=c*9.8*h*r*0.735/7500
110 print "Potencia: ";p;"Kw."
120 goto 10
```

LISTADO 4

```
1 rem programa que escribe los n primeros
2 rem numeros de la serie de Fibonacci
3 rem Un elemento de esta serie es la
4 rem suma de los dos que le preceden.
```

```
10 k=2
20 n1=0
30 n2=1
40 print "Cuantos numeros desea?"
50 input n
60 print "Serie (";n;")"
70 print "0";"1";
80 if k<n then 100
85 print "Fin de la serie."
90 end
100 k=k+1
110 n3=n2+n1
120 print n3;
130 n1=n2
140 n2=n3
150 goto 80
```

LISTADO 5

```
10 rem supuesta subrutina de captacion por el teclado
20 rem de un nombre especial que debe cumplir ciertas
30 rem caracteristicas:
40 rem 11 caracteres en total. Los ocho primeros deben
50 rem ser letras mayusculas, el noveno un
60 rem punto, y los ultimos tres, tambien letras.
70 input "Escriba el nombre ";nom$
80 nom$=upper$(nom$)
90 if len(nom$)<>11 then print "Numero de caracteres no
correcto. Repita.":goto 70
100 if mid$(nom$,9,1)<>"." then print "Falta el punto en el
noveno lugar. Repita.":goto 70
110 nom$=left$(nom$,8)+rigth$(nom$,3):rem eliminamos el punto de
la cadena
120 for a=1 to len(nom$):rem hacemos un rastreo para buscar mas
puntos
130 if mid$(nom$,a,1)="." then print "Se ha introducido mas de
un punto. Repita.":goto 70
140 next:rem fin del rastreo
150 for a=1 to len(a$):rem hacemos un rastreo para ver si solo
hay letras mayusculas
160 codigo=asc(mid$(nom$,a,1)):rem codigo de la letra
170 if codigo<65 or codigo>90 then nom$="invalido"
180 next
190 if nom$="invalido" then print "Solo se admiten letras.
Repita.":goto 70 else print "Nombre aceptado":end
```

LISTADO 6

```
10 rem funcionamiento de ON-GOTO
20 input "Escribe un numero de 1 a 3 ";num
30 on num goto 50,70,90
40 print "Numero no valido. Reescribe.":goto 20
50 input "Cuando murio Beethoven ";fecha
60 if fecha=1827 then print "Bien!":goto 20 else print
"Mal.":goto 50
70 input "Cuantos bytes tiene una K ";bytes
80 if bytes=1024 then print "Bien!":goto 20 else print
"Mal.":goto 50
90 input "Cuanto es 2 elevado a 8 ";mucho
100 if mucho=256 then print "Bien":goto 20 else print
"Mal.":goto 50
```

EJERCICIOS

61. ¿Qué instrucción plantea decisiones en los programas?

A) GOTO
B) IF... THEN
C) El Gobierno.

62. ¿Cuál es el valor numérico asignado a un hecho cierto por un ordenador Amstrad?

A) −1.
B) 1.
C) 0.

63. ¿Es posible efectuar evaluaciones lógicas con cadenas?

A) Sí, si se rompen previamente.
B) No.
C) Sí.

64. ¿Cuál será el resultado de escribir PRINT "ORDENADOR" = "Ordenador"?

A) Cansancio muscular.
B) 0.
C) −1.

65. ¿Cuándo se ejecuta una instrucción situada tras un ELSE?

A) Nunca. Es un comentario al programa.
B) Cuando la condición propuesta en el IF se cumple.
C) Cuando la condición propuesta en el IF no se cumple.

66. ¿Qué instrucción o instrucciones emplearíamos para bifurcar la ejecución a diversos sectores del programa según el valor de una variable?

A) Contrataríamos un guardia urbano.
B) Varios IF... THEN GOTO...
C) Un ON... GOTO...

67. ¿Es posible que un programa contenga más de una instrucción END?

A) Sí.
B) Sólo si es un mentiroso.
C) No.

68. ¿Es necesario poner END en la última instrucción del programa?

A) No.
B) Sí.
C) Sólo si queremos tener que volver a apagar y encender.

69. ¿Es STOP instrucción de efectos idénticos que END, como el interrogante (?) y PRINT?

A) Sí.
B) No.
C) Cansa más escribirlo porque tiene una letra más.

MANEJO DE DATOS

lmacenar información es una de las tareas más importantes del ordenador. Realmente, lo único que pueden hacer es guardar y operar con datos, cuyos resultados se volverán a guardar.

Según hemos visto hasta ahora, los datos (numéricos o alfanuméricos) se almacenan en variables. El problema surge cuando necesitamos mucha información sobre la cual hay que operar. Evidentemente, si empezamos a poner nombres de variables, no terminamos nunca.

Para hacer las variables más manejables se dimensionan. Esto significa que establecemos una variable con subíndices, es decir, para *a* tendremos a(1), a(2), a(3)... Imaginemos lo cómodo que resultaría manejar miles de variables tan sólo cambiando el subíndice, a través de un FOR-NEXT o algún otro medio.

La instrucción para dimensionar variables es DIM y la forma general de escribirlo es DIM v1(n1), v2(n2)... donde *v1* y *v2* son varia-

bles numéricas y *n1, n2* indican el máximo subíndice alcanzable que se impone.

Para variables alfanuméricas se opera igual, pero intercalando el signo $ entre la variable y el primer paréntesis.

Así, si queremos 50 valores para la variable *grande,* escribiremos DIM grande(50). Si queremos procesar 20 nombres a través de variables con subíndice (lo cual es lo más recomendable) escribiremos DIM nombre$(20).

Vamos a ver cómo DIM simplifica enormemente ciertos procesos repetitivos y que afectan a todas las variables bajo subíndice. Imaginemos un programa que toma cinco nombres, los pasa a mayúsculas, limita su longitud a 10 caracteres y a continuación los muestra. Sin utilizar DIM, sería más o menos así:

```
10 input "Dame el nombre 1 ";nomb1$
20 input "Dame el nombre 2 "; nomb2$
...
50 input"Dame el nombre 5 ";nomb5$
60 nomb1$=upper$(nomb1$)
70 nomb2$=upper$(nomb2$)
...
100 nomb5$=upper$(nomb5$)
110 if len(nomb1$)>10 then nomb1$=left$(nomb1$,10)
120 if len(nomb2$)>10 then nomb1$=left$(nomb2$,10)
...
160 print nomb1$
170 print nomb2$
180 print nomb3$
190 print nomb4$
200 print nomb5$
```

El resultado, como vemos, es excesivamente largo, lo cual implica gran cantidad de memoria utilizada, además de un aspecto del listado mucho más confuso. Con DIM el programa se reduciría a:

```
10 dim nomb$(5)
20 for a=1 to 5
30 print "Dame el nombre ";a;
40 input nomb$(a)
50 next a
60 for a=1 to 5
70 nomb$(a)=upper$(nomb$(a))
80 if len(nomb$(a))>10 then nomb$(a)=left$(nomb$(a),10)
```

```
90 print nomb$(a)
100 next
110 end
```

Veamos las tareas que BASIC debe efectuar aquí. Primero dimensiona nomb$ con un subíndice cuyo valor máximo será cinco. Al empezar la ejecución del bucle, toma nomb$(1), cuyo valor se introducirá a través del teclado. Después vendrá nomb$(2), etc... Al entrar en el segundo bucle, se efectuará el proceso de "paso a mayúsculas" y "recorte" para todos los subíndices, tras lo cual se imprimen.

DIM amplía considerablemente las posibilidades de manejo de información, pero no termina aquí su capacidad. BASIC nos proporciona variables «multidimensionales»: podemos asignar a una variable más de un subíndice, y el único límite es la capacidad de la memoria del ordenador; BASIC admite cualquier número de dimensiones sin problema.

Si queremos, por ejemplo, guardar las coordenadas espaciales (es decir, en 3 dimensiones) de cuatro puntos escribiremos DIM puntos(4,3).

El punto 1 tendrá entonces la variable *puntos(1,1)* como coordenada x, *puntos(1,2)* como coordenada y, *puntos(1,3)* de coordenada z, y así sucesivamente.

Aprovechamos para advertir que DIM admite operar con el subíndice cero. Es decir, en los ejemplos anteriores serán variables válidas nomb$(0), puntos (0,1), etc... Esto se ajustará o no a nuestro programa; en el último caso basta con ignorar las variables con subíndice de valor 0, pero tengamos en cuenta que entonces se desperdicia memoria, puesto que dichas variables han sido creadas y no se utilizan. (En el PCW, se puede remediar esto. Lo veremos en el número 15 de esta colección.)

El uso de subíndices es tan común que BASIC admite la operación con variables de un subíndice cuyo valor no supere 10, sin tener que dimensionarla previamente con DIM. Es decir, si en un programa empezamos a manejar por las buenas una variable como q(5), BASIC asumirá esto, y supondrá en adelante que se ha escrito DIM q(10), aunque no haya sido así.

Así, resulta que en nuestro ejemplo de los cinco nombres sobraría el DIM escrito al principio. Pero como ya hemos dicho alguna vez, conviene hacer las cosas bien. Queremos decir con esto que, al omitir el DIM, BASIC tomará implícitamente un subíndice máximo 10, lo cual es un derroche de memoria, puesto que sólo tomamos hasta 5, contando además que desaprovechamos en subíndice 0.

Aprovecharemos este conflicto sobre memoria para mencionar un método que proporciona la memoria libre disponible para BASIC. Se trata de FRE(n) (viene de free, libre, desocupado) donde *n* es cualquier tipo de argumento. Por ejemplo:

```
print fre(0)
print fre("Comamos cereales")
```

Ambas líneas dan como resultado la memoria libre en bytes. Como indicación diremos que una instrucción como dim a(100) ocupa unos 400 bytes.

La memoria no parece normalmente un problema muy importante; sin embargo, cuando confeccionemos programas grandes nos daremos cuenta de que son lentos, y a veces sobrepasan la capacidad de la máquina. Las variables dimensionadas ocupan tanto que conviene borrarlas en cuanto dejen de ser útiles para el programa.

El borrado se efectúa con ERASE v1, v2, v3... donde v1, v2... son los nombres de las variables que han sido dimensionadas, pero no se especifican los subíndices de las mismas. Por ejemplo, para anular el efecto de dim sigh(3,10,2,2) se escribe erase sigh y no erase sigh(3,10,2,2).

READ, DATA Y RESTORE

Imaginemos que un programa tiene cierta información almacenada previa a la ejecución. Por ejemplo, un programa con calendario perpetuo debe «saberse» los meses (los cuales serán datos fijos) y la asignación en las correspondientes variables se efectuará al principio del programa.

Ya sabemos que lo más cómodo en estos casos es establecer un subíndice: DIM mes$(12) será la instrucción apropiada. Pero ahora debemos asignar los valores. Lo haríamos así:

```
...
40 mes$(1)="Enero"
50 mes$(2)="Febrero"
```

Lo cual resulta excesivamente pesado.

Pero hay una forma más sencilla de hacer todo esto, a través de READ (en inglés, leer) y DATA (datos). READ v1, v2, v3... ordena a BASIC que busque un dato constante y lo asigne a *v1*, que busque otro y lo asigne a *v2*, etc...

La forma de declarar datos fijos es:

DATA dato1, dato2, ...

Por ejemplo, el programa:

```
10 read a,b,c$
20 data 10,20,"Mantequilla de cacahuete"
30 print a,b,c$
```

Asigna a *a* el valor 10, a *b* el valor 20 y a *c$* esa cadena. Podemos comprobarlo con la línea 30.

Este sistema es muy cómodo, pero implica la posibilidad de cometer ciertos errores. En concreto, cuando se intenta asignar un valor numérico a una variable alfanumérica, o una cadena a una variable numérica a través de READ-DATA, BASIC emite el error «Syntax error».

Combinando DIM, READ y DATA obtenemos la forma más rápida de asignar a las variables información fija. Por ejemplo, nuestro programa de meses sería ahora así:

```
10 data Enero,Febrero,Marzo,Abril,Mayo,Junio
20 data Agosto, Septiembre, Octubre, Noviembre, Diciembre
30 dim mes$(12)
40 for a=1 to 12
50 read mes$(a)
60 next
```

Este ejemplo nos muestra tres cosas. Primero, que los DATAs se pueden colocar en cualquier parte del programa, antes o después del READ, puesto que BASIC se encargará de buscarlos cuando llegue el momento, sin que ello deba preocuparnos más. Segundo, que no hace falta encerrar entre comillas las cadenas (esta es la única excepción), ya que cuando BASIC lee en el READ «sabe» lo que debe recoger en el DATA. Esto puede llevar a error, puesto que un número cualquiera puede ser recogido por una variable alfanumérica. Mucho cuidado.

Y, por último, vemos que los datos fijos pueden distribuirse en distintas líneas (todas empezarán por DATA) para hacer más cómoda la lectura. BASIC se encarga de memorizar en cada momento a qué dato le corresponde ser leído la próxima vez, gracias a un puntero.

Con RESTORE podemos mover ese puntero a nivel de líneas. Vamos a poner un ejemplo en el que dos variables dimensionadas necesitan los mismos datos:

```
10 dim a(15),b(15)
20 for s=1 to 15
30 read a(s),b(s)
40 next s
50 data 1,3,5,7,9,2,4,6,8,10,5,4,3,2,1
60 data 1,3,5,7,9,2,4,6,8,10,5,4,3,2,1
```

Para ahorrarnos la línea 60 (que es al fin y al cabo una repetición inútil) escribiremos:

```
10 dim a(15),b(15)
20 for s=1 to 15
30 read a(s)
40 next
50 restore 90
60 for s=1 to 15
70 read b(s)
80 next
90 data 1,3,5,7,9,2,4,6,8,10,5,4,3,2,1
```

El RESTORE en este caso se ha encargado de dejar el puntero de datos en la línea 90, para que BASIC lea allí de nuevo. En el caso de que se hubiera omitido, BASIC habría intentado leer datos más allá de esa línea, pero no habría encontrado nada. El mensaje de error que se emite en estos casos, es decir, cuando faltan datos o no se han escrito, es «Data exhausted». En el caso de que RESTORE se refiera a una línea inexistente obtendremos el ya conocido error «Line does not exist».

LISTADOS

LISTADO 1

```
1 rem programa que ordena de menor a mayor
2 rem una cantidad de numeros no superior
3 rem a cien.
10 dim x(100)
20 input "Cantidad de numeros: ";n1
30 for n=1 to n1
35 input "Escriba el numero ";x(n)
40 next
40 i=0
50 i=i+1
60 if i=n1 then 150
70 if x(i)<=x(i+1) then 50
80 m=x(i)
85 x(i)=x(i+1)
89 x(i+1)=m
90 j=i
100 j=j-1
110 if j=0 then 50
120 if x(j)<=x(j+1) then 100
130 m=x(j)
135 x(j)=x(j+1)
139 x(j+1)=m
```

```
140 goto 100
150 for n=1 to n1
155 print x(n)
159 next
160 end
```

LISTADO 2

```
10 rem Programa de ordenacion
20 rem de una lista de nombres
30 rem por orden alfabetico
40 input "Cuantos nombres tiene la lista ";num
50 num=int(num)
60 if num<1 then print "Esto no es una lista.":goto 40
70 if num>100 then print "Esta es una lista muy grande.":goto 40
80 dim lista$(num)
90 for a=1 to num
100 print "Escribe el nombre numero ";a
110 input lista$(a)
120 next
130 rem ahora empiezo a ordenar
140 indicador=0
150 for a=1 to num-1
160 if lista$(a) > lista$(a+1) then cambio$ = lista$(a) :
lista$(a) = lista$(a+1):lista$(a+1)=cambio$:indicador=1
170 rem lo que hemos hecho en la linea 160 es
180 rem cambiar el contenido de lista$(a) y
190 rem lista$(a+1)
200 next a
210 if indicador=1 then goto 140
220 rem si el indicador es 0, no ha habido cambios
230 rem y por tanto la lista esta ordenada
240 for a=1 to num
250 print lista$(a)
260 next
270 end
```

LISTADO 3

```
1 rem conjugacion de verbos regulares con ayuda de DATA
10 dim p$(6)
20 dim t$(6)
30 for a=1 to 6
40 read p$(a),t$(a)
50 next
60 data yo,o,tu,as,el,a,nosotros,amos,vosotros,ais,ellos,an
70 input "Verbo ";v$
80 l=len(v$)
90 r$=left$(v$,l-2)
100 print "Presente:"
110 print
120 for n=1 to 6
130 print p$(n);" ";r$;t$(n)
140 next
150 print
160 print "Preterito imperfecto:"
170 print
180 print p$(1);" ";r$;"aba"
190 for n=2 to 6
200 print p$(n);" ";r$;"ab";t$(n)
210 next
220 end
```

LISTADO 4

```
5 rem calculo de la media aritmetica y multiplos de 3
10 data 2,7,3,14,12,9,"media=","multiplos="
20 s=0:rem s es la suma de los numeros
30 for a=1 to 6
40 read n
50 s=s+n
60 next a
70 m=s/6:rem m es la media aritmetica
80 read m$,t$
90 restore
100 c=0: rem c es el contador de mutiplos de 3
110 for b=1 to 6
120 read n
130 e=n/3
135 rem la parte entera de e es int(e)
140 if e=int(e) then c=c+1
150 next
155 rem resultados
160 print m$,m
170 print t$,c
180 end
```

EJERCICIOS

70. ¿Qué instrucción se emplea para generar variables suscritas o con subíndice?

A) INPUT.
B) DIM.
C) Las del cupón de suscripción adjunto.

71. ¿Qué longitud por defecto adopta un conjunto de variables suscritas?

A) 10.
B) 5.
C) Ninguna.

72. ¿Es posible emplear variables suscritas de varias dimensiones?

A) Está por demostrarse.
B) Sí.
C) No.

73. ¿Qué función se emplea para averiguar la cantidad de memoria disponible?

A) MEMORY.
B) QUEMEMORIAQUEDA$.
C) FRE.

74. ¿Borra ERASE variables no suscritas?

A) Sí.
B) No.
C) Borra lo que le echen.

75. ¿Cuál es la misión de DATA?

A) Almacenar datos en líneas de programa.
B) Dar pie para hacer preguntas estúpidas.
C) Suministrar datos a bucles FOR-NEXT.

76. ¿Es posible situar una instrucción READ antes de su correspondiente DATA?

A) Sí.
B) No.
C) BASIC se encargará de ponerla después.

77. ¿Qué instrucción restaura el puntero de READ al primer dato de la primera DATA del programa?

A) Ninguna.
B) FRE.
C) RESTORE.

LAS SUBRUTINAS

Los procesos más corrientes siempre se repiten varias veces a lo largo de un programa. Son cosas muy simples, como por ejemplo escribir los textos en la pantalla con un determinado formato, etc.

Pero repetir significa ocupar memoria innecesariamente. El remedio es crear *subrutinas* o *subprogramas*. Un subprograma es un fragmento de éste que se ejecuta varias veces, pero que sólo está escrito una vez. Puede utilizar variables, condiciones, datos fijos y cualquiera de las instrucciones disponibles en BASIC.

Las subrutinas se escriben «a un lado», normalmente al final del programa. Para acceder a ellas se utiliza la instrucción GOSUB, que equivale a la instrucción GOTO pero con una ligera diferencia. Cuando BASIC encuentra un GOSUB, desvía la ejecución del programa hacia la subrutina, tal y como lo haría con un GOTO.

La diferencia está en que memoriza la línea de la cual parte y cuando se llegue al final de la subrutina la ejecución retornará al punto en el cual se la dejó, es decir, justo después del GOSUB. La forma

de decir a BASIC que la subrutina ha terminado es escribir RETURN (vuelve).

Veamos ahora un ejemplo. Se trata de un programa que presenta ciertos datos en la pantalla, pero centrados y en mayúsculas.

```
10 a$="Programa para calcular media aritmética"
20 print:print tab((80-len(a$))/2);upper$(a$)
30 a$="Escriba el primer número"
40 print:print tab((80-len(a$))/2);upper$(a$)
50 input primernum
60 a$="Escriba el segundo número
70 print:print tab((80-len(a$)/2);upper$(a$)
80 input segundonum
90 media=(primernum+segundonum)/2
100 a$="La media es "+str$(media)
110 print:print tab((80-len(a$)/2);upper$(a$)
120 end
```

El programa calcula la media aritmética de dos números, centrando todos los mensajes. Para centrar se calcula la distancia en una pantalla de 80 columnas; se resta la anchura de la pantalla a la longitud de la cadena. El resultado es el número de columnas libres que habrá que dividir entre dos para saber lo que corresponde a cada lado de la cadena.

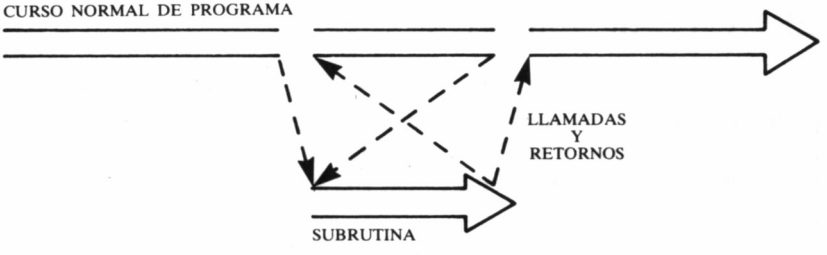
Como podemos observar, las líneas 20, 40, 70 y 110 son la misma, repetida inútilmente. Con la utilización de GOSUB el programa quedaría así:

```
10 a$="Programa para calcular media aritmética"
20 gosub 130
30 a$="Escriba el primer número"
40 gosub 130
50 input primernum
60 a$="Escriba el segundo número"
70 gosub 130
80 input segundonum
90 media=(primernum+segundonum)/2
100 a$="La media es "+str$(media)
110 gosub 130
120 end
130 print:print tab((80-len(a$)/2);upper$(a$)
140 return
```

Hemos creado una subrutina en la línea 130 y su función es imprimir la variable *a$* centrada y en mayúsculas. El programa tiene más líneas, pero es tan sólo un ejemplo y además su lectura es ahora mucho más clara, y modificarlo es mucho más sencillo. Si quisiéramos que no fuera en mayúsculas tendríamos que cambiar la línea 130. En el programa anterior la 20, la 40, etc.

El camino que sigue BASIC es algo complicado; cuando llega a la línea 20 se apunta el 20 y pasa a la línea 130. Al encontrarse con RETURN vuelve a la 20. Como GOSUB era la única instrucción de la línea 20, pasa a la 30. Al llegar a la 40 se apunta el 40 y pasa a la 130. Cuando encuentre el RETURN volverá a ella y así sucesivamente. Podemos decir que es un GOTO «de ida y vuelta» con las ventajas que eso supone.

Pero, cómo no, el añadir más instrucciones supone introducir más errores. Si gosub no encuentra el RETURN seguirá indefinidamente y además memorizando la línea de partida. En el caso de que esto se repita muchas veces, es decir, que queden muchos GOSUBs pendientes de sus correspondientes RETURNs, puede llevar a bloquear el ordenador. El otro error ya debemos deducirlo, es un viejo conocido: cuando la línea a la cual hace referencia el GOSUB no existe se emite el mensaje «Line does not exist».



8.9.1. Esquema general de llamadas y retornos a subrutinas.

Por otro lado, si BASIC encuentra un RETURN pero no existe ninguna línea «apuntada», detendrá la ejecución del programa emitiendo el mensaje «Unexpected RETURN» (lo que significa, más o menos, que eso no se lo esperaba).

Las subrutinas son el medio más corriente para estructurar un programa. Se dice que la calidad de codificación de un programa está en proporción directa al número de GOSUBs que contenga y en proporción inversa al número de GOTOs. Esta opinión es demasiado generalizada y, aunque lo que dice es cierto, debemos pensar siempre en nuestro programa en concreto.

Por ejemplo, el comienzo de un programa «super estructurado» de marcianos sería más o menos así:

```
10 gosub 200:rem subrutina de impresión de paisaje lunar
20 gosub 400:rem colocar marcianos
30 gosub 750:rem colocar nave
40 gosub 1200:rem explorar el teclado
50 gosub 1290:rem modificar posiciones
60 goto 20:rem repetición del proceso
...(subrutinas)...
```

La ventaja de estos programas es la claridad; basta con leer las primeras líneas para saber el funcionamiento general, para esto son de gran ayuda los REM. Pero debemos recordar que las subrutinas se escriben cuando deben utilizarse varias veces a lo largo de la ejecución. En el ejemplo anterior no necesitamos realmente los GOSUB, bastaría con escribir en lugar de ellos la rutina correspondiente. El programa sería más rápido, y con los correspondientes REM no ha de ser mucho más confuso. Como vemos, la utilización de GOSUB depende no sólo de su finalidad, sino que también es cuestión de gustos.

ENCADENAR SUBRUTINAS

Los bucles pueden anidarse. Las llamadas a subrutinas también pueden repetirse, es decir, una subrutina puede llamar a otra, que será subrutina suya. BASIC recuerda todos los números de línea hasta un límite al cual jamás llegaremos: 100 subllamadas en el PCW y 83 en el CPC.

Ya conocemos el peligro que supone esto: un GOSUB sin su correspondiente RETURN hace que BASIC contenga un número de

línea en su memoria interna. Demasiados olvidos suponen el bloqueo de BASIC o, en caso contrario, una ejecución lenta. De todos modos, en muchas ocasiones BASIC llegará a su límite interno. En ese caso emite el mensaje «Memory full» (memoria completa). No se refiere en este caso a la memoria total, sino a la empleada internamente para recordar números de línea, FORs, WHILEs, etc.

Como ocurre con las demás instrucciones de BASIC, GOSUB puede incluirse en las expresiones condicionales:

```
if gastos>ingresos then gosub 1000:
    rem mentalización quiebra
```

Pero otra forma disponible de hacer saltos condicionales a una subrutina es con la expresión ON GOSUB, cuyo resultado es el mismo que con ON GOTO, pero el salto queda pendiente de un RETURN. Cuando BASIC lo encuentre volverá a la instrucción más próxima y posterior al ON GOSUB.

Ya sabemos cómo funciona ON v GOSUB n1, n2, n3...: Dependiendo del valor de la variable numérica *v* (ha de ser entero positivo) BASIC efectuará la llamada a una u otra rutina. Para v=1, equivale a GOSUB n1; para v=2, GOSUB n2... Si v vale 0, se ignora la instrucción completa. Si la línea a la cual hace referencia ON no existe, BASIC emitirá el error «Line does not exist».

LISTADOS

LISTADO 1

```
10 rem Juego de preguntas. Una subrutina se encarga
20 rem de recoger las respuestas ("si" o "no")
30 dim preg$(10)
40 dim resp$(10)
50 for a=1 to 10
60 read preg$(a),resp$(a)
70 input "Dame un numero del 1 al 10 ";a
80 print preg$(a);"?"
90 gosub
100 if  s$=resp$(a)  then  print  "Bien!"  else  print  "No.  lo
correcto es responder ";resp$(a)
110 print "Quieres volver a jugar?"
120 gosub
130 if s$="SI" then goto 20 else end
140 input s$
150 s$=upper$(s$)
160 if s$<>"SI" and s$<>"NO" then print "Contesta  solo  'si'  o
'no'":goto 150
170 return
180 data Se puede borrar una RAM,SI,Se puede borrar una
ROM,NO,Se puede borrar una EPROM,SI,El Z80 trabaja con 16
bits,NO,El 8088 trabaja con 16 bits,SI
```

```
190 data El 68000 trabaja con 16 bits,NO,El 6502 es una pieza de
museo,SI,Este programa funciona bien,SI,Me prestas 30000
pesetas,SI,Te estas aburriendo,NO
```

LISTADO 2

```
10 rem programa para calcular
20 rem un coeficiente binomico
30 rem
40 rem la formula es a!/(b!*(a-b)!)
50 print "A debe ser mayor o igual que b"
60 input "Escriba valor de 'a' ";a
70 input "Escriba valor de 'b' ";b
80 if a<b then 50
90 n=a
100 gosub 500
110 x=f
120 n=b
130 gosub 500
140 y=f
150 n=a-b
160 gosub 500
170 z=f
180 r=x/(y*z)
190 print "El valor de ";a;" sobre ";b;" es igual a ";r
200 end
500 rem
510 rem subprograma para el calculo
520 rem del factorial de n
530 rem
540 f=1
550 for i=1 to n
560 f=f*i
570 next i
580 return
```

LISTADO 3

```
10 rem programa que calcula el area
15 rem de un triangulo o de un rectangulo
20 rem con dos niveles de subrutinas.
30 print "Escriba:"
40 print "1 para area de un triangulo"
50 print "2 para area de un rectangulo"
60 input "Opcion ";a
70 on a gosub 90,140
80 end
90 rem opcion 1
100 gosub 190
110 t=(b*h)/2
120 print "El area del triangulo es ";t
130 return
140 rem opcion.2
150 gosub 190
160 r=b*h
170 print "El area del rectangulo es ";r
180 return
190 rem subrutina para entrada de datos
200 input "Base ";b
210 input "Altura ";h
220 return
```

EJERCICIOS

78. ¿Qué es una subrutina?

A) Una subruta chiquitita.
B) Un sector de programa accedido repetidas veces mediante GOSUB.
C) Una técnica de programación.

79. ¿Qué instrucción se emplea para retornar de una subrutina al punto en que se la llamó?

A) VUELVEAQUI.
B) RETURN.
C) Otro GOSUB.

80. ¿Es posible el anidamiento de subrutinas?

A) Sí.
B) No.
C) Sólo si apagamos y encendemos el ordenador previamente.

81. ¿Se pueden utilizar subrutinas para llamadas a sectores de programas, aunque éstas sólo se produzcan una vez?

A) No.
B) Depende de las ganas que tengamos de liar las cosas.
C) Sí.

82. ¿Es posible emplear la instrucción GOSUB para instrucciones ON, como con GOTO?

A) Sí.
B) No.
C) Sí. Y también existe el ON... FOR, ON... END, ON... DA CORTA.

ALGUNOS DETALLES SUELTOS...

uedan por ver algunas instrucciones de difícil clasificación, especialmente porque ya se alejan un poco de lo más común del BASIC; es decir, mucho de lo que explicamos en este capítulo no sirve, o al menos no de igual manera, en otros ordenadores. De todas formas, diremos en su momento otras expresiones que utilizan distintas versiones BASIC, siempre que sean equivalentes a las de Amstrad en cuanto a su resultado.

DEFINIR FUNCIONES

BASIC tiene muchas funciones, pero muchas veces necesitaremos una en concreto para un programa específico. Podemos definirla, y tendrá tantos argumentos como necesitemos, numéricos o alfanuméricos, todo a nuestro gusto. Estas funciones se llamarán «funciones definidas por el usuario» *(user functions)*.

La instrucción para definir es DEF FN *(DEFine FuNction)*. Debe

escribirse antes de ser utilizada por el programa, normalmente se definen todas las funciones en las primeras líneas. Para obtener el resultado de una función definida se utiliza FN.

Por ejemplo, si queremos una función con dos argumentos cuyo resultado sea la media de ambos, escribiremos:

```
def fn media(x,y)=(x+y)/2
```

Creamos con ello una función llamada *media* que necesita dos argumentos numéricos (puesto que las variables puestas en ella lo son) y cuyo resultado es también numérico.

Para comprobar el resultado de esta definición podemos escribir:

```
print fn media(4,10)
```

Que dará como resultado 7.

Si queremos una función que devuelva el número de columnas a la que debemos tabular una cadena para que quede centrada, escribiremos:

```
def fn espacios(f$)=80-len(f$)/2
```

Para utilizarla, podemos escribir el siguiente programa:

```
10 def fn espacios(f$)=(80-len(f$)/2)
20 input "Dime algo y yo lo centro";a$
30 print tab(fn(a$));a$
40 goto 20
```

Las variables mencionadas en la definición no tienen demasiada importancia. En realidad, no necesitamos emplearlas para referirnos a esa función, incluso podemos dar valores directamente y no a través de variables. Su único cometido es decir a BASIC si lo que ha de escribirse allí es numérico o no.

Al definir funciones hay que tener un poco de cuidado. BASIC las memoriza al leerlas en el programa, pero las olvida fácilmente, por ejemplo cuando se modifica una línea, se borran las variables, etc. En ese caso, y si se pide el resultado de una función "olvidada", BASIC emitirá el mensaje *Unknown user function* (función del usuario desconocida). El mismo error se obtendrá si pretende utilizarse una función no definida previamente.

NÚMEROS ALEATORIOS... ¿O NO?

BASIC dispone de una función para generar números aleatorios entre el 0 y 1 (el uno excluído). Los números aleatorios son de gran utilidad en simulación de ensayos científicos y en cosas menos serias como los juegos de marcianos. El caso es que siempre hace falta en alguna parte un número inventado.

La función es RND y su funcionamiento así de simple:

```
print rnd
```

El resultado será un número entre 0 y 0.9999999.

Para ajustar este margen al que nosotros deseamos se utiliza la fórmula li+(rnd*(ls−li+1)), donde *li* es el límite inferior y *ls es el* límite superior del margen. Por ejemplo, para simular una tirada de dados debemos escribir:

```
print 1+rnd*6
```

Lo cual da un número de margen 1-6. Pero como los dados no dan decimales, tenemos que redondear a entero:

```
print int(1+rnd*6)
```

Si repetimos varias veces el PRINT veremos que el resultado es aleatorio. Para comprobarlo basta con teclear el siguiente programa:

```
10 input "Cuantas tiradas";n
20 for a=1 to n
30 print int(1+rnd*6)
40 next a
50 end
```

Podemos complicar la expresión para ajustarnos a nuestras necesidades, por ejemplo números pares entre 10 y 20:

```
print int((rnd*6)+5)*2
```

El problema es que RND no es totalmente aleatorio. Los programas que lo explotan al máximo acusarán repeticiones que nada tienen que ver con lo aleatorio. Esto se debe a que BASIC emplea fórmulas, más o menos complejas según la versión, para obtener un número aleatorio a partir del anterior.

Visto de esta manera, los números obtenidos a partir de RND

pertenecen a una lista que, aunque muy extensa, no deja de ser una lista y, por tanto, no aleatoria.

Como lo que hace BASIC es tomar el siguiente elemento de la lista, necesitamos una instrucción que lo desbarate un poco. Esa es la función de RANDOMIZE. Esta instrucción va seguida normalmente de un argumento numérico que se encarga de «desorientar» a BASIC para que no tome el siguiente elemento, sino otro. Tecleemos:

```
randomize
Random number seed?
```

BASIC pide un número para desordenar, llamado *semilla (seed)*. Si hubiéramos incluído el número, BASIC no habría necesitado intervención posterior:

```
randomize 2371
Ok
```

La regularidad de RND no ha de preocuparnos excesivamente en el caso de que nuestro programa lo utilice pocas veces. Pero si se trata de generar laberintos aleatorios, o repetir ensayos, habrá que acudir al usuario para introducir un número antes del proceso con RND. En un juego, algo tan sencillo como preguntar el nombre del jugador puede sernos útil; la longitud de la cadena puede darse a RANDOMIZE:

```
60 input "Dame tu nombre, intrépido jugador ";a$
70 randomize 40*len(a$):rem los números serán bien aleatorios
```

En algunos ordenadores para obtener un número aleatorio se escribe RND(0) o RAND en lugar de RND.

LEER EL TECLADO

Muchas veces necesitamos una respuesta breve, muy breve, del usuario. Por ejemplo, que pulse una tecla. Sólo una, y además cualquiera. Si lo hacemos con INPUT resulta molesto, porque al final debemos pulsar la tecla <RETURN>.

La función que explora el teclado es INKEY$ y su resultado es un carácter si en el momento de ejecutarse la instrucción había pulsada alguna tecla, o la cadena vacía si no se había pulsado ninguna.

Aunque parezca que no, esto nos será útil en muchas ocasiones, desde los típicos programas de marcianos...

```
90 if inkey$="p" then gosub 300:
   rem mover nave a la derecha
```

Hasta una pausa en un programa cualquiera...

```
110 print "Pulse cualquier tecla para continuar"
120 if inkey$="" then 120
```

En el último ejemplo la línea 120 espera a que INKEY$ no produzca la cadena vacía; es decir, hasta que se haya pulsado una tecla. En ese momento BASIC ignorará el resto de la línea para pasar a la siguiente (el programa continuará).

El CPC tiene otros métodos de captación por el teclado, que serán estudiados en el siguiente número sobre BASIC.

SIGUIENDO LOS PASOS...

Existen un par de instrucciones cuya función es activar y desactivar un sistema de seguimiento de la ejecución. Este sistema consiste en imprimir en la pantalla el número de la línea cuyo contenido está siendo ejecutado en ese momento. Conseguimos así saber en cualquier momento por dónde «anda» BASIC.

Estas dos instrucciones sólo se utilizan para corregir errores, nada más, pero son realmente de gran ayuda, sobre todo para corregir saltos y bucles; es decir, estructuras en las cuales se varía el curso normal de ejecución.

Para activar el mecanismo de seguimiento escribiremos TRON (viene de *trace on,* activar rastreo). No ocurrirá nada, pero al ejecutar un programa con RUN, además del resultado normal del propio programa en pantalla, aparecerán continuamente números de línea entre corchetes. Se trata de los números por los cuales va pasando BASIC.

La desactivación se realiza tecleando TROFF *(trace off).* La verdad es que la utilización de TRON dice muy poco en favor del programador, ya que significa reconocer que no se sabe ni por dónde se anda. No queremos decir con esto que esté prohibido su uso: quien esté aprendiendo, o quien trabaje con grandes programas tiene en TRON una de las mejores herramientas para detectar errores.

En otros ordenadores TRON se escribe TRACE ON y TROFF es TRACE OFF, o bien TRACE y NOTRACE. En general, pocas versiones de BASIC cuentan con esta magnífica ayuda.

Dos Grandes Desconocidos

Se trata de PEEK y POKE. La verdad es que su utilización provoca tantos resultados distintos que cada uno tiene su versión sobre lo que significa: hay quien piensa que se trata de órdenes para modificar las «vidas» y «armas» de los juegos comerciales; otros dicen que sirven para bloquear el ordenador...

Lo cierto es que PEEK y POKE son dos instrucciones bastante modestas, cuya única función es mantener un trato directo con el bloque de memoria que alberga al propio BASIC (en los Amstrad, se trata de un bloque de 64K).

PEEK devuelve el contenido de la dirección de memoria especificada:

```
print peek(20)
```

Da como resultado el octeto contenido en la dirección 20. Se trata de un número comprendido entre 0 y 255, ya que trabajamos con un microprocesador de 8 bits, el Z80A.

POKE introduce en la dirección de memoria especificada un dato, también entre 0 y 255:

```
poke 20,128
```

Introduce en la dirección 20 de memoria el valor 128.

POKE es una herramienta peligrosa. Modifica posiciones de memoria sin ninguna clase de condición y precaución. Por lo tanto, podemos modificar valores de gran importancia para el sistema operativo o para el intérprete, con lo cual obtendremos un comportamiento del ordenador extraño o, simplemente, se bloqueará o reinicializará. Es por esto que POKE «sirve para bloquear el ordenador».

Evidentemente, POKE puede afectar a nuestro propio programa, puesto que éste se encuentra almacenado en la memoria. Basta con saber las direcciones en las cuales se halla. Si modificamos un programa podemos provocar codificación errónea o bien cambiar sus características. Es lo que ocurre cuando tecleamos ciertos POKEs en un juego para que nos de vidas infinitas, etc.

En realidad, la utilización de PEEK y POKE es tan compleja que escapa del objetivo de este libro. Su verdadero poder está en la comunicación BASIC-código máquina.

Recomendamos que no se modifiquen posiciones de memoria. La mayoría de las veces lo que conseguiremos será bloquear el ordenador, y alguna que otra vez algún resultado espectacular. Lo que no haremos nunca es estropear la máquina, sean cuales sean los números tecleados.

Por el contrario, recomendamos que, en cuanto se domine con una considerable soltura el BASIC, se emplee PEEK para averiguar dónde se esconden las cosas: intérprete BASIC, nuestro propio programa, las variables, la memoria dedicada a la pantalla, etc. En el caso sel PCW será una tarea muy interesante teniendo en cuenta que BASIC esté instalado en RAM y, por tanto, se puede modificar con POKE. Podremos redefinir los mensajes de error, y muchas otras cosas, pero... todo a su tiempo. PEEK y POKE son las últimas instrucciones para aprender.

LISTADOS

LISTADO 1

```
5 rem calculo de logaritmos en distintas bases
10 def fn l(b,n)=log(n)/log(b)
20 data 7,8,4,3,6,4,3,9
30 for i=1 to 4
40 read b,n
50 j=fn l(b,n)
60 print "El logaritmo de ";n; "en base ";b; "es ";j
70 next
80 input desea calcular otro ";r$
90 if r$="si" then 100 else end
100 input "Numero ";h
110 input "Base ";c
120 k=fn l(c,h)
130 print k;" es el logaritmo de ";h;" en base "}c
140 goto 80
```

LISTADO 2

```
10 rem programa para adivinar un numero tomado al azar
20 rem por el ordenador. El numero esta comprendido en
30 rem un margen que indica el usuario
40 input "Dame el numero inferior ";n1
50 input "Dame el segundo ";n2
60 if n1>n2 then print "No vale este margen.":goto 40
70 if n2-n1<20 then print "No vale. Asi seria muy facil.":  goto
40
```

```
80 adivina=int(rnd*(n2-n1))+n1:c=0
90 print "El numero esta entre ";n1;" y ";n2
100 input "Cual es ";num:c=c+1
110 if num>adivina then n2=num:goto 90
120 if num<adivina then n1=num:goto 90
130 print "Muy bien! Lo has conseguido en ";c;" intentos."
140 end
```

LISTADO 3

```
10 rem Prueba de uniformidad de RND, con  y sin RANDOMIZE
20 dim s(5)
30 for a=1 to 100
40 c=int(rnd*5)+1
50 s(c)=s(c)+1
60 next
70 gosub 180
80 print "Diferencia entre limites: ";dif
90 for a=1 to 10
100 randomize
110 for b=1 to 10
120 c=int(rnd*5)+1
130 s(c)=s(c)+1
140 next b,a
150 gosub 180
160 print "Diferencia entre limites: ";dif
170 end
180 min=100:max=0
190 for a=1 to 5
200 if s(a)<min then min=s(a)
210 if s(a)>max then max=s(a)
220 s(a)=0
230 next
240 dif=max-min
250 return
```

EJERCICIOS

83. ¿Es posible definir funciones propias para un programa?

A) Sólo en el BASIC del PCW.
B) Sí.
C) Sí, si quedan localidades.

84. ¿Son de importancia los nombres de las variables empleadas en la definición de una función?

A) Sí, siempre y cuando nos gusten.
B) Sí. Sólo se pueden utilizar esos nombres para llamar la función.
C) No. Sólo indican el tipo de valor a emplear.

85. ¿La función RND proporciona números aleatorios con decimales?

A) No proporciona números aleatorios.
B) Proporciona números aleatorios, pero no decimales.
C) Sí.

86. ¿Qué instrucción se emplea, además de INPUT, para la lectura del teclado?

A) GAFAS ON.
B) INKEY$.
C) GET.

87. ¿Qué es TRON?

A) Sólo es el título de una película de ciencia ficción.
B) Una herramienta utilizada en la depuración de programas.
C) La instrucción para generar números aleatorios.

88. ¿Mediante la programación en BASIC, Código Máquina o cualquier otro lenguaje podemos causarle un daño irreparable a nuestro ordenador?

A) No.
B) Sí.
C) Apaga y vámonos.

APÉNDICE

LISTADO 1

```
10 REM programa para dividir polinomios
20 REM
30 DIM m$(5)
40 DIM n$(5)
50 DIM pa$(20)
60 DIM pe$(20)
70 PRINT "Programa para dividir polinomios de tipo"
80 PRINT "a0x^m+a1x^(m-1)+...+a(m-1)x+am/b0x^n+b1x^(n-1)+...
+b(n01)x+bn"
90 PRINT "Siendo m=n o m>n, y ambos numeros naturales"
100 REM Valor del mayor exponente (M)
110 PRINT "Introduzca el valor de M"
120 INPUT m$
130 jo=ASC(m$)
140 IF jo>57 OR jo<48 THEN 170
150 m=VAL(m$)
160 GOTO 180
170 PRINT "M=";m$;" Error. M solo admite valores de numeros
naturales":GOTO 110
180 va=m-INT(m)
190 IF m<0 OR va<>0 THEN PRINT "M=";m$;" Error. M ha de ser un
numero natural":GOTO 110
200 REM Valor del menor exponente (N)
210 PRINT "Introduzca el valor de N"
220 INPUT n$
230 ji=ASC(n$)
240 IF ji>57 OR ji<48 THEN 270
250 n=VAL(n$)
260 GOTO 280
270 PRINT "M=";m$;". N=";n$;" Error. N solo admite valores de
numeros naturales":GOTO 210
280 ve=n-INT(n)
290 IF n<0 OR ve<>0 THEN PRINT "M=";m$;". N=";n$;" Error. N ha
de ser un numero natural":GOTO 210
300 IF m<n THEN PRINT "M=";m$;". N=";n$;" Error. M ha de ser
mayor que n":GOTO 110
310 REM dimensionado
320 DIM a(2*(m-n)+2,m)
330 FOR u=0 TO 2*(m-n)+2
340 FOR v=0 TO m
350 a(u,v)=0
360 NEXT v,u
370 DIM b(1,m)
380 FOR x=0 TO 1
390 FOR y=0 TO m
400 b(x,y)=0
```

```
410 NEXT y,x
420 REM asignacion de valores a las constantes
430 PRINT "Introduzca de a0 a am"
440 FOR k=0 TO m
450 PRINT "A";k;
460 INPUT pa$
470 ti=ASC(pa$)
480 IF ti>57 OR ti<45 THEN 510
490 IF ti=47 THEN 510
500 GOTO 520
510 PRINT "Error. A solo admite valores numericos":GOTO 450
520 pa=VAL(pa$)
530 a(0,k)=pa
540 NEXT
550 PRINT "Introduzca de b0 a bn"
560 FOR r=0 TO n
570 PRINT "B";r;
580 INPUT pe$
590 tu=ASC(pe$)
600 IF tu>57 OR tu<45 THEN 630
610 IF tu=47 THEN 630
620 GOTO 640
630 PRINT "Error. B solo admite valores numericos":GOTO 570
640 pe=VAL(pe$)
650 b(0,r)=pe
660 NEXT
670 REM lazo de operaciones
680 FOR h=o TO m-n
690 b(1,h)=a(2*h,h)/b(0,0)
700 FOR c=h+1 TO n+h
710 a(2+h+1,c)=-b(1,h)*b(0,c-h)
720 NEXT c
730 FOR d=h+1 TO n+h
740 a(2*(h+1),d)=a(2*h,d)+a(1+2*h,d)
750 NEXT
760 IF h=m-n THEN 780
770 a(2*(h+1),h+n+1)=a(0,h+n+1)
780 NEXT h
790 REM lazo de impresion del dividendo
800 PRINT "Dividendo: ";
810 o=m
820 FOR i=0 TO m
830 IF a(0,i)<0 THEN 920
840 IF o=0 THEN 880
850 IF o=1 THEN 900
860 PRINT "+";INT(a(0,i)*100+0.5)/100;"X^";o;
870 GOTO 990
880 PRINT "+";INT(a(o,i)*100+0.5)/100
890 GOTO 990
900 PRINT "+";INT(a(0,i)*100+0.5)/100;"X";
910 GOTO 990
920 IF o=0 THEN 960
930 IF o=1 THEN 980
940 PRINT "int(a(0,i)*100+0.5)/100;"X^";o;
950 GOTO 990
960 PRINT INT(a(0,i)*100+0.5)/100
970 GOTO 990
980 PRINT INT(a(0,i)*100+0.5)/100;"X";
990 o=o-1
1000 NEXT
1010 REM lazo de impresion del divisor
1020 PRINT "Divisor :";
1030 e=n
910 GOTO 990
920 IF o=0 THEN 960
930 IF o=1 THEN 980
940 PRINT "int(a(0,i)*100+0.5)/100;"X^";o;
950 GOTO 990
960 PRINT INT(a(0,i)*100+0.5)/100
970 GOTO 990
```

```
980 PRINT INT(a(0,i)*100+0.5)/100;"X";
990 o=o-1
1000 NEXT
1010 REM lazo de impresion del divisor
1020 PRINT "Divisor :";
1030 e=n
1040 FOR f=0 TO n
1050 IF b(0,f)<0 THEN 1140
1060 IF e=0 THEN 1100
1070 IF e=1 THEN 1120
1080 PRINT "+";INT(b(0,f)*100+0.5)/100;"X^";e;
1090 GOTO 1210
1100 PRINT "+";INT(b(0,f)*100+0.5)/100
1110 GOTO 1210
1120 PRINT "+";INT(b(0,f)*100+0.5)/100;"X";
1130 GOTO 1210
1140 IF e=0 THEN 1180
1150 IF e=1 THEN 1200
1160 PRINT INT(b(0,f)*100+0.5)/100;"X^";e;
1170 GOTO 1210
1180 PRINT INT(b(0,f)+100+0.5)/100
1190 GOTO 1210
1200 PRINT INT(b(0,f)*100+0.5)/100;"X";
1210 e=e-1
1220 NEXT
1230 REM lazo de impresion del cociente
1240 PRINT "Cociente: ";
1250 g=m-n
1260 FOR h=0 TO m-n
1270 IF b(1,h)<0 THEN 1360
1280 IF g=0 THEN 1320
1290 IF g=1 THEN 1340
1300 PRINT "+";INT(b(1,h)*100+0.5)/100;"X^";g;
1310 GOTO 1430
1320 PRINT "+";INT(b(1,h)*100+0.5)/100
1330 GOTO 1430
1340 PRINT "+";INT(b(1,h)*100+0.5)/100;"X";
1350 GOTO 1430
1360 IF g=0 THEN 1400
1370 IF g=1 THEN 1420
1380 PRINT INT(b(1,h)*100+0.5)/100;"X^";g;
1390 GOTO 1430
1400 PRINT INT(b(1,h)*100+0.5)/100
1410 GOTO 1430
1420 PRINT INT(b(1,h)*100+0.5)/100;"X";
1430 g=g-1
1440 NEXT
1450 REM lazo de impresion del resto
1460 PRINT "Resto :";
1470 j=n-1
1480 FOR k=m-n+1 TO m
1490 IF a(2*(m-n)+2,k)<0 THEN 1580
1500 IF j=0 THEN 1540
1510 IF j=1 THEN 1560
1520 PRINT "+";INT(a(2*(m-n)+2,k)*100+0.5)/100;"X^";j;
1530 GOTO 1650
1540 PRINT "+";INT(a(2*(m-n)+2,k)*100+0.5)/100
1550 GOTO 1650
1560 PRINT "+";INT(a(2*(m-n)+2,k)*100+0.5)/100;"X";
1570 GOTO 1650
1580 IF j=0 THEN 1620
1590 IF j=1 THEN 1640
1600 PRINT INT(a(2*(m-n)+2,k)*100+0.5)/100;"X^";j;
1610 GOTO 1650
1620 PRINT INT(a(2*(m-n)+2,k)*100+0.5)/100
1630 GOTO 1650
1640 PRINT INT(a(2*(m-n)+2,k)*100+0.5)/100;"X";
1650 j=j-1
1660 NEXT k
1670 END
```

LISTADO 2

```
10 REM Codificador de claves a partir de cinco silabas. Estas
silabas pueden
20 REM cambiarse para dar otros criterios de codificacion. Se ha
de respetar
30 REM la longitud de estas silabas o bien modificar las lineas
530 y 540
40 PRINT "CODIFICADOR"
50 PRINT "1- Castellano - codificado"
60 PRINT "2- Codificado - castellano"
70 INPUT "Opcion ";r
80 ON r GOTO 610,660
90 END
100 REM traduce de castellano a codificado
110 REM
120 b$=""
130 FOR i=1 TO LEN(a$)
140 c=ASC(MID$(a$,i,1))
150 n1=INT(c/25)
160 n2=INT((c-25*n1)/5)
170 n3=c-25*n1-5*n2
180 n=n1
190 GOSUB 260
200 n=n2
210 GOSUB 260
220 n=n3
230 GOSUB 260
240 NEXT
250 RETURN
260 IF n=0 THEN b$=b$+"-"
270 IF n=1 THEN b$=b$+"GA"
280 IF n=2 THEN b$=b$+"BU"
290 IF n=3 THEN b$=b$+"ZO"
300 IF n=4 THEN b$=b$+"MEU"
310 RETURN
320 REM traduce codigo a castellano
330 REM
340 i=1
350 a$=""
360 GOSUB 450
370 n1=n
380 GOSUB 450
390 n2=n
400 GOSUB 450
410 n3=n
420 a$=a$+CHR$(n3+5*n2+25*n1)
430 IF i>LEN(b$) THEN RETURN
440 GOTO 360
450 IF i>LEN(b$) THEN 570
460 n=5
470 IF MID$(b$,i,1)="-" THEN n=0
480 IF MID$(b$,i,1)="G" THEN n=1
490 IF MID$(b$,i,1)="B" THEN n=2
500 IF MID$(b$,i,1)="Z" THEN n=3
510 IF MID$(b$,i,1)="M" THEN n=4
520 IF n=0 THEN i=i+1
530 IF n=1 OR n=2 OR n=3 THEN i=i+2
540 IF n=4 THEN i=i+3
550 IF n=5 THEN 570
560 RETURN
570 PRINT "Error en el texto codificado"
580 PRINT "Pulsar RETURN"
590 INPUT r$
600 RUN
610 PRINT "Introducir texto en castellano"
620 INPUT a$
630 b$=""
640 GOSUB 100
```

```
650 GOTO 700
660 PRINT "Introducir texto codificado"
670 INPUT b$
680 a$=""
690 GOSUB 320
700 REM
710 PRINT "*";b$;"*"
720 PRINT
730 PRINT "(";a$;")"
740 RUN
```

LISTADO 3

```
10 REM Agenda de telefonos y direcciones.
20 REM Permite la introduccion en memoria de hasta 100
direcciones
30 n=0
40 DIM nombre$(100),apellido$(100),direccion$(100),tele(100)
50 PRINT"Escoge":PRINT:PRINT"1=Introducir fichas nuevas":PRINT
"2=Modificar fichas":PRINT "3=Borrar fichas":PRINT "4=Ver una
ficha":PRINT "5=Ver todas las fichas"
60 INPUT o
70 ON o GOTO 90,200,410,630,760
80 GOTO 60
90 REM
100 n=n+1:PRINT "Ficha nº ";n
110 INPUT "Nombre: ";nombre$(n)
120 INPUT "Apellidos: ";apellido$(n)
130 INPUT "Direccion: ";direccion$(n)
140 INPUT "Telefono: ";tele(n)
150 PRINT:INPUT "Mas fichas(s/n)";v$
160 IF v$="s" THEN 90
170 IF v$="n" THEN 50
180 GOTO 150
190 REM                    MODIFICAR
200 REM
210 PRINT "Escoge campo:" :PRINT: PRINT "1=Nombre" :PRINT
"2=Apellidos":PRINT "3=Direccion":PRINT "4=Telefono": PRINT "5=
Volver al menu principal"
220 INPUT o:IF o>4 THEN 50
230 INPUT "Numero de ficha";z
240 ON o GOTO 250,290,330,370
250 PRINT:PRINT nombre$(z)
260 PRINT:INPUT "Nuevo nombre ";nnombre$
270 nombre$(z)=nnombre$
280 GOTO 200
290 PRINT:PRINT apellido$(z)
300 PRINT:INPUT "Nuevo apellido ";napellido$
310 apellido$(z)=napellido$
320 GOTO 200
330 PRINT:PRINT direccion$(z)
340 PRINT:INPUT "Nueva direccion ";nadireccion$
350 direccion$(z)=nadireccion$
360 GOTO 200
370 PRINT:PRINT tele(z)
380 PRINT:INPUT "Nuevo telefono ";ntele
390 tele(z)=ntele
400 GOTO 200
410 REM                    BORRAR
420 REM
430 PRINT"BORRAR FICHAS"
440 PRINT:PRINT:INPUT "Numero de ficha ";z
450 PRINT:PRINT nombre$(z)
460 PRINT:PRINT apellido$(z)
470 PRINT:PRINT direccion$(z)
480 PRINT:PRINT tele(z)
490 PRINT:PRINT"Estas seguro de que quieres borrarla?":INPUT o$
500 IF o$="s" THEN 540
510 IF o$="n" THEN PRINT"Ah, bueno":GOTO 50
```

```
520 GOTO 490
530 REM
540 nombre$(z)=nombre$(z+1): apellido$(z) = apellido$(z+1) :
direccion$(z) = direccion$(z+1): tele(z)=tele(z+1)
550 FOR a=z+1 TO n
560 nombre$(a)=nombre$(a+1)
570 apellido$(a)=apellido$(a+1)
580 direccion$(a)=direccion$(a+1)
590 tele(a)=tele(a+1)
600 NEXT
610 n=n-1
620 GOTO 50
630 REM                    VER UNA FICHA
640 REM
650 PRINT "Veamos una ficha"
660 PRINT:INPUT "Numero de ficha";z
670 IF nombre$(z)="" THEN PRINT"No existe esa ficha.":FOR a=1 TO
1500:NEXT:GOTO 640
680 PRINT:PRINT nombre$(z)
690 PRINT:PRINT apellido$(z)
700 PRINT:PRINT direccion$(z)
710 PRINT:PRINT tele(z)
720 PRINT:PRINT"Mas fichas?(s/n)";o$=INPUT$(1)
730 IF o$="s" THEN 640
740 IF o$="n" THEN 50
750 GOTO 720
760 REM                        VER TODAS
770 REM
780 f=1
790 IF nombre$(f)="" THEN GOTO 850
800  PRINT  nombre$(f):PRINT  apellido$(f):PRINT  direccion$(f):
PRINT tele(f)
810 PRINT:INPUT "Paso a la siguiente (s/n)";o$
820 IF o$="s" THEN f=f+1:GOTO 790
830 IF o$="n" THEN 50
840 GOTO 810
850 PRINT"No hay mas fichas"
860 IF INKEY$="" THEN 860
870 GOTO 50
```

TABLA DE RESPUESTAS

UNIDADES

DECENAS	0	1	2	3	4	5	6	7	8	9
0	*	C	A	A	A	A	B	B	B	B
1	B	A	C	A	A	A	B	B	B	B
2	B	B	B	A	C	C	B	A	A	B
3	B	C	C	A	A	C	B	B	B	A
4	A	A	A	B	B	A	A	A	B	C
5	A	A	A	B	A	C	B	B	B	A
6	A	B	A	C	B	C	C	A	A	B
7	B	A	B	C	B	A	A	C	B	B
8	A	C	A	B	C	C	B	B	A	*

Manejo de la tabla: En las columnas localizaremos la unidad del número de respuesta a buscar, mientras que en las filas se encuentran las decenas; el punto de intersección entre fila y columna nos proporcionará el resultado.

Por ejemplo: Para hallar la respuesta a la pregunta 72 nos situaremos en la intersección de la fila 7 y la columna 2.

INDICE GENERAL DE LA OBRA

NOTAS

Todo empezó en 1964 tras el diseño por parte de John G. Kemeny y Thomas Kurtz de un ienguaje de programación que ellos mismos llamaron *«Beginner's All-purpose Symbolic Instruction Code»*, más conocido como BASIC. El objetivo de estos dos caballeros era conseguir un lenguaje de programación «accesible» para cualquier persona con unos determinados conocimientos; se trataba en definitiva de diseñar un lenguaje fácil. Los comandos e instrucciones más comunes en él son, en gran parte, el contenido de este primer volumen sobre el lenguaje BASIC de la Gran Biblioteca Amstrad.

GRAN BIBLIOTECA AMSTRAD

450 ptas.
(incluido IVA)
Precio en Canarias, Ceuta y Melilla: 435 ptas.