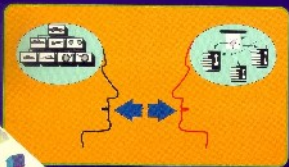


Introducción a la
**PROGRAMACIÓN ESTRUCTURADA
Y ORIENTADA A OBJETOS**
con Pascal

CONTIENE DISQUETE



E. M. CUBA LONTELL
M. E. A. GARCÍA FERNÁNDEZ
E. LÓPEZ PEREZ
M. C. LUENGO DIEZ
M. ALONSO BELLERU

A mis padres, hermanos, Toñi, Guillermo, Antonio y Paloma
J.M.C.L.

A los tres hombres que más han influido en mi vida:
mi padre, Javier y Fernando Pendás
M^a P.A.G.F.

A mis padres
B.L.P.

A mi familia
M^a C.L.D.

A nuestros alumnos como parte del intento de cumplir
con nuestra misión docente
M.A.R.

INTRODUCCION A LA PROGRAMACION ESTRUCTURADA Y ORIENTADA A OBJETOS CON PASCAL

Copyright © 1994 por Juan Manuel Cueva Lovelle, M^a del Pilar Almudena García Fuente, Benjamín López Pérez, M^a Candida Luengo Díez y Melchor Alonso Requejo.

Reservados todos los derechos. De conformidad con lo dispuesto en el art. 534-bis del Código Penal vigente, podrán ser castigados con penas de multa y privación de libertad quienes reprodujeran o plagiaran, en todo o en parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte sin la preceptiva autorización.

ISBN: 84-600-8646-1

Depósito legal: AS/2.897-93

Impreso en Gráficas Oviedo, S.A.

C/ Almacenes Industriales, 10. 33012-OVIEDO

DEPARTAMENTO DE MATEMATICAS
CUADERNO DIDACTICO N° 69

**INTRODUCCION A LA
PROGRAMACION ESTRUCTURADA
Y ORIENTADA A OBJETOS
CON PASCAL**

Juan Manuel Cueva Lovelle

Catedrático de E.U. de Lenguajes y Sistemas
Informáticos de la Universidad de Oviedo

M^a del Pilar Almudena García Fuente

Profesora Titular de E.U. de Lenguajes y Sistemas
Informáticos de la Universidad de Oviedo

Benjamín López Pérez

Profesor Asociado de Lenguajes y Sistemas
Informáticos de la Universidad de Oviedo

M^a Cándida Luengo Díez

Profesora Asociada de Lenguajes y Sistemas
Informáticos Universidad de Oviedo

Melchor Alonso Requejo

Profesor Titular de E.U. de Lenguajes y Sistemas
Informáticos de la Universidad de Oviedo

PREFACIO

Pascal siempre ha jugado un papel clave dentro de la estrategia de los productos de Borland, y este año celebrando el registro de dos millones de usuarios de Pascal, Borland ha desarrollado su Hoja de Cálculo Quattro Pro 5.0 para DOS con la versión 7.0 de Pascal, resaltando de esta forma la viabilidad del uso de Turbo Pascal para el desarrollo de aplicaciones comerciales.

Nos preguntaremos el porqué muchos programadores eligen Pascal. La respuesta es simple, se fomenta la productividad a medida que se aventajan de las técnicas de programación que ya posee.

- El lenguaje Pascal es más potente y eficiente que BASIC, aún menos complejo que C y C++, por lo que es más fácil de aprender.*
- La seguridad de tipos de datos en el Lenguaje Pascal significa que la mayoría de los errores de programación pueden ser detectados antes de que el programa se esté ejecutando, facilitando así la depuración y permitiendo encontrar problemas en las aplicaciones antes de que sea utilizado por el usuario final.*
- Pascal es el lenguaje que más se utiliza para el aprendizaje de las técnicas de programación por la mayoría de los centros universitarios y académicos.*
- La integración de la sintaxis necesaria para la Programación Orientada a Objetos (OOP) en Pascal, no podría ser más simple, sólo cuatro palabras reservadas nuevas permiten que los programadores puedan migrar a OOP sin tener que aprender un nuevo lenguaje.*

"Introducción a la Programación estructurada y Orientada a Objetos con Pascal" es la obra que refleja la experiencia de la Universidad de Oviedo en la enseñanza del lenguaje Pascal, y la profesionalidad de los autores desarrollando en este entorno de programación.

Didácticamente el lector comenzará adquiriendo los conocimientos del desarrollo en Pascal mediante Programación Estructurada, y finalmente, asimilará la nueva tecnología de desarrollo, la Programación Orientada a Objetos utilizando Turbo Visión, las librerías de clases DOS, que facilitan la realización del interfaz de usuario. Sus aplicaciones heredarán un interfaz de usuario multi-ventana con soporte de ratón, con ventanas, menús, diálogos, editores, validación de datos, barras de deslizamiento, etc.

Estamos orgullosos del trabajo que han realizado los autores explicando cómo se debe programar en Pascal en esta obra, e igualmente por el trabajo excepcional que el Area de Lenguajes y Sistemas Informáticos del Departamento de Matemáticas de la Universidad de Oviedo, ha realizado en la traducción de los manuales de nuestro producto Turbo Pascal 7.0 para DOS.

Deseamos mediante estas líneas expresar públicamente nuestro agradecimiento.

Pedro Robledo
Product Manager de Lenguajes
Borland España

PROLOGO

El presente libro está pensado para dar soporte a un primer curso universitario de programación estructurada (PE) y programación orientada a objetos (POO) de nueve meses de duración y cinco horas semanales de clase. Su objetivo es eminentemente didáctico, incluyéndose más de 400 ejemplos y ejercicios resueltos. Además se incorporan ejercicios propuestos al final de cada capítulo.

Se ha utilizado el lenguaje Pascal, como vehículo de implementación de los conceptos teóricos impartidos. Se ha elegido por su claridad frente a otros lenguajes como C o C++ (que encajan mejor en un segundo curso de programación). Para la implementación de los tipos abstractos de datos se han utilizado las units que incorpora Turbo Pascal, como extensión al Pascal estándar.

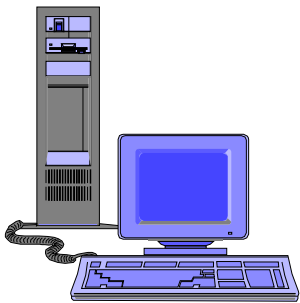
El libro parte de los conceptos más elementales de la Informática (capítulo 1) para ir introduciendo unas ideas generales de construcción de programas (capítulo 2), y paso a paso como si fuera un juego se van incorporando los conceptos de la programación estructurada.

Los capítulos 3 al 12 están dedicados a la programación de las estructuras de control y de datos tradicionales. Cada capítulo se completa con una serie de ejercicios resueltos y propuestos, así como notas bibliográficas específicas para cada tema. Su objetivo es inculcar en el lector una metodología de diseño estructurada, a la vez que se estudia en paralelo el lenguaje Pascal estándar, haciendo especial énfasis en las diferencias y ampliaciones del Turbo Pascal, que es actualmente, sin lugar a dudas, el entorno de desarrollo de programas en Pascal más ampliamente difundido y especialmente indicado para el aprendizaje.

En el capítulo 13 se introducen los conceptos fundamentales de la programación orientada a objetos, que se aplicarán en los capítulos 14 (Turbo Vision) y 15 (Object Windows). Además en los capítulos 14 y 15 se introduce la programación dirigida por eventos, y sus aplicaciones prácticas en combinación con la POO a los interfaces de usuario en entornos DOS y Windows. El último capítulo hace un repaso general a las técnicas de la PE y de la POO comparándolas entre sí.

Quisiéramos expresar nuestro agradecimiento a todos aquellos compañeros y alumnos que nos han hecho llegar sus sugerencias para la elaboración de este libro.

LOS AUTORES, Enero de 1994



CAPITULO 1

INTRODUCCION A LA INFORMATICA

CONTENIDOS

- 1.1 Introducción
- 1.2 Representación de la información
- 1.3 Hardware o soporte físico
- 1.4 Software o soporte lógico
- 1.5 Cuestiones
- 1.6 Ampliaciones y notas bibliográficas

1.1 INTRODUCCION

El objetivo de este capítulo es definir los conceptos y términos necesarios para introducir al lector en el mundo de la Informática. Si el lector ya los conoce puede pasar directamente al siguiente capítulo.

Informática es una palabra de origen francés formada por la contracción de los vocablos INFORmación y autoMATICA. La Real Academia Española de la Lengua define la **Informática** como el *conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores*. En los países latinoamericanos se emplea el vocablo *Computación*, influencia de la palabra sajona *Computation*.

REPRESENTACION DE LA INFORMACION

La Informática trata de la adquisición, almacenamiento, representación, tratamiento y transmisión de la información. Estas operaciones se realizan automáticamente utilizando máquinas denominadas *ordenadores* o *computadoras*.

El término *información* en Informática tiene un significado amplio, que se puede definir como el conjunto de símbolos que representan hechos, objetos o ideas.

El **ordenador** o **computadora** *es una máquina capaz de aceptar unos datos de entrada, efectuar con ellos operaciones lógicas y aritméticas y proporcionar la información resultante a través de un medio de salida; todo ello sin intervención de un operador humano y bajo el control de un programa de instrucciones previamente almacenado en el ordenador.* Se entiende por operaciones lógicas las que realizan operaciones de comparación, selección, copia de símbolos, ya sean numéricos, alfanuméricos, gráficos, etc... Un ordenador puede considerarse como un sistema cuyas salidas o resultados dependen de sus entradas constituidas por *datos* e *instrucciones*.

Los **datos** *son conjuntos de símbolos utilizados para expresar o representar un valor numérico, un hecho, un objeto o una idea; en la forma adecuada para ser objeto de tratamiento por el ordenador.* Los datos pueden ser por ejemplo el peso en kg. de una persona, su nombre y apellidos, su fotografía, y un conjunto de frases acerca sus cualidades. La información es el resultado de la transformación o proceso de estos datos.

Las **instrucciones** son las indicaciones que se hacen al ordenador para que realice las tareas encomendadas. Se define **algoritmo** como el *conjunto de instrucciones dadas a un ordenador para resolver un determinado problema en una cantidad finita de tiempo.* Un **programa** *es un conjunto ordenado de instrucciones que se dan al ordenador indicándosele las operaciones o tareas que se desea que realice.*

La **programación de ordenadores** es la parte de la Informática dedicada al estudio de las distintas metodologías, algoritmos y lenguajes para construir programas.

La **disciplina de Informática** se define como *el estudio sistemático del procesamiento automático de algoritmos que describen y transforman la información: su teoría, análisis, diseño, eficacia, implantación, y aplicación.*

1.2 REPRESENTACION DE LA INFORMACION

En los ordenadores la información se almacena, procesa y transfiere entre las distintas unidades en forma binaria, es decir en base 2. Esto quiere decir que la unidad básica de información es el **bit**. Un bit es la cantidad de información que puede almacenarse en un dispositivo binario (por ejemplo un interruptor: abierto o cerrado; un toroide de material magnético: magnetizado o no; ...). Para representar los dos posibles valores de los mensajes, se utilizan los símbolos "0" y "1". De hecho, la palabra "bit" es una contracción de "*Binary digiT*".

Si bien la unidad básica es el bit, los bits no se manejan individualmente, sino por grupos de un tamaño fijo. Así a 8 bits se le da el nombre de octeto o en inglés **byte**.

Habitualmente se utilizan los múltiplos del byte:

- Kilobyte** = 1024 bytes, es decir 2^{10} bytes.
- Megabyte** = 1.048.576 bytes, es decir 2^{20} bytes.
- Gigabyte** = 1.073.741.824 bytes, es decir 2^{30} bytes.
- Terabyte** = 2^{40} bytes.

REPRESENTACION DE CARACTERES

Los ordenadores sólo trabajan con números binarios, sin embargo la información escrita por los seres humanos suele expresarse con un alfabeto o conjunto de símbolos denominados **caracteres**. Estos caracteres pueden agruparse en cinco categorías:

Caracteres alfabéticos: Son las letras mayúsculas y minúsculas del alfabeto inglés: A, B, C, ..., X, Y, Z, a, b, ..., x, y, z

Caracteres numéricos: Están constituidos por los diez dígitos decimales: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Caracteres especiales: Son signos de puntuación, comparación, admiración, interrogación y otros: \, |, @, #, [, {,], }, etc...

Caracteres de control: Representan órdenes de control como pitido, fin de página, fin de línea, etc...

Caracteres expandidos: Dado que las distintas lenguas tienen símbolos que no existen en el idioma inglés, se han de añadir éstos, así como otros caracteres gráficos. Por ejemplo la letras ñ, Ñ, las vocales acentuadas, letras griegas, etc...

Para representar caracteres se utiliza un código, es decir a cada letra o símbolo se le asigna un valor en binario.

Los códigos más utilizados son el código **ASCII** (*American Standard Code for Information Interchange*) y el código **EBCDIC** (*Extended Binary Coded Decimal Information Code*). Ambos códigos son de 8 bits o un byte, es decir cada carácter se representa por un byte, por lo tanto se podrán representar 256 caracteres, aunque el código ASCII sólo tiene normalizados los 127 primeros caracteres (7 bits).

El código ASCII es el más extendido y es el estándar internacional. En el anexo I se muestra la tabla ASCII; puede observarse que desde el código 0 al 31 son caracteres de control; del 32 al 47 son símbolos como !, ", #, \$, ..., /; del símbolo 48 al 57 son los dígitos; del 58 al 63 son otros símbolos como :, ;, <, =, >, ?, y @; del 64 al 90 son las letras mayúsculas; del 91 al 96 son más símbolos como [, \,], ..., ' ; del 97 al 122 son letras minúsculas; del 123 al 126 son otros símbolos; el 127 el carácter retroceso usado para borrar; y del código 128 al 255 es el código ASCII extendido que es diferente para cada fabricante, siendo el más usado el utilizado por los ordenadores IBM PC y compatibles.

REPRESENTACION DE LA INFORMACION

REPRESENTACION DE NUMEROS ENTEROS

Los datos de tipo entero también denominados de **punto fijo** o de coma fija tienen dos alternativas de representación: con signo y sin signo.

• Enteros sin signo

Los enteros sin signo se representan directamente en base dos.

Por ejemplo con una longitud de palabra de 16 bits se pueden representar enteros sin signo entre 0 y 65535 (**entero corto**), que tiene 4 dígitos significativos.

Base 2	Base 10
0000000000000000	0
1111111111111111	65535

Tabla 1.1 Rango de enteros sin signo de 2 bytes.

Si la longitud de palabra es de 32 bits se pueden representar enteros sin signo entre 0 y 4.294.967.295 (**entero largo**), que tiene 9 dígitos significativos.

De lo expuesto anteriormente se deduce que existe un **rango de representación** de los números enteros sin signo.

• Enteros con signo

Si los enteros tienen signo el bit situado más a la izquierda de la palabra representa al signo. Este bit es 0 si el número es positivo o 1 si es negativo. Si la longitud de palabra es de 16 bits, sólo se pueden utilizar 15 bits, siendo su rango entre -32768 y 32767 (**entero corto**). Si la longitud de palabra es de 32 bits, sólo se pueden utilizar 31 bits, siendo su rango entre -2.147.483.648 y 2.147.483.647 (**entero largo**).

REPRESENTACION DE NUMEROS REALES

Los datos de tipo real también denominados de **punto flotante**, son un subconjunto del conjunto de los números reales R definido en Matemáticas. Los datos de tipo real tienen un número finito de dígitos decimales, mientras que por ejemplo los números reales irracionales de Matemáticas tienen infinitos dígitos decimales. La representación de un número real es de la forma:

$$\textit{mantisa} \times 2^{\textit{exponente}}$$

Por ejemplo un número *real* de 6 bytes (48 bits) su representación interna es:

- 1 bit para el signo de la mantisa
- 39 bits para la mantisa
- 8 bits para el exponente

Con la codificación anterior se puede representar el siguiente rango de números, con 11-12 dígitos significativos:

+1.7 E+38	Máximo nº real positivo
+2.9 E-39	Mínimo nº real positivo
-1.7 E-38	Máximo nº real negativo
-2.9 E+39	Mínimo nº real negativo

También se representan reales con 4 bytes (precisión simple, *single*), 8 (doble precisión, *double*) y 10 bytes (precisión extendida, *extended*). En la tabla 1.2 se representan como ejemplo los tipos real que soporta Turbo Pascal 7 con su rango, dígitos significativos, y tamaño que ocupan en bytes.

Tipo	Rango de los reales positivos	Dígitos significativos	Tamaño en bytes
Real	$2,9 \times 10^{-39} .. 1,7 \times 10^{38}$	11-12	6
Single	$1,5 \times 10^{-45} .. 3,4 \times 10^{38}$	7-8	4
Double	$5,0 \times 10^{-324} .. 1,7 \times 10^{308}$	15-16	8
Extended	$3,4 \times 10^{-4932} .. 1,1 \times 10^{4932}$	19-20	10

Tabla 1.2 Tipos real en Turbo Pascal 7

Con la representación de números reales se pueden obtener las siguientes conclusiones:

- a) Existe un **límite de magnitud** de los números reales, que depende del tamaño de palabra utilizada.
- b) Existe un **límite de precisión** debido a que no se pueden almacenar todas las cifras decimales que se deseen.
- c) De las dos conclusiones anteriores se deduce que existe un **error representacional**, debido a que muchos números no se pueden representar exactamente en la memoria del ordenador por tres motivos:
 - ser irracionales
 - ser racionales que no tienen representación binaria exacta
 - tener excesivos números significativos

HARDWARE O SOPORTE FISICO

REPRESENTACION DE DATOS DE TIPO LOGICO O BOOLEANO

La representación de los datos de tipo lógico o booleano se realiza por medio de un 0 (falso en inglés "*false*") y un 1 (cierto en inglés "*true*"). Para la representación interna de este tipo de datos con un sólo bit sería suficiente, pero dado que el proceso de información del ordenador se hace por palabras (*word*), que son bytes o múltiplos de bytes, la representación del tipo lógico es muy variada según el tipo de máquina y compilador. En unos tipos lógicos se ponen a cero o a uno todos los bits de un byte, o de una palabra (*word*, 2 bytes), o de doble palabra (4 bytes).

REPRESENTACION DE TIPOS DE DATOS DE TIPO PUNTERO

Un puntero es un tipo de datos que representa una dirección o posición de memoria del ordenador. Suelen utilizarse los punteros para el manejo de estructuras dinámicas de datos. El manejo de punteros se introduce en el capítulo 12. En Turbo Pascal 7, un puntero se almacena en dos palabras (4 bytes), en una palabra se almacena el desplazamiento (palabra baja) y en otra el segmento (palabra alta).

REPRESENTACION DE TIPOS ESTRUCTURADOS

Los tipos estructurados son estructuras de datos de otros tipos simples o estructurados. Su representación interna es la misma que la de sus componentes más simples.

Los tipos estructurados del lenguaje Pascal son: tipos array, registros, conjuntos y ficheros. Turbo Pascal añade los tipos estructurados string, y cadenas acabadas en carácter nulo. Un estudio más detallado de cada uno de los tipos estructurados se trata en los capítulos 8, 9, 10, 11 y 12.

REPRESENTACION DE TIPOS OBJETO

Los tipos objeto son la implementación que realiza Turbo Pascal de los tipos abstractos de datos o *clases*. Los tipos objeto o tipos *object* incluyen tanto datos como programas de manejo de dichos datos. Los tipos objeto se introducen en el capítulo 13.

1.3 HARDWARE O SOPORTE FISICO

El equipo físico que compone el sistema de un ordenador se le denomina con la palabra inglesa "*hardware*", cuya traducción directa significa *ferretería*, sin embargo en castellano se denomina soporte físico. Es decir el **hardware o soporte físico** es el *conjunto de dispositivos electrónicos y electromecánicos, circuitos, cables, tarjetas, armarios, periféricos de todo tipo y otros elementos físicos que componen el sistema del ordenador*. En la figura 1.1 se muestra un sistema de un ordenador y sus periféricos.

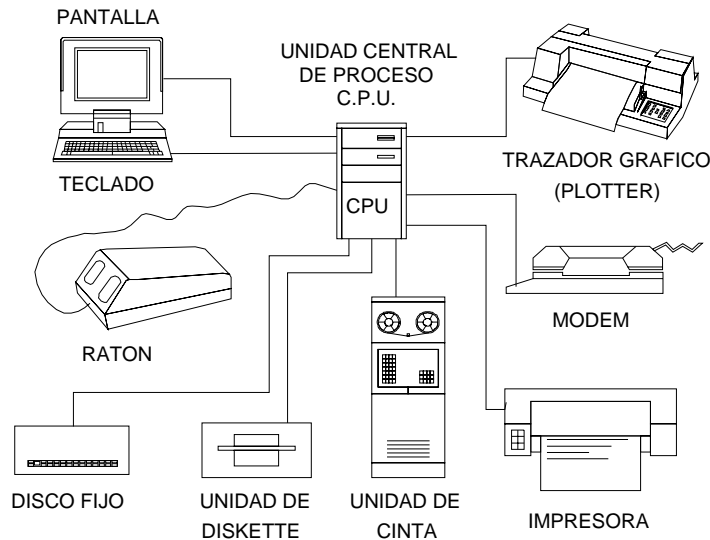


Figura 1.1 El ordenador y sus periféricos

COMPONENTES DE UN ORDENADOR

Los elementos básicos de un ordenador, son los siguientes:

- La *unidad central de proceso* (CPU)
- La *memoria principal*
- Las *unidades de entrada/salida* (E/S, en inglés I/O)

En general, el trabajo de un ordenador consiste en realizar una serie de operaciones con unos datos y presentar posteriormente los resultados. Para lograr esto, el ordenador debe de ejecutar secuencialmente un conjunto de instrucciones, que constituyen el programa. Este programa se introduce a través de las unidades de entrada y queda almacenado en la memoria. La ejecución del programa consiste en la lectura sucesiva de las instrucciones y su realización por parte de la unidad central de proceso.

Así pues, las unidades de E/S se encargan de los intercambios de información del ordenador con el exterior, mientras que la unidad central de proceso ejecuta el programa previamente almacenado en memoria. Dentro de la CPU pueden diferenciarse dos partes:

- La **unidad de control**, que va extrayendo secuencialmente de la memoria las instrucciones, las analiza y produce las órdenes necesarias para su ejecución dentro de la otra unidad, la unidad aritmético-lógica.

HARDWARE O SOPORTE FISICO

- La **unidad aritmético-lógica (ALU)**, realiza las operaciones aritméticas y lógicas. Sólo ejecuta cálculos de sumas, restas, multiplicaciones, divisiones, y operaciones lógicas de comparación (mayor, menor, igual que etc...)

La memoria principal, además de almacenar las instrucciones que constituyen el programa, puede almacenar también datos y resultados. La memoria principal de un ordenador se puede imaginar como un conjunto de múltiples celdas. A cada una de ellas se le asigna un número, que se denomina **dirección** de esa **posición de memoria**.

Las unidades de entrada/salida permiten comunicar al ordenador con el exterior, los dispositivos más comunes que se conectan por estas unidades son teclados, pantallas, unidades de almacenamiento, impresoras, etc...

Cada uno de los componentes anteriores del ordenador, están interconectados entre sí, por medio de líneas de control y de datos llamadas **bus**.

TIPOS DE ORDENADORES

Los ordenadores se clasifican habitualmente por su tamaño, capacidad y potencia de cálculo en los siguientes grupos:

- Nanocomputadoras (*nanos*), ordenadores familiares o domésticos
- Microordenadores (*micros*), o ordenadores personales
- Miniordenadores (*minis*) o ordenadores medianos
- Estaciones de trabajo (*workstations*)
- Macroordenadores (*mainframes*)
- Superordenadores

La clasificación anterior tiene límites muy difusos y variables en el tiempo, debido a que los avances de la microelectrónica posibilitan una mayor integración de los circuitos, en conjunción con una potencia de cálculo superior. Por ejemplo un ordenador clasificado hace diez años como macroordenador puede ser hoy en día más lento que un microordenador.

• **Nanocomputadoras (nanos)**

Las nanocomputadoras son ordenadores que se introdujeron en el ámbito familiar para jugar, realizar cálculos domésticos, proceso de textos, etc.... Ejemplos de este tipo de ordenadores son los ZX-81, Spectrum, Commodore-64, MSX, Dragón, AMSTRAD-PCW, etc... Normalmente están basados en los microprocesadores Z-80 de Zilog, Motorola 6809 y MOS 6502. Actualmente prácticamente han desaparecido.

• **Microordenadores (micros)**

Se utiliza el término microordenador para designar a los ordenadores que tienen toda su CPU en un sólo microprocesador o pastilla (en inglés "*chip*"). Un **microprocesador** es un circuito integrado o **chip** que contiene la CPU y de 18 a 40 patillas, y con tamaño algo menor que una caja de cerillas.

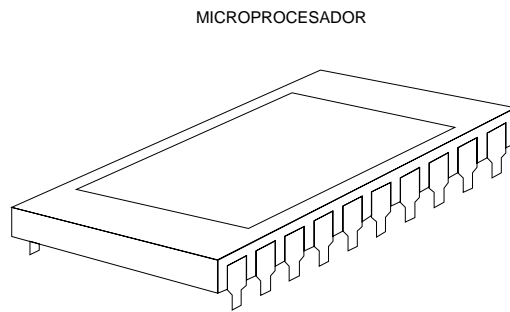


Figura 1.2 Microprocesador

La popularización de los microordenadores comenzó en los años 70 con el ordenador Apple II, sin embargo su impacto comercial comienza cuando IBM lanza en agosto de 1981 su ordenador personal PC (*Personal Computer*) basado en el microprocesador 8088 de Intel. A este lanzamiento le siguieron otros muchos fabricantes: Compaq, ALR, Olivetti, Tandom, Hewlett-Packard, etc... Desde entonces la evolución ha sido continua tanto en la aparición de nuevos microprocesadores y modelos de microordenadores cada vez más potentes y veloces. Hoy en día prácticamente todos los fabricantes de ordenadores comercializan ordenadores personales. Se puede decir en comparación con la industria del automóvil, que los ordenadores personales vienen a ser los utilitarios de la Informática.

Actualmente la mayor parte de los microordenadores están basados en dos grandes familias de microprocesadores: Intel y Motorola. La familia Intel (8088, 8086, 80186, 80286, 80386, 80486 y Pentium) es la base de todos los ordenadores IBM y compatibles (PC, XT, AT, PS/2, 386, 486, Pentium). La familia Motorola (68000, 68010, 68020, 68030, 68040) es la base de los ordenadores Appel Macintosh, NEXT, AMIGA, ATARI, y de algunos miniordenadores. También existen otros microprocesadores de otras marcas o alianzas de marcas como por ejemplo PowerPC.

El **coprocesador matemático** o procesador numérico es otro microprocesador adicional que se puede incorporar a los microordenadores. El objetivo de este microprocesador es descargar de las operaciones aritméticas al procesador principal. Los coprocesadores matemáticos realizan directamente las operaciones con números reales y con funciones trascendentes (Seno, Coseno, Tangente, potencias del número e, etc...). La presencia del coprocesador matemático en un ordenador es importante siempre que se vayan a realizar cálculos matemáticos (manejo de reales),

HARDWARE O SOPORTE FISICO

gráficos (son necesarias funciones trigonométricas para realizar las operaciones), y autoedición (se necesitan para la generación de fuentes de letra y para la mezcla de imágenes y textos). Los coprocesadores matemáticos de la familia Intel son el 8087, 80287, y el 80387; los de la familia Motorola son el 68881 y el 68882. Los microprocesadores 80486 y Pentium llevan el microprocesador incorporado en el mismo chip.

Otro aspecto a tener en cuenta es la **frecuencia del reloj** con la que trabaja el microprocesador, y que marca el ritmo con el cual se ejecutan todas las tareas. La frecuencia se mide en **megahercios** (MHz, o millón de ciclos por segundo). Por ejemplo el 8088 funciona a 4,77 Mhz, el resto de los microprocesadores de la familia Intel tienen frecuencias de reloj de 6, 8, 10, 12, 16, 20, 25, 33, 50 y 66 MHz. La familia Motorola alcanza en este momento los 50 MHz. Aunque hay prototipos de una nueva generación que pasa de los 100 Mhz.

Los microordenadores tienen un conjunto de **ranuras o slots de expansión** para colocar tarjetas de expansión o controladoras de periféricos. En el caso de los ordenadores PC existen actualmente varios tipos de estándares en función del bus y de la norma de diversos fabricantes:

- Slots de expansión tipo XT de 8 bits
- Slots de expansión tipo AT de 16 bits, también denominados ISA.
- Slots de expansión tipo Micro Channel de 32 bits
- Slots de expansión tipo EISA de 32 bits
- Slots de expansión tipo Local Bus de 64 bits.

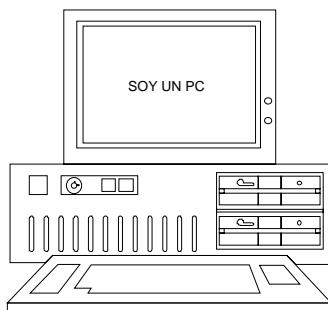
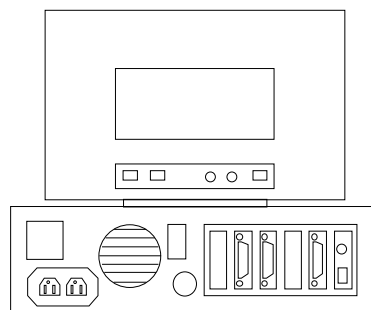


Figura 1.3 Ordenador personal



PARTE POSTERIOR DE UN PC

Figura 1.4 Conexiones de un ordenador personal

Los microordenadores utilizan distintos tipos de memoria:

- La **memoria de acceso al azar o aleatorio RAM** (*Random Acces Memory*). La memoria RAM es de lectura y escritura, es decir que pueden escribirse en ella datos, además de

poderse leer. Normalmente la memoria principal suele ser de este tipo. En los ordenadores tipo PC, por encima de las 640 Kbytes existen dos especificaciones: la especificación de memoria extendida (XMS) y la especificación de memoria expandida (EMS).

- La **especificación de memoria extendida**, en inglés "*eXtended Memory Specification*" (XMS) define una interfaz software que permite al sistema operativo MS-DOS gestionar la memoria independientemente de hardware. Tan sólo es permitida en ordenadores con microprocesadores 80286 o superior.
 - La **especificación de memoria expandida** Lotus/Intel/Microsoft (LIM), en inglés "*Lotus/Intel/Microsoft Expanded Memory Specification*" (EMS) define una interfaz hardware/software que permite al sistema operativo MS-DOS acceder hasta a 32 Mbytes de memoria, aunque el procesador principal sea un 8086 o 8088.
- La **memoria de sólo lectura ROM** (*Read Only Memory*). Es una memoria no destructible, es decir no se puede escribir sobre ella, y aún en el caso de interrupción de corriente conserva intactas las informaciones que contiene. Suele almacenar la configuración del sistema, el programa de arranque, y en algunas marcas como IBM un intérprete de BASIC. Algunas de las informaciones y servicios contenidas en la ROM son empleados para realizar operaciones de entrada y salida, por eso una parte de la ROM suele denominársele ROM-BIOS o simplemente BIOS (*Basic Input Output Services*).
 - La **memoria programable de sólo lectura PROM** (*Programmable Read-Only Memory*) son memorias de sólo lectura pero que pueden ser programables por un método especial, pero sólo una vez.
 - La **memoria programable de sólo lectura y que se puede borrar** para volver a programarse EPROM (*Erasable Programmable Read-Only Memory*).
 - La **memoria caché** (*memory cache*). Es un área de memoria intermedia. Si la memoria caché se refiere al microprocesador almacena las instrucciones, liberando así más rápidamente los dispositivos o tareas que realiza. Los microprocesadores Intel 486 tienen una caché interna de 8 Kbytes, pero la mayor parte de las placas base¹ vienen con 128, 256 y 512 Kbytes de memoria caché. También puede haber memorias caché entre el microprocesador y distintos periféricos.

¹ La placa base es la placa donde está instalado el microprocesador y otros circuitos electrónicos para su comunicación con los periféricos.

HARDWARE O SOPORTE FISICO

Los microordenadores se utilizan actualmente para todo tipo de aplicaciones tanto de gestión como técnicas, reemplazando en muchos casos a ordenadores superiores como son las estaciones de trabajo y los miniordenadores. Habitualmente son utilizados como puestos de trabajo en redes, o como terminales de ordenadores de más potencia.

• Miniordenadores (minis)

Los miniordenadores son ordenadores dedicados al soporte de varios usuarios simultáneamente, habitualmente tienen una CPU con varios microprocesadores, estando cada uno de los cuales especializado en una tarea, por ejemplo entrada y salida.

La velocidad de cálculo en estas máquinas suele medirse en **MIPS** (*millones de instrucciones por segundo*).

Las principales aplicaciones de este tipo de ordenadores suele ser la gestión de empresas grandes y medianas, dada su capacidad de tratamiento de grandes volúmenes de datos a gran velocidad. Otra aplicación es la de actuar como servidores de redes de microordenadores.

• Estaciones de trabajo (workstations)

Este tipo de ordenadores suele estar orientado a un usuario con grandes capacidades gráficas y potencia matemática de cálculo. Sus principales campos de aplicación son el diseño asistido por ordenador, cartografía, y aplicaciones científicas.

Actualmente existe una banda difusa entre los microordenadores de gama alta y las estaciones de trabajo de gama baja, siendo la principal característica de diferencia entre ambos tipos la existencia de procesadores especializados por ejemplo para el manejo de gráficos, cálculos matemáticos, etc... De hecho la existencia de este tipo de ordenadores proviene de miniordenadores especializados en gráficos o de microordenadores muy ampliados.

Los microprocesadores empleados en las estaciones de trabajo son de la familia Intel Pentium; de la familia Motorola el 68030 y 68040; también pueden ser de tecnología RISC, *Reduced Instruction Set Computer*, es decir ordenadores con un conjunto reducido de instrucciones a nivel de máquina. La arquitectura RISC nace en contraposición de la arquitectura CISC, *Complex Instruction Set Computer*, es decir ordenadores con un conjunto complejo de instrucciones. La arquitectura CISC utiliza el 80% de sus ciclos en la ejecución de un 20% de las instrucciones disponibles (que son las básicas). La arquitectura RISC es un hardware especializado en estas instrucciones simples que ocupan el 80% de los ciclos. Los microprocesadores de las familias 80X86 de Intel y 68000 de Motorola son arquitecturas CISC, mientras que con arquitecturas RISC se encuentran el 88000 de Motorola, i860 de Intel, ALPHA de DEC, MIPS Computer Systems (que equipa a Silicon Graphics, WANG, y otros), SPARC de Sun, POWER en la familia IBM RS/6000, CLIPPER en Intergraph y otros de diversos fabricantes.

La velocidad de cálculo se suele medir en MIPS (millones de instrucciones por segundo) o en **MFLOPS** (*millones de operaciones en punto flotante*).

• **Macroordenadores (mainframes)**

La principal característica de este tipo de ordenadores es que pueden ser utilizados simultáneamente por gran cantidad de usuarios, normalmente soportan entre 64 y 3000 usuarios.

La aplicación principal de este tipo de ordenadores es la gestión de grandes bases de datos, usadas por multitud de usuarios directamente y por control remoto. Este tipo de ordenadores son utilizados para realizar la gestión de grandes empresas.

• **Superordenadores**

Los superordenadores son la gama más alta, y su principal característica es la rapidez y precisión de sus cálculos. Su aplicación más importante es la realización de cálculos científicos. La CPU de los superordenadores está compuesta por varios procesadores que trabajan en paralelo.

El principal campo de aplicación de este tipo de ordenadores es en aplicaciones científicas, simulación, y cálculo. Son empleados por empresas que utilizan tecnología punta y por Universidades.

LOS PERIFERICOS

Los periféricos son los dispositivos que se conectan al ordenador para realizar las entradas y salidas de información. A continuación se muestra algunos de los más habituales.

• **Teclado**

El teclado es el dispositivo más utilizado en los ordenadores, es similar al teclado de un máquina de escribir eléctrica, al que se añaden unas teclas adicionales. En un teclado se pueden distinguir (fig. 1.5):

- *Teclado principal*: Agrupa las teclas típicas de una máquina de escribir, letras, números, signos de puntuación, etc...
- *Teclado numérico*: Contiene los diez dígitos y algunas teclas para operaciones aritméticas.
- *Teclas de funciones programables*: Contiene diversas teclas que se pueden programar por el usuario: F1, F2, ..., F12.
- *Teclas de control del cursor o de gestión de imágenes*: Permiten mover el cursor por la pantalla, suelen ser flechas hacia arriba, hacia abajo y hacia ambos lados, Inicio, Fin, Retrocede página y Avanza página.

HARDWARE O SOPORTE FISICO

- *Teclas de funciones locales*: Controlan funciones como pausa, imprimir pantalla, etc...

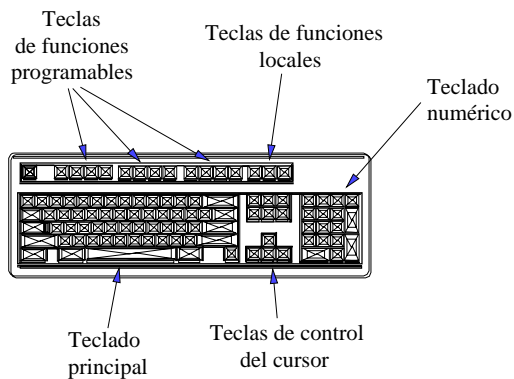


Figura 1.5 Teclado

Cuando se elige un determinado teclado es importante comprobar si tiene las teclas utilizadas en español ñ, Ñ, acentos, etc... o en otros idiomas que le interesen al usuario, por ejemplo Ç, para catalán o francés.

• Pantalla

La pantalla de un ordenador es un tubo de rayos catódicos, que forma la imagen al incidir un haz de electrones sobre la superficie interna de la pantalla que está recubierta de un material fosforescente.

En un principio se pueden clasificar las pantallas en: *pantallas de caracteres* y *pantallas gráficas*. Las pantallas alfanuméricas o de caracteres tan sólo permiten escribir letras, dígitos y signos; es lo que se llama modo texto. Las pantallas en modo texto están formadas por celdas dentro de cada una de ellas tan sólo se escribe un carácter. Las pantallas gráficas permiten la representación de gráficos, es lo que se denomina también modo gráfico. La mayor parte de las pantallas permiten ambos modos: texto y gráficos.

La imagen para ser visualizada durante un determinado tiempo debe ser repetida o "refrescada" periódicamente (al menos 25 veces por segundo).

La imagen de una pantalla gráfica no es continua sino que se forma por multitud de puntos de imagen o **pixels** (abreviatura en inglés de "*picture elements*"). Se denomina **resolución gráfica** al número de pixels que tiene, suele expresarse en forma de producto como el número de pixels en dirección horizontal por el número de pixels en dirección vertical. En la figura 1.6, cada uno de los cuadraditos sería un pixel, y la representación de una curva continua como la circunferencia se aproxima con los pixels.

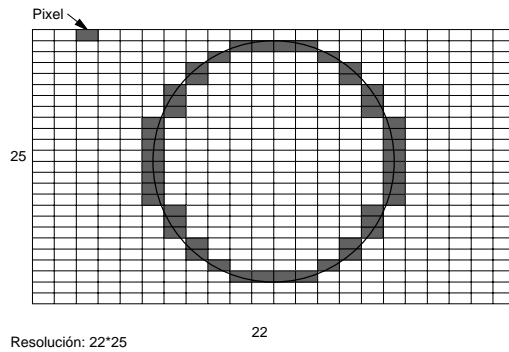


Figura 1.6 Resolución y píxeles

Las pantallas gráficas son capaces de mostrar un conjunto finito de colores, conocido con el nombre de **paleta**. El número de *bits* almacenados por *pixel* define el número de colores que es capaz de manejar la pantalla gráfica (véase tabla 1.3 y figura 1.7).

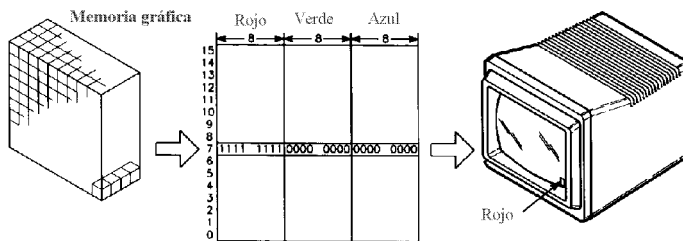


Figura 1.7 Píxeles y colores

El número de bits por píxel se puede distribuir en función del total de la memoria de la tarjeta gráfica (o memoria de video) entre resolución y número de colores.

A continuación se indica una lista con los tipos de pantallas y tarjetas controladoras.

Sistemas monocromáticos

- *Adaptador MDA (Monochrome Display Adapter)*
Sólo admite modo texto: 25 líneas y 80 columnas, es decir 25×80 celdas con 9×14 puntos.
- *Adaptador HGC (Hercules Graphics Card)*
Modo texto: igual que MDA.
Modo Gráfico: resolución de 720×348 píxeles

HARDWARE O SOPORTE FISICO

Nº de bits por pixel (n)	Nº de colores simultáneos (2 ⁿ)
1	2
2	4
4	16
8	256
12	4.096
16	65.536
20	1.048.576
24	16.777.216
32	4.294.967.296

Tabla 1.3 Relación entre el nº de colores y el nº de *bits* por *pixel*

Sistemas color/monocromáticos

- *Adaptador CGA (Color Graphics Adapter)*
Modo texto: 25 líneas y 80 columnas, es decir 25×80 celdas con 8×8 puntos.
Modo gráfico:
320 \times 200 puntos con 4 colores
640 \times 200 puntos con 2 colores
- *Adaptador EGA (Enhanced Graphics Adapter)*
Modo texto: 25 líneas y 80 columnas, es decir 25×80 celdas con 8×14 puntos.
Modo Gráfico: resolución de 640 \times 350 pixels con 16 colores.
- *Adaptador VGA (Video Graphics Array, 256 Kb)*
Modo texto: 25 líneas y 80 columnas, es decir 25×80 celdas con 9×16 puntos.
Modo Gráfico:
640 \times 480 pixels con 16 colores
320 \times 200 pixels con 256 colores
La memoria de video es de 256 Kb.
- *Super VGA (512Kb de RAM)*
Modo texto: igual que VGA.
Modo Gráfico:
640 \times 480 pixels con 16 colores
320 \times 200 pixels con 256 colores
800 \times 600 pixels con 256 colores.
1024 \times 768 con 16 colores.
- *Super VGA (1Mb de RAM)*
Modo texto: igual que VGA.
Modo Gráfico:
640 \times 480 pixels con 16 colores

320 × 200 pixels con 256 colores

800 × 600 pixels con 256 colores.

1024 × 768 con 256 colores.

- *Adaptador IBM 8514*

Modo texto: igual que VGA.

Modo Gráfico: resolución de 1024 × 768 pixels con 256 colores.

- *Adaptador XGA (eXtended Graphics Adapter)*

Modo texto: igual que VGA.

Modo Gráfico: resolución de 1024 × 768 pixels con 256 colores.

- *Adaptadores con resolución de 1280 × 1024*

Modo Gráfico: resolución de 1280 × 1024 pixels con 256 colores con una paleta de 16 millones, con velocidades de refresco de 60 y 72 Hz.

Actualmente existe la asociación VESA (*Video Electronics Standards Association*) para definir los estándares de las tarjetas gráficas Super VGA.

- **Unidades de lectura de tarjetas perforadas**

Las unidades de lectura de tarjetas perforadas permitían leer información previamente perforada en unas tarjetas de cartón. Hoy en día están obsoletas totalmente.

- **Escritura y lectura de información en forma magnética**

Los discos, disquetes, y cintas magnéticas contienen soportes de información constituidos por un sustrato de plástico o aluminio recubierto por un material magnetizable como óxido férrico u óxido de cromo. La información se graba en unidades elementales o celdas que forman **líneas** (cintas) o **pistas** (discos). Cada celda puede estar sin magnetizar o estar magnetizada, que corresponderán a los valores lógicos 0 y 1. Para escribir o leer en una celda se utilizan señales eléctricas que actúan en una cabeza de lectura/escritura, tal y como se muestra esquemáticamente en la figura 1.8.

Unidades de lectura/escritura de disquetes

Las unidades de lectura de disquetes constituyen el medio habitual para almacenar la información en soporte magnético en los ordenadores. Los disquetes (*floppys disks* o *diskettes*) son discos circulares recubiertos de una capa de óxido magnetizable, envueltos en una carcasa de plástico. Cada una de las circunferencias grabadas constituye una **pista** (*track*). Asimismo el disco se considera dividido en arcos iguales denominados **sectores** (*sectors*), de esta forma cada pista está compuesta de sectores (fig. 1.9).

HARDWARE O SOPORTE FISICO

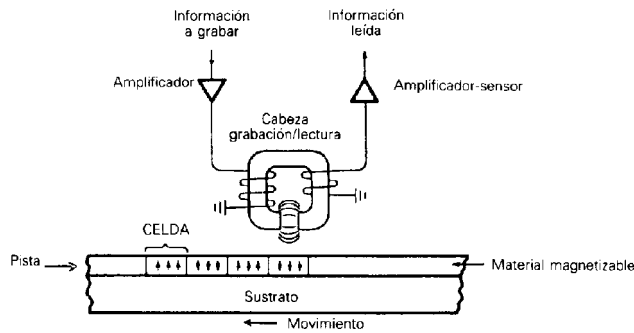


Figura 1.8 Fundamento de la grabación y lectura de un soporte magnético.

Según se observa en la figura 1.9 los sectores de las pistas más exteriores son de mayor longitud que los interiores, ahora bien el número de bits grabados en cada sector es siempre el mismo, con lo que la **densidad de grabación** (*bits grabados por pulgada*) será menor en las pistas exteriores que en las interiores. Esto es evidente si se tiene en cuenta que la velocidad angular de transferencia de información hacia, o desde, la superficie del disco es constante, con lo que el tiempo en recorrer un sector interior es igual al de uno exterior, en ambos casos se grabará la misma cantidad de información.

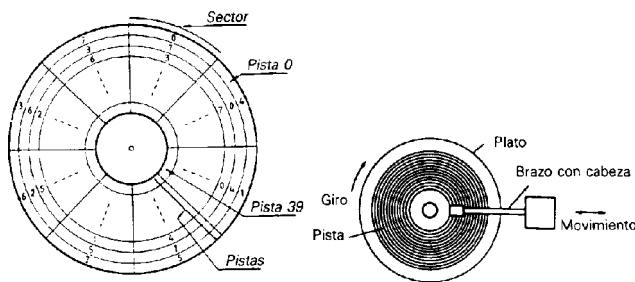
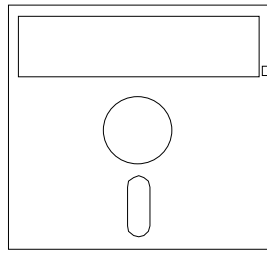


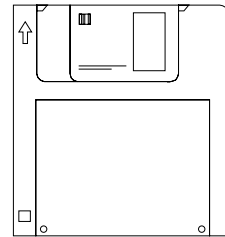
Figura 1.9 Distribución de sectores y pistas en la superficie de un disco.

Actualmente se utilizan habitualmente dos tipos de disquetes según su diámetro, los de 5"1/4 (fig. 1.10) y los 3"1/2 (fig. 1.11).

Los disquetes tienen una abertura para acceso de las cabezas de lectura y escritura, y otro orificio circular para que la unidad por medios ópticos tenga una referencia de alineamiento para localizar pistas y sectores. También tienen una apertura lateral que permite su protección contra escritura.



DISKETTE DE 5 Y 1/4



DISKETTE DE 3 1/2

Figura 1.10 Disquete de cinco pulgadas y cuarto

Figura 1.11 Disquete de tres pulgadas y media

Los disquetes de 5 y 1/4 pulgadas pueden ser de doble cara y doble densidad (almacenan 360 Kb) o de alta densidad (almacenan 1,2 Mb); y los disquetes de 3 y 1/2 pulgadas también pueden ser de doble cara y doble densidad (almacenan 720 Kb), de alta densidad (almacenan 1,44 Mb), y de 2,88 Mb.

La introducción de un disquete en la unidad de lectura/escritura se realiza siempre con la etiqueta hacia arriba (disquetera horizontal) o hacia la izquierda (disquetera vertical).

Unidades de disco fijo o duro

Las unidades de disco fijo o duro (*hard disk* o *winchester*) se caracterizan por su gran capacidad y mayor velocidad que las unidades lectoras de disquetes, pero con la diferencia de que están fijos en el ordenador. Su capacidad va de 40 Mb a 4 Gb en microordenadores, de 0,5 Gbytes a 10 Gbytes en workstations, y capacidades mayores en el resto de los tipos de ordenadores. La palabra duro se utiliza por contraposición de flexible (disquetes); y la palabra fijo por contraposición de móvil (disquetes).

Las unidades de disco duro están compuestas por varios platos que giran solidariamente alrededor de un eje común (fig. 1.12). Las pistas correspondientes se agrupan en **cilindros**.

Las principales características de un disco duro son:

- Capacidad en Mbytes
- Velocidad de acceso en milisegundos

Los tipos más usados de discos duros son IDE (*Integrated Drive Electronics*) y SCSI (*Small Computer System Interface*). Los de tipo IDE tienen un rendimiento menor, pero son los más baratos. Los de tipo SCSI son más caros y ofrecen mejor rendimiento. SCSI es un estándar tanto para microordenadores como para estaciones de trabajo. Los discos duros de tipo SCSI necesitan una tarjeta controladora SCSI, que a su vez puede manejar hasta siete dispositivos SCSI (periféricos que se conectan por la puerta SCSI).

HARDWARE O SOPORTE FISICO

Un aspecto importante en el rendimiento de un disco duro es: el tipo de tarjeta controladora; el tipo de bus donde está conectada (ISA , EISA, Local Bus, ...); y la memoria caché de la tarjeta controladora (área de memoria intermedia entre la CPU y el disco duro).

Existen en el mercado los *Disk Array* que permiten manejar baterías de discos duros, como si fuesen un disco duro único, alcanzando capacidades de 100 GBytes y con mecanismos de protección y recuperación automática de errores.

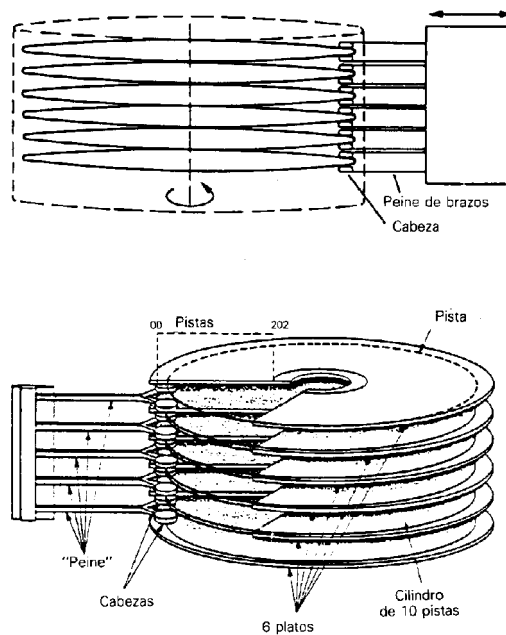


Figura 1.12 Esquema de un disco duro

Unidades de cinta

La unidad de cinta es la forma clásica de almacenar gran cantidad de información en miniordenadores y mainframes. La diferencia principal respecto a los discos es que la información se almacena en líneas, cuyo acceso es secuencial, es decir hay que recorrer la cinta hasta alcanzar la zona de información deseada. Por lo tanto son más lentas que los discos, pero permiten

grabar gran cantidad de información. Un parámetro importante es la densidad de grabación, que se mide en bits por pulgada bpi (*bits per inch*). Las densidades de grabación habituales son 600, 800, 1200, y 1600 bpi.

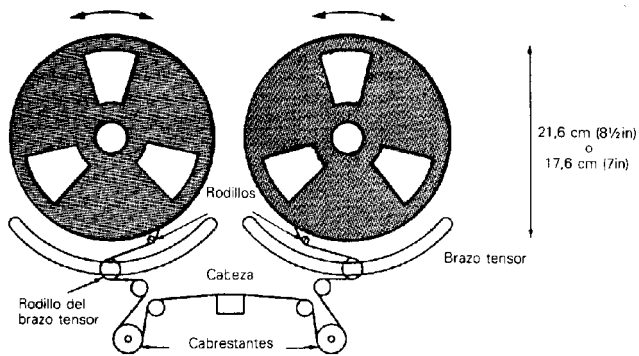


Figura 1.13 Esquema de una unidad de cinta

Unidades de cartucho (cartridge tape)

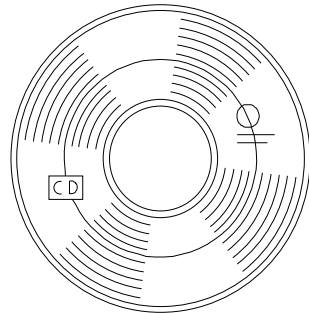
Son un caso particular de unidad de cinta, donde la cinta va introducida en un carrete parecido a los cassettes musicales. Se suelen utilizar para realizar copias de seguridad.

• Unidades de disco óptico

Actualmente se presentan diversas tecnologías:

- **Discos ópticos de sólo lectura CD-ROM** (*Optical Read Only Memory*). Son discos como los de música (*Compact Disc*), pero que almacenan información (véase figura 1.14). Un disco CD-ROM almacena como máximo 680 Mbytes. Las unidades de lectura de discos CD-ROM también pueden manejar discos musicales. Actualmente gran cantidad de programas, manuales, libros, enciclopedias, imágenes fotográficas (*Photo CD*), música, etc... se distribuyen en CD-ROM. La velocidad de acceso a los datos de las unidades de CD-ROM actuales es de 150 y 300 Kbytes/segundo.
- **Discos ópticos que admiten una sola escritura y posteriormente muchas lecturas WORM** (*Write Once Read Many*).
- **Discos magneto-ópticos**. Permiten leer y escribir muchas veces WMRA (*Write Many Read Always*). Los de tamaño de 3,5" tienen una capacidad de 128 Mbytes y los de 5,25" almacenan 680 Mbytes.

HARDWARE O SOPORTE FISICO



DISCO OPTICO

Figura 1.14 Disco CD-ROM

• Impresoras

Las impresoras permiten escribir en papel los resultados obtenidos en los ordenadores. Pueden ser de distintos tipos: matriciales, de margarita, de chorro de tinta, de cadena, de banda, de tambor, laser, etc...

Las impresoras suelen admitir hojas sueltas o el denominado **papel continuo** preparado para ser manejado por las impresoras, y que viene plegado en hojas de igual tamaño con perforaciones laterales para el arrastre (figura 1.15).

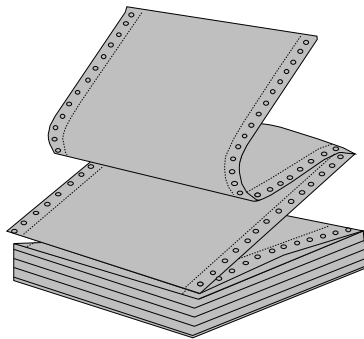


Figura 1.15 Papel continuo

Las impresoras se pueden clasificar desde muchos puntos de vista, una de las clasificaciones está en función de las unidades de impresión:

- ✕ *Impresoras de caracteres*, imprimen carácter a carácter, y su velocidad de impresión se mide en caracteres por segundo (cps). Son impresoras de este tipo: las impresoras matriciales, y las de margarita.

- ⌘ *Impresoras de líneas*, imprimen línea a línea, y su velocidad de impresión se mide en líneas por minuto (lpm). Por ejemplo las impresoras de cadena.
- ⌘ *Impresoras de página*, imprimen página a página, y su velocidad de impresión se mide en paginas por minuto (ppm). Por ejemplo las impresoras laser.

Impresoras matriciales

Las impresoras matriciales están dentro del grupo de las impresoras de caracteres. Son impresoras de impacto que forman los distintos caracteres o gráficos por medio del impacto de las agujas del cabezal, dando lugar a los clásicos listados de ordenador con caracteres formados por puntitos. Existen modelos de 8, 24, y 48 agujas, los dos últimos tipos en alta calidad hacen desaparecer prácticamente la discontinuidad entre puntitos. Ejemplos: IBM Proprinter, EPSON FX-80, etc...

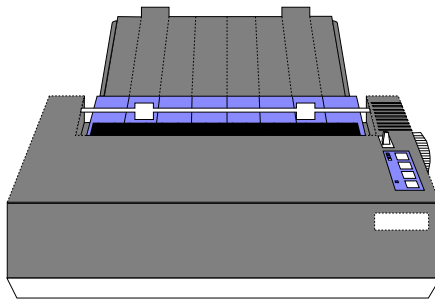


Figura 1.16 Impresora matricial

Impresoras de margarita

Son impresoras de impacto donde los caracteres están en los extremos de tiras metálicas agrupadas alrededor de un eje central común de giro a semejanza de los pétalos de una margarita. No permiten la impresión de gráficos. Ejemplo DIABLO 80.

Impresoras de chorro de tinta

Son impresoras sin impacto, por lo tanto más silenciosas que las anteriores, y que utilizan pequeños inyectores de diminutos chorros de tinta que gracias a que la tinta viene cargada eléctricamente pueden ser dirigidos con bastante precisión mediante campos electromagnéticos.

Estas impresoras no requieren papel especial. Pueden imprimir en blanco y negro o en color. Su resolución suele ser de 300 ppp (puntos por pulgada). Ejemplos: HP-DeskJet 510 (blanco y negro); y HP-DeskJet 550C (color).

HARDWARE O SOPORTE FISICO

Impresoras de cadena

Son impresoras de impacto con los caracteres en una cadena de tipos que gira a gran velocidad alrededor de un par de ejes detrás de los martillos percusores.

Impresoras laser

Son impresoras de página (*page printer*) utilizan las técnicas electroestáticas de las fotocopiadoras para imprimir toda la hoja de una vez.

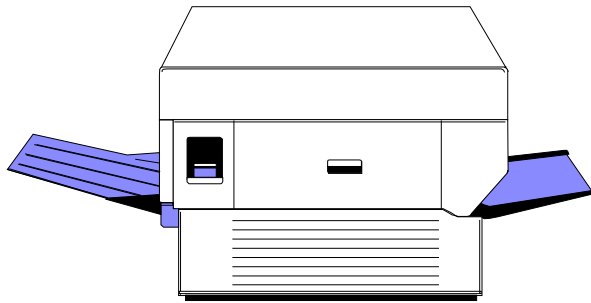


Figura 1.17 Impresora laser

El mecanismo de impresión utiliza un haz de rayos de luz o rayos láser que induce cargas químicas en el papel que después atraen el **toner** químico con la tinta para formar la imagen.

Las impresoras laser pueden utilizar y componer distintos tipos o fuentes (*fonts*) de letra, así como entender los distintos lenguajes de descripción de página: Postscript®, PCL®, etc...

Las principales características de una impresora laser son las siguientes:

- Resolución en puntos por pulgada (ppp). Habitualmente 300, 600 y 1200 ppp.
- Velocidad en páginas por minuto (ppm). Entre 4 y 20 ppm.
- Tamaño de papel (A4, A3,...), número de hojas en cada bandeja y número de bandejas. Capacidad de imprimir en sobres y transparencias.
- Lenguajes de descripción de página que acepta (Postscript®, PCL®, ...)
- Emulaciones de impresoras matriciales que acepta (EPSON, IBM Proprinter, ...)
- Consumo de toner y precio de éste.
- Memoria RAM
- Fuentes (*fonts*) incorporadas, y capacidad para incorporar nuevas fuentes por medio de cartuchos, o cargadas en RAM.

- Número de bocas para introducir los cartuchos de fonts.
- Memoria RAM de la impresora.

Ejemplos: HP LaserJet series I, II, III, y 4; Apple LaserWriter; Tandy LP-1000; etc...

• Trazadores gráficos (plotters)

Son periféricos de salida que permiten representar gráficos en papel o transparencias. Normalmente se clasifican por su tamaño expresado en la norma DIN, suelen ir desde DIN A4 hasta DIN A0.

Existen dos tecnologías: los trazadores vectoriales o mecánicos, y los trazadores raster o electrostáticos.

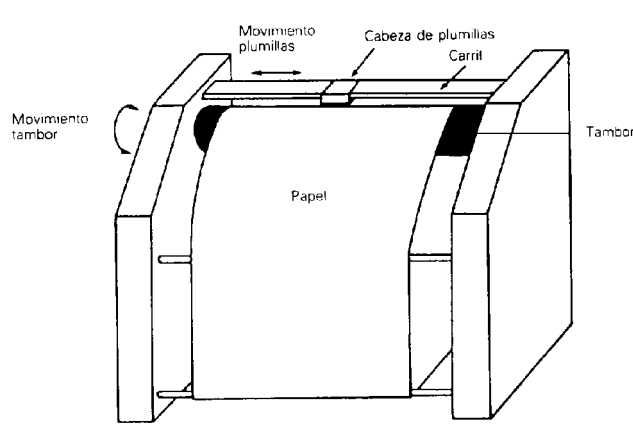


Figura 1.18 Trazador gráfico de tambor

Los **trazadores vectoriales o mecánicos** representan los gráficos mediante el uso de plumas o rotuladores. Se pueden clasificar en dos clases los de **tambor** y los de **cama o planos**. En los de tambor el papel se coloca encima de un tambor que gira, y los de cama o planos el papel permanece estático. Las características principales de este tipo de plotters son:

- Tamaño máximo de papel que admiten, expresado habitualmente según la norma DIN. Los habituales son A4, A3, A1, y A0. También pueden admitir papel en rollo.
- Resolución o capacidad mínima de dibujo expresada en milímetros.
- Número de plumas que acepta simultáneamente: 1, 2, 4, 8 y 16.
- Velocidad de trabajo en cm/s. Habitualmente entre 45 y 520 cm/s.
- Aceleración de la pluma en múltiplos de g (gravedad).

HARDWARE O SOPORTE FISICO

- Soportes que acepta: papel, transparencias, papel vegetal, papel poliester, etc...
- Tipos de plumas que acepta: rotulador, rotring, etc...
- Tamaño del buffer. Es una memoria intermedia entre el ordenador y el *plotter* que permite almacenar el dibujo, y liberar el ordenador. Suelen ser de 1 a 4 Mbytes en los plotters de tamaño DIN A0.
- Conexiones: RS-232, GP-IB, paralelo (Centronics), ...
- Lenguajes que soporta. El más extendido es el HP-GL.

Los **trazadores raster o electrostáticos** representan los planos mediante transferencia electrostática del plano completo, son mucho más rápidos y más caros que los vectoriales. Sus características principales son:

- Resolución en puntos por pulgada (ppp).
- Número de colores que utiliza.
- Tamaño máximo de papel que admite.
- Memoria RAM.
- Tamaño de buffer.
- Conexiones: RS-232, GP-IB, paralelo (Centronics),...
- Lenguajes que soporta. El más extendido es el HP-GL y Postscript[®] Color.

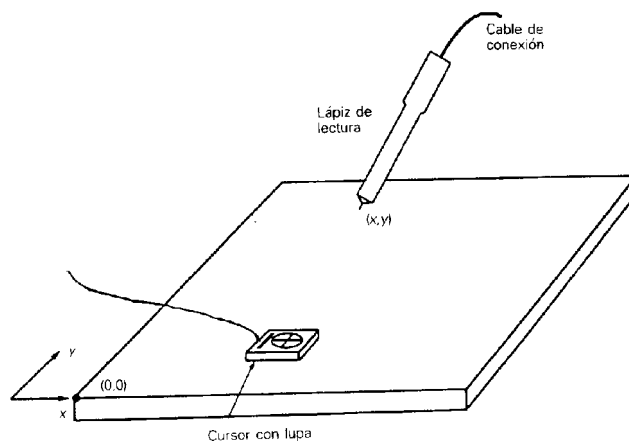


Figura 1.19 Esquema de tableta digitalizadora

• **Tabletas o mesas digitalizadoras**

Las tabletas digitalizadoras permiten introducir gráficos y planos en los ordenadores, pasando manualmente el cursor (botonera o lápiz) por encima de la línea a digitalizar (como si se estuviese calcando), automáticamente se transfieren las coordenadas (x,y) de los distintos puntos que forman la imagen, unas detrás de otras (fig. 1.19).

Una tableta digitalizadora consta de tres elementos:

- Tabla o tablero rectangular donde se ubica el dibujo a digitalizar.
- Cursor con el cual el operador recorre el dibujo.
- Circuitos electrónicos internos a la tabla que permiten en todo momento determinar las coordenadas del cursor

Las tabletas también se utilizan para facilitar el manejo de los programas de CAD al permitir configurarse como menú de opciones del programa (fig. 1.20).

Las principales características de las tabletas digitalizadoras son las siguientes:

- Tamaño máximo de documento a digitalizar.
- Resolución en milímetros.
- Posibilidad de trabajo con distintos programas de CAD.
- Conexiones: RS-232, GP-IB.
- Tipos de cursores: botoneras (nº de botones) y lápices.

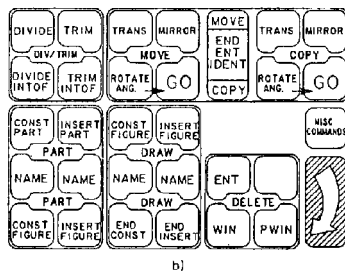
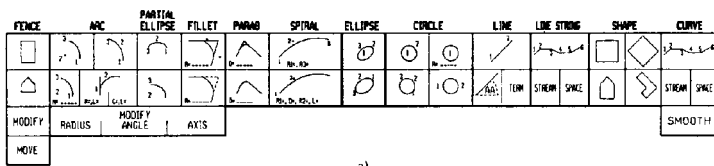


Figura 1.20 Menú de tableta

HARDWARE O SOPORTE FISICO

• Lápiz óptico

El lápiz óptico (*light pen*) consiste en un lápiz con una fotocélula que conectado al ordenador mediante un cable permite pintar directamente en pantalla. Permite introducir gráficos directamente señalando a la pantalla.

• Scanner

El scanner permite introducir imágenes directamente al ordenador, por medio del rastreo punto a punto de la imagen. La resolución de un scanner se mide en puntos por pulgada (ppp), habitualmente alcanzan los 400, 600, 800, 1200 y 2400 puntos por pulgada.

Otra característica del scanner es el tamaño de la imagen que es posible introducir, normalmente oscila entre los scanner de mano y el DIN A0.

Las imágenes introducidas se almacenan en ficheros con formatos estándar, el más utilizado es el formato **TIFF** (*Tag Image File Format*), otros formatos utilizados son: PCX, BMP y GIF. Estos formatos reciben el nombre de formatos *raster* dado que la información se almacena punto a punto, en contraposición a los formatos *vectoriales* en los cuales se almacenan las coordenadas de los vectores o líneas que componen las figuras (ejemplos de formatos vectoriales: DXF, CGM, e IGES). Existen programas, denominados *vectorizadores*, que permiten el paso de formatos *raster* a *vectoriales*, pero tan sólo dan resultados satisfactorios en algunos casos particulares.

Las principales características de un scanner son:

- Tamaño máximo del documento a introducir: van desde el scanner de mano, DIN A4, A3 y A0.
- Resolución en puntos por pulgada (ppp). Normalmente 400, 600, 1200, y 2400 ppp.
- Tipos de formatos para almacenar imágenes (TIFF, PCX, BMP,...).
- Capacidad de funcionar acoplado a un software de reconocimiento óptico de caracteres (OCR), con alimentación automática de papel.
- Disponibilidad de alimentador automático de papel.

• Modem

Las líneas telefónicas transmiten la información en forma analógica, mediante una onda portadora sinusoidal que es modificada a conveniencia por la voz. Los ordenadores procesan la información en forma digital.

El MODEM (*MOdulación-DEModulación*) es un periférico que permite comunicarse a los ordenadores entre sí, por vía telefónica, para poder realizarlo necesita realizar dos operaciones **modulación** (convierte la señal digital en analógica) y **demodulación** (convierte la señal analógica

en digital), de ahí el nombre de modem. Las velocidades de transmisión se miden en *baudios*, aproximadamente un bit por segundo. Las velocidades de transmisión habituales son 1200, 2400, 9600 y 14400 baudios.

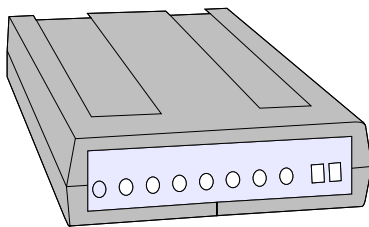


Figura 1.21 Modem externo

• **Unidades de lectura de códigos de barras**

La lectura por código de barras (*Bar Coding Reading*) se ha ido extendiendo desde sus inicios en la década de los 60, para el control de almacenes, ventas, piezas, etc...

Los artículos se codifican mediante combinaciones de barras de distintos espesores, y las unidades de lectura los decodifican rastreando con fotocélulas la presencia o ausencia de luz al pasar un lápiz luminoso o un scanner de mesa.

Existen diversos estándares de codificación UPC, EAN, CODE39, y el del servicio postal de los EE.UU.

Código 39



123456789

Código postal EE.UU.



02364

Figura 1.22 Diversos códigos de barras

HARDWARE O SOPORTE FISICO

• Palanca de juegos (joystick)

Es un dispositivo empleado para mover rápidamente el cursor por la pantalla en cualquier dirección sin tener que recurrir a las teclas de movimiento del cursor. Su uso se ha popularizado por los videojuegos y ordenadores domésticos.

El mecanismo es una palanca que gira sobre una rótula en cualquier dirección de los 360 grados posibles del plano, conectada a dos potenciómetros circulares y perpendiculares, que producen dos tensiones $V(x)$ y $V(y)$, proporcionales a los valores x e y , respectivamente. Estos valores analógicos se convierten en digitales por medio de un conversor analógico/digital.

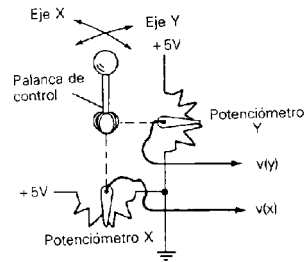


Figura 1.23 Palanca de juegos o joystick

• Ratón

El ratón (*mouse*) dispositivo auxiliar para señalar en la pantalla las distintas instrucciones al ordenador.

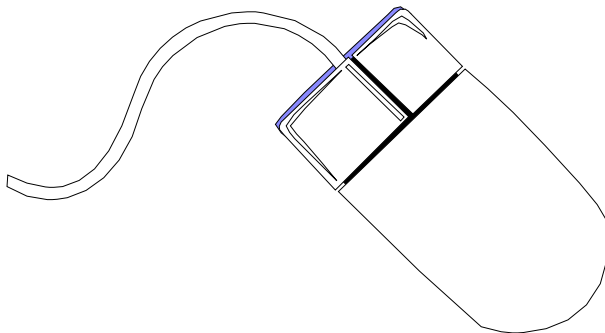


Figura 1.24 Ratón

Internamente está constituido por una bola que puede girar libremente, y se acciona haciéndola rodar sobre una superficie. La bola es solidaria con dos rodamientos o sensores perpendiculares entre sí, cuyos desplazamientos se detectan eléctricamente de forma similar a la palanca de juegos. En función de como se detecten estos desplazamientos los ratones pueden ser electromecánicos, opticomecánicos y ópticos.

- **Tarjetas de sonido**

Son tarjetas que se introducen en las ranuras de expansión (*slots*) del ordenador y permiten la generación de sonido. Existen diversos estándares *Sound Blaster*[®], *Adlib*[®], etc...

- **Tarjetas de entrada/salida de video**

Son tarjetas que se introducen en las ranuras de expansión (*slots*) del ordenador y permiten la captura de imágenes de video (tarjetas de entrada); o la generación de imágenes a video (tarjetas de salida).

- **Filmadora**

Es un periférico utilizado en autoedición por las imprentas, la salida en vez de hacerse en papel se hace en papel fotográfico. Su principal ventaja es su resolución, por encima de los 1200 ppp, y la posibilidad de construir directamente las planchas para ser introducidas a las máquinas offset. Este libro está editado en ordenadores personales y filmado en imprenta.

- **Unidades de entrada/salida de señales analógicas**

Permiten conectar al ordenador con diversas máquinas o dispositivos de medida que envían una señal analógica. Por ejemplo registros de temperaturas que se convierten a una señal en voltios, y que a su vez se convierte por medio de un convertidor analógico-digital en un valor numérico dentro de un rango previamente establecido.

REDES DE ORDENADORES

Una red local o LAN (*Local Area Network*) es un sistema de transmisión de datos que permite compartir recursos e información por medio de ordenadores o redes de ordenadores. Las redes locales están diseñadas para facilitar la interconexión de una gran variedad de equipos de tratamiento de información dentro de un centro. El término red incluye el *hardware* y *software* necesarios para la conexión de los dispositivos y para el tratamiento de la información. El término *local* indica que la red está en un entorno restringido de radio inferior a 4 kilómetros. Lo contrario son las WAN, o redes de área amplia.

Las características que definen una red local son:

SOFTWARE O SOPORTE LOGICO

- a) Un medio de comunicación a través del cual se pueden compartir todos los periféricos, dispositivos, programas y equipos, independientemente del lugar físico.
- b) Una velocidad de transmisión elevada para que pueda adaptarse a las necesidades de los usuarios y del equipo. Normalmente entre 1 y 20 millones de bits por segundo.
- c) Una distancia entre puestos de trabajo relativamente corta (2000 o 3000 metros).
- d) Los cables de conexión suelen ser coaxiales o de fibra óptica.
- e) La topología de la red local: en bus, anillo o estrella. Estas topologías están definidas mediante diversos estándares: Ethernet (IEEE 802.3), Token Ring®, y Apple LocalTalk®.
- f) La facilidad de uso, por medio de un sistema operativo de red. Los más utilizados son: Novell Netware®, Microsoft LAN Manager®, Windows para grupos®, Windows NT®, Lantastic®, IBM LAN Server®, Apple LocalTalk®, y Apple EtherTalk®.
- g) Un ordenador o varios que gestionan todas las operaciones que se realizan en la red.

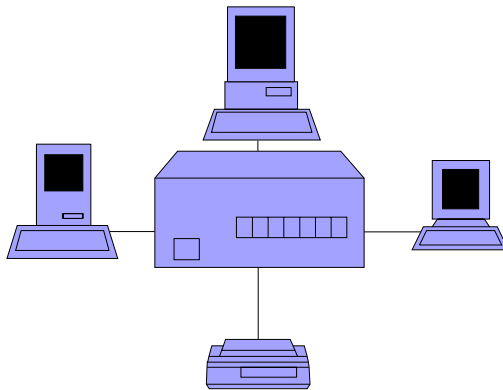


Figura 1.25 Red local

1.4 SOFTWARE O SOPORTE LOGICO

Hasta este momento tan sólo se ha mostrado el hardware del ordenador. El **software o soporte lógico** (en francés *logical*) lo componen *todos los programas necesarios para realizar con el ordenador el tratamiento de la información*. Así por ejemplo son *software* los programas de tratamiento de textos, las hojas de cálculo, los sistemas operativos, los compiladores y los intérpretes de los lenguajes de programación y cualquier programa escrito por nosotros mismos o por un programador.

SISTEMAS OPERATIVOS

Anteriormente se estudiaron las componentes básicas de un ordenador, viéndose los conceptos de hardware y software. El sistema operativo es un conjunto de programas que sirven de puente entre el hardware y los programas de aplicación.

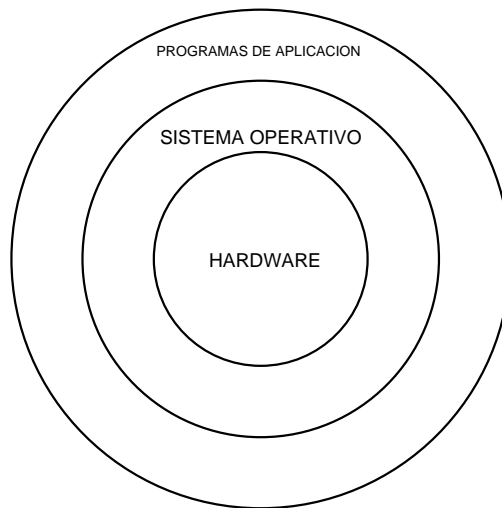


Figura 1.26 Situación del sistema operativo

Se pueden definir los **sistemas operativos** como el *conjunto de programas encargados de coordinar las tareas que debe de ejecutar el ordenador, optimizando su rendimiento y permitiendo una utilización cómoda al usuario.*

El fin primordial de todo sistema operativo es simplificar al máximo el trabajo al usuario, encargándose de realizar tareas tales como gestionar el espacio en las unidades de almacenamiento, proporcionar acceso fácil y rápido a los ficheros, dialogar con los diversos periféricos, etc...

En resumen, los dos **objetivos** fundamentales de los sistemas operativos son:

- *favorecer la relación usuario-máquina*
- *optimizar la eficacia de los recursos del ordenador*

Existen gran cantidad de sistemas operativos, entre ellos destacan los que no están orientados a un hardware concreto, y que se pueden adaptar a muchos ordenadores. Algunos de estos sistemas operativos son : CP/M, UNIX, MS-DOS, PICK, ... Una clasificación puede ser sistemas operativos **abiertos**, si son independientes de un hardware concreto o fabricante; y sistemas operativos **propietarios** en caso contrario.

• Funciones del sistema operativo

Entre las distintas **tareas** del sistema operativo se pueden destacar las siguientes:

- ⊠ *La gestión del propio sistema operativo.* Por ejemplo controla en todo momento cuales son los módulos que deben estar en la memoria del ordenador y carga del disco los necesarios.
- ⊠ *Asignación de recursos.* Es una de las tareas más complejas, pues tiene que dar prioridades entre los distintos periféricos o usuarios, organizando colas de espera.
- ⊠ *Gestión de ficheros.* El primer concepto que surge, cuando se habla de almacenar información, es el concepto de archivo o fichero, en inglés "*file*". Se puede definir un **fichero** como un *conjunto de elementos que contiene información*, al cual se le asigna un nombre. El sistema operativo se encarga de buscar espacio disponible para los ficheros en el dispositivo de almacenamiento, y de controlar que la lectura y la grabación sean correctas. También facilita otras funciones, como la obtención de copias de ficheros.
- ⊠ *Protección de la información.* El usuario puede proteger sus ficheros para que los demás no puedan leer y/o escribir en ellos, evitando el acceso a la información reservada.
- ⊠ *Planificación, carga, y supervisión de la ejecución de programas o tareas (scheduler).*
- ⊠ *Coordinación de las comunicaciones entre el ordenador y los periféricos.*
- ⊠ *Mantenimiento de un registro de todas las operaciones del ordenador*
- ⊠ *Tratamiento de errores.* En el caso de que el usuario cometa algún error el sistema operativo le enviará un mensaje explicativo.
- ⊠ *Inicialización del sistema o arranque.*

• Estructura general de un sistema operativo

El conjunto de programas que compone un sistema operativo se clasifica en programas de control y programas de proceso.

a) Los **programas de control** pueden ser de tres tipos:

- ⊠ *Programas que realizan la gestión del ordenador:* Controlan el hardware, los programas del usuario y también los propios del sistema operativo.
- ⊠ *Programas que realizan la gestión del trabajo:* planifican el encadenamiento y carga de las tareas a ejecutar.

- ⌘ *Programas que realizan la gestión de datos:* controlan la transmisión de los datos entre la memoria del ordenador y los periféricos.

b) Los **programas de proceso**, facilitan la labor de programación, se subdividen en dos tipos muy diferentes entre sí:

- ⌘ *Programas traductores, compiladores, e intérpretes:* hacen la traducción de un lenguaje de programación a otro lenguaje. Pueden ser de distintos tipos : ensambladores, preprocesadores, compiladores e intérpretes, muchas veces son suministrados independientemente. También existen otros programas que se pueden introducir en este apartado tales como los generadores de programas.
- ⌘ *Programas de utilidad.* Sirven para resolver muchos problemas que aparecen reiteradamente, por ejemplo programas de ordenación, editores, desborradores de ficheros, etc...

• Tipos de sistemas operativos

En los microordenadores, se pueden clasificar los sistemas operativos desde el punto de vista de las tareas y de los usuarios que pueden soportar simultáneamente:

- § **Monousuario y monotarea:** Sólo admite a un usuario y ejecuta un sólo programa o tarea a la vez. Ejemplos CP/M y MS-DOS.
- § **Monousuario y multitarea:** Este tipo de sistema operativo soporta un sólo usuario, pero puede procesar más de un programa o tarea al mismo tiempo. Ejemplos OS/2, y Windows.
- § **Multiusuario y monotarea:** Este tipo de sistema operativo permite que dos o más usuarios compartan la misma CPU y los mismos periféricos. El sistema operativo distribuye la memoria RAM en particiones para cada usuario. Cada uno está limitado a procesar un sólo programa. La CPU ejecuta los múltiples programas de usuario conjuntamente. Ejemplos PICK y PROLOGUE.
- § **Multiusuario y multitarea:** Es el tipo más complejo de sistema operativo, y permite que dos o más usuarios compartan el ordenador y ejecuten múltiples programas simultáneamente. Ejemplos Windows/NT Server, UNIX, THEOS, AIX, XENIX, y OSF/1.

LENGUAJES DE PROGRAMACION

Los ordenadores son máquinas capaces de realizar trabajos muy pesados en periodos cortos de tiempo. Pero se necesita indicarles que es lo que deben de hacer. La programación es el proceso de comunicar al ordenador una secuencia de instrucciones, que señalan las acciones que ejecuta éste.

SOFTWARE O SOPORTE LOGICO

¿Cómo se comunica una persona con un ordenador?

Normalmente a través de un dispositivo de entrada/salida (pantalla y teclado), y por medio de un lenguaje específico, llamado **lenguaje de programación**.

Los textos con las indicaciones al ordenador se llaman programas. Es decir, un **programa** es un *conjunto de instrucciones de un lenguaje de programación*.

¿Cómo se diseña el trabajo con un ordenador?

En primer lugar se analiza la tarea a realizar y se busca la forma más fácil de llevarla a cabo mediante un **algoritmo**, que se puede definir como un procedimiento paso a paso para resolver un problema en una cantidad finita de tiempo. A continuación se escribe el programa, o lo que es lo mismo se indica al ordenador en un lenguaje de programación las distintas sentencias que realizan el trabajo propuesto.

A esta fase se le denomina fase de análisis, y se realiza solamente con papel, lápiz y la inteligencia del analista. Con la frase anterior se quiere poner de manifiesto que desde que surge un problema o trabajo, hasta que se programa en el ordenador, se siguen varias fases.

Una vez que el programa está elaborado sobre papel (muchas veces sólo son bocetos) se pasa al programador para que lo escriba en el ordenador en un lenguaje de programación determinado.

Las relaciones humanas se llevan a cabo a través del lenguaje. Una lengua permite la expresión de ideas y de razonamientos, y sin ella la comunicación sería imposible.

Los ordenadores sólo aceptan y comprenden un lenguaje de bajo nivel, que consiste en largas secuencias de ceros y unos. Estas secuencias son ininteligibles para muchas personas, y además son específicas para cada ordenador, constituyendo el llamado "lenguaje máquina".

La programación de ordenadores se realiza en los llamados *lenguajes de programación* que posibilitan la comunicación de órdenes al ordenador.

Un **lenguaje de programación** se puede definir como una *notación formal para describir algoritmos o funciones que serán ejecutadas por un ordenador*.

• Tipos de lenguajes de programación

Los lenguajes de programación se pueden clasificar según su grado de independencia de la máquina en:

- lenguaje máquina
- lenguaje ensamblador (en inglés "*assembly*")
- lenguajes de medio nivel
- lenguajes de alto nivel
- lenguajes orientados a objetos
- lenguajes orientados a problemas concretos

El **lenguaje máquina** es la forma más baja de un lenguaje de programación. Cada instrucción en un programa se representa por un código numérico, y una dirección (que es otro código numérico) que se utiliza para referir las asignaciones de memoria en la memoria del ordenador.

El **lenguaje ensamblador** es esencialmente una versión simbólica de un lenguaje máquina. Cada operación se indica por un código simbólico. Por ejemplo ADD para adición y MUL para multiplicación. Además, las asignaciones de memoria se dan con nombres simbólicos, tales como PAGO y RATIO. Algunos ensambladores contienen macroinstrucciones cuyo nivel es superior a las instrucciones del ensamblador. Este tipo de lenguajes ofrecen posibilidades de diagnóstico y corrección de errores (*debugging*, literalmente desparasitador) que no son posibles a nivel de lenguaje máquina.

Los **lenguajes de medio nivel** están a caballo entre los de bajo nivel y los de alto nivel, tienen acceso directo a todas las posibilidades de la máquina como los lenguajes ensambladores, y permiten su trasportabilidad como los lenguajes de alto nivel. Ejemplos C y Forth.

Los **lenguajes de alto nivel** tales como ADA, FORTRAN, COBOL, Pascal,... tienen características superiores a los lenguajes de tipo ensamblador, aunque no tienen algunas posibilidades de acceso directo al sistema. Facilitan la escritura de programas con estructuras de datos complejas, la utilización de bloques, y procedimientos o subrutinas.

Los **lenguajes orientados a objetos** permiten manejar tipos abstractos de datos, es decir integrar los datos y los subprogramas en las denominadas clases (tipos objeto en Turbo Pascal), que a su vez pueden heredar propiedades de otras clases, permitiendo un conjunto de características nuevas al lenguaje, que desemboca en una nueva metodología de programación. Ejemplos: Pascal orientado a objetos, C++, Eiffel, Simula, CLOS, y Smalltalk.

Los **lenguajes orientados a problemas concretos** se utilizan para la resolución de problemas en un campo específico. Ejemplos de tales lenguajes son el SQL, DBASE para el manejo de bases de datos, SPSS y BMDP para tratamiento estadístico de datos y el COGO para aplicaciones en ingeniería civil.

Otra clasificación de los lenguajes de programación es basándose en la forma de sus instrucciones, y tipos de datos: *lenguajes imperativos*, *lenguajes declarativos*, *lenguajes funcionales*, *lenguajes lógicos*, *lenguajes orientados a objetos*, y *lenguajes concurrentes*.

Los **lenguajes imperativos o procedimentales** son los que usan la instrucción o sentencia de asignación² como constructor básico de la estructura de los programas. Son lenguajes orientados a instrucciones, es decir la unidad de trabajo básica de estos lenguajes es la instrucción o sentencia. Ejemplos de lenguajes imperativos son Pascal, C, C++, ADA, FORTRAN, COBOL, Modula-2, etc...

² La sentencia de asignación se explica en el capítulo 3

SOFTWARE O SOPORTE LOGICO

Los **lenguajes declarativos** son lenguajes de muy alto nivel que describen de forma muy proxima al problema real el algoritmo a resolver. Hay dos tipos de lenguajes declarativos: *lenguajes funcionales* y *lenguajes lógicos*.

Los **lenguajes funcionales o aplicativos** tienen todas sus construcciones como funciones matemáticas. Es decir no hay instrucciones, todo el programa es una función y las operaciones que se realizan es por composición de funciones más simples. Para ejecutar un programa funcional se "aplica" la función a los datos de entrada (que son los argumentos o parámetros de la función) y se obtiene el resultado (el valor calculado de la función). Ejemplos de lenguajes funcionales: LISP, CLOS, Scheme, APL, Standard ML, Miranda, etc...

Los **lenguajes lógicos** definen sus instrucciones siguiendo una Lógica. El lenguaje de programación lógica más utilizado es el PROLOG, que utiliza la lógica clausal (restringida a las cláusulas de Horn). La programación lógica maneja *relaciones* (predicados) entre objetos (datos). Las relaciones se especifican con *reglas* y *hechos*. La ejecución de programas lógicos consiste en la demostración de hechos sobre las relaciones por medio de preguntas.

Los **lenguajes concurrentes** son los que permiten la ejecución simultánea ("paralela" o "concurrente") de dos o varias tareas. Ejemplo: ADA, Concurrent C, Pascal-S, etc...

Otra forma de clasificar los lenguajes es separándolos en **generaciones**. En la actualidad hay cinco generaciones.

La *primera generación* son los lenguajes máquina y ensamblador.

La *segunda generación* la marcaron los lenguajes con asignación estática de memoria, es decir toda la memoria se asigna en tiempo de compilación, y no se permite ni recursividad³ ni estructuras dinámicas de datos⁴. Por ejemplo los lenguajes FORTRAN y COBOL.

La *tercera generación* la marcaron los lenguajes con asignación dinámica de memoria en tiempo de ejecución, es decir permiten recursividad y estructuras dinámicas de datos. Ejemplos de este tipo de lenguajes son: Pascal, C, Algol68, PL/I, etc...

La *cuarta generación* está marcada por lenguajes de muy alto nivel dedicados a tareas específicas, en muchos casos son denominados herramientas. Una gran parte de ellos están dedicados al manejo de bases de datos y a la generación de aplicaciones. Ejemplos: SQL y sus distintas adaptaciones comerciales, NATURAL, IDEAL, APPLICATION FACTORY, etc...

La *quinta generación* se ha asociado a los lenguajes ligados a la Inteligencia Artificial: sistemas basados en el conocimiento, sistemas expertos, mecanismos de inferencia o procesamiento de lenguaje natural. La mayor parte de este tipo de lenguajes son versiones actualizadas de LISP y PROLOG.

³ El concepto de recursividad se explica en el capítulo 7

⁴ Las estructuras dinámicas de datos se explican en el capítulo 12

• **Traductores, compiladores e intérpretes**

Un lenguaje de alto nivel ha de transformarse en lenguaje máquina antes de ejecutarse. Esta tarea de transformación se puede llevar a cabo de dos formas mediante un traductor o mediante un intérprete.

Un **traductor** es un programa que procesa un texto fuente y genera un texto objeto. El traductor esta escrito en un lenguaje de implementación (LI). El texto fuente está escrito en lenguaje fuente (LF), que suele ser un lenguaje de alto nivel. El texto objeto se produce en lenguaje objeto (LO), que suele ser lenguaje máquina o ensamblador.

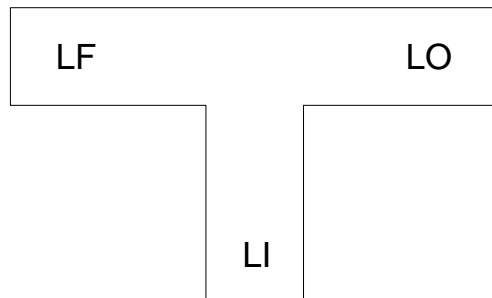


Figura 1.27 Diagrama en T de un traductor

Si el lenguaje fuente es el lenguaje ensamblador y el lenguaje objeto es el lenguaje máquina, entonces al traductor se le llama **ensamblador**, en inglés "*assembler*". Los ensambladores son traductores sencillos, en los que el lenguaje fuente tiene una estructura simple, que permite una traducción de una sentencia fuente a una instrucción en lenguaje máquina, guardándose en casi todos los casos esta relación uno a uno. Hay ensambladores que tienen macroinstrucciones en su lenguaje. Estas macroinstrucciones, de acuerdo con su nombre, se suelen traducir a varias instrucciones de máquina. A este tipo de ensambladores se les denomina **macroensambladores**.

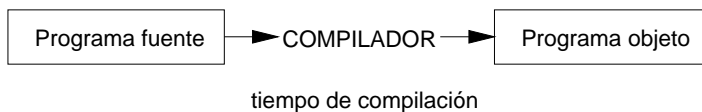


Figura 1.28 Tiempo de compilación

Un traductor que transforma textos fuente de lenguajes de alto nivel (ej. FORTRAN, COBOL, Pascal,...) a lenguaje máquina o a lenguaje ensamblador se le denomina **compilador**. El

SOFTWARE O SOPORTE LOGICO

tiempo que se necesita para traducir un lenguaje de alto nivel a lenguaje objeto se denomina *tiempo de compilación* (figura 1.28). El tiempo que tarda en ejecutarse un programa objeto se denomina *tiempo de ejecución* (figura 1.29).

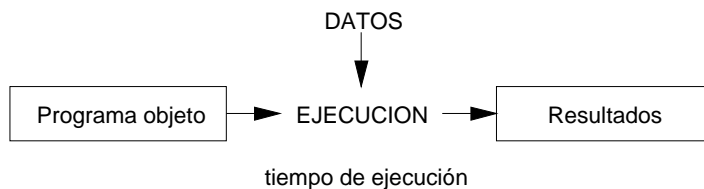


Figura 1.29 Tiempo de ejecución

Entre el proceso de compilación y la ejecución existe el proceso de montaje, que se produce cuando el lenguaje fuente permite una fragmentación de los programas en trozos, denominados de distintas formas según el lenguaje de programación empleado (procedimientos, funciones, subrutinas,...). Dichas partes o trozos pueden compilarse por separado, produciéndose los códigos objetos de cada una de las partes. El **montador de enlaces** (*linker*) realiza el montaje de los distintos códigos objeto, produciendo el módulo de carga, que es el programa objeto completo, siendo el **cargador** (*loader*) quien lo transfiere a memoria (fig. 1.30).

Los **intérpretes** traducen el texto fuente simultáneamente a su ejecución, coexistiendo en memoria el programa fuente y el programa intérprete. Nótese que en este caso la compilación ocurre en tiempo de ejecución. Los lenguajes comúnmente interpretados son el BASIC, LOGO, APL y LISP.

Evidentemente la ejecución de un programa compilado será más rápida que la del mismo programa interpretado. Sin embargo los intérpretes son más interactivos y facilitan la puesta a punto de programas.

SOFTWARE DE APLICACION

Los ordenadores son máquinas de propósito general, que pueden ser programados para hacer prácticamente cualquier cosa.

La programación de los ordenadores es más o menos costosa en función de las características del problema a resolver. Sin embargo existen muchas áreas comunes entre los distintos usuarios de ordenadores, con lo que se consigue el abaratamiento del software. El conjunto de programas comunes a distintos usuarios es lo que se conoce habitualmente con el nombre de **software de aplicación**. En contraposición está el **software a medida**, escrito para un usuario y ajustado exactamente a sus necesidades.

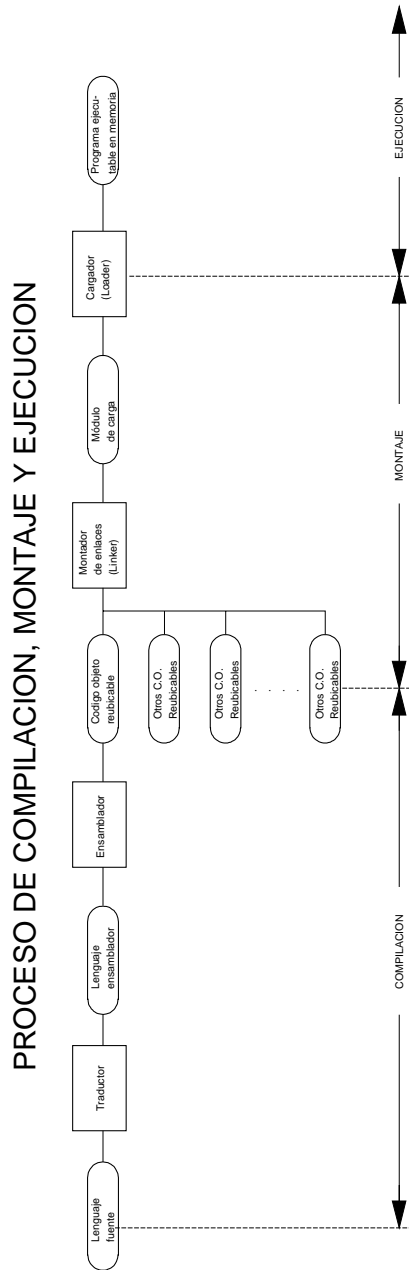


Figura 1.30 Fases de compilación, montaje y ejecución

SOFTWARE O SOPORTE LOGICO

Dentro del software de aplicación más común están los procesadores de texto, las hojas de cálculo, los sistemas de gestión de bases de datos, los programas de comunicaciones, etc...

• Procesadores de texto

Un procesador de textos o tratamiento de texto (en inglés "*word processing*"), es un programa que nos convierte el ordenador en una máquina de escribir, con memoria y capaz de manejar con soltura los textos, haciendo modificaciones, cambiando de orden o mezclando distintos textos. Los textos se pueden guardar a su vez en una unidad de almacenamiento (por ejemplo en un disco).

La secuencia normal de trabajo con un procesador de textos es la siguiente:

- *Introducción del texto*: El primer paso es teclear todo el material manuscrito al ordenador. Hoy en día también se puede lograr esto directamente mediante un scanner y un programa de reconocimiento óptico de caracteres (OCR).
- *Manipulación del texto*: Una vez que el texto esta en la memoria del ordenador, se le puede manejar a voluntad. Puede ser moldeado o corregido, a base de cambiar palabras o párrafos enteros, subsanar errores, intercalar palabras o frases, cambiar de orden,... Hasta que consigamos que el texto quede en las condiciones deseadas.
- *Almacenamiento*: El texto se puede grabar para dejarlo almacenado de forma permanente.
- *Impresión*: Una vez satisfechos con el texto y almacenado este se puede proceder a su impresión en papel.

De esta forma los procesadores de texto pueden mejorar notablemente la productividad respecto a una máquina de escribir.

La calidad de la escritura dependerá en mayor medida de la impresora, que del procesador de textos.

La selección de un determinado producto como procesador de textos puede no ser una tarea fácil. Existen cientos de ellos en el mercado, se citan a continuación algunos para ordenadores personales: Wordstar, EasyWriter, PeachText, Word, Word Perfect, XYwriter, Writing Assistant, VolksWriter, Manuscript, Chiwrite, AmiPro, ...

La evaluación de un producto depende mucho de las necesidades propias del usuario, y de su experiencia. Se pueden destacar algunos aspectos:

- a) Capacidad de tratamiento de caracteres españoles y proceso de fórmulas matemáticas.
- b) Capacidad de mezcla de textos y gráficos.
- c) Capacidad de comunicación con otros procesadores de texto o con otro software de aplicación, por ejemplo: hojas de cálculo, paquetes de gráficos, programas de impresión de direcciones (*mailing*), bases de datos, etc...

- d) Adaptación a las impresoras del usuario.
- e) Diccionarios que permitan la corrección ortográfica, búsqueda de sinónimos y correcciones gramaticales.
- g) Manejo de distintos tipos de letra.
- h) Potencia de instrucciones para lograr una buena maquetación de los textos.
- i) Capacidad para ver en pantalla exactamente lo que se va a imprimir, WYSIWYG (*What You See Is What You Get*), es decir "lo que ve es lo que obtiene".
- j) Capacidad para generar tablas de contenidos, tablas de figuras, tablas de tablas, índices de referencias cruzadas, portadas, etc...
- k) Capacidad de importar textos de otros procesadores de texto.
- l) Capacidad para importar gráficos en distintos formatos: DXF, CGM, TIFF, PCX, etc...
- m) Posibilidad de trabajo en red.
- n) Posibilidad de trabajo en un entorno operativo con iconos.
- o) Lenguaje de programación de macros.

• Editores

Un editor manipula textos pero difiere de los procesadores de textos, en que estos últimos introducen en los ficheros unos caracteres de control para marcar márgenes, punto y aparte, interespaciados,... Mientras que los editores no introducen ninguna de estas marcas, pues suelen usarse para escribir programas de ordenador.

Los editores pueden ser de línea o de pantalla completa. Los editores de línea tan sólo permiten la edición de documentos línea a línea, por ejemplo el editor EDLIN de las primeras versiones del sistema operativo DOS. Los editores de pantalla completa permiten recorrer todo el texto de la pantalla. Por ejemplo EDIT del DOS, Norton Editor, Personal Editor, *vi* del sistema operativo UNIX, etc...

• Hojas de cálculo

Las hojas de cálculo, o planillas electrónicas es uno de los tipos de software de aplicación más utilizado hoy en día como herramienta para la realización de cálculos aritméticos con filas y columnas.

Una hoja de cálculo no es más que la representación en el ordenador de una de esas enormes hojas que los contables suelen usar para registrar información económica y operar con ella. Tienen

SOFTWARE O SOPORTE LOGICO

una forma de cuadro o matriz, compuesta por líneas horizontales y columnas verticales; la intersección de cada fila con cada columna compone una casilla, en la que puede introducirse una cantidad, aunque también puede asignársele un texto o etiqueta a algunas casillas para utilizarlas como información adicional o como títulos.

Una hoja de cálculo típica de ordenador personal permite introducir datos al menos en una matriz de 63 columnas y 254 líneas.

El objetivo perseguido es introducir datos numéricos y programar diversas relaciones matemáticas entre casillas y/o grupos de ellas.

Los números pueden representar presupuestos, previsiones de ventas, análisis de los costes de fabricación o cualquier otro tipo de información.

El proceso de trabajo con una hoja de cálculo consta de la siguientes fases:

- a) Introducción de los datos. Puede hacerse mediante teclado o importándolos de un fichero creado anteriormente.
- b) Manejo de los datos. En esta fase se opera con los datos, hasta obtener los resultados deseados.
- c) Grabación de los datos. Se guardan los datos y/o los resultados en un fichero, para su posterior uso.
- d) Salida de resultados: tablas, gráficos e informes. En general se dirigen a una impresora.

Los principales factores que intervienen para seleccionar una hoja de cálculo son los siguientes:

- a) Nº de filas y de columnas máximo
- b) Velocidad de cálculo
- c) Capacidad de importación de ficheros ASCII, y de otras hojas de cálculo, bases de datos, ...
- d) Capacidad de comunicación con otros productos. Esta faceta es tan importante, que la mayoría de las hojas de cálculo del mercado están dentro de los llamados paquetes integrados.
- e) Disponibilidad de ventanas que permitan visualizar varias hojas o resultados simultáneamente.
- f) Posibilidad y potencia de la programación de macroinstrucciones.
- g) Capacidad de realizar gráficos y tipos de éstos.
- h) Posibilidad de trabajo en red.
- i) Posibilidad de trabajo en un entorno operativo con iconos y ratón.

Algunas hojas de cálculo comercializadas para ordenadores personales son las siguientes: LOTUS 1-2-3, VisiCalc, SuperCalc, Multiplan, CalcStar, PeachCalc, EXCEL, Quattro,...

• **Bases de datos**

Las bases de datos se pueden definir como grandes conjuntos de información interrelacionada, accesible por medio de un sistema de gestión de bases de datos (SGBD).

El objetivo de un sistema de gestión de bases de datos es disponer de una estructura de manejo cómoda, que evite la gestión de muchos archivos individuales, muchas veces con información duplicada, y con un mantenimiento laborioso.

En definitiva una base de datos es un conjunto de ficheros, gestionados de tal forma que el usuario no tiene por qué conocer la estructura interna de su funcionamiento. La principal característica de las bases de datos es la interrelación, que se da entre los datos que la componen, y que permite acceder a un dato por distintos caminos.

Las bases de datos pueden tener distintos tipos de estructura:

- a) *Estructura jerárquica o en árbol.* La conexión entre los datos se establece por medio de jerarquías entre los mismos.
- b) *Estructuras en red.* Permite el acceso a la información de la base de datos estructurada en nodos, con conexiones multidireccionales entre ellos, y sin ninguna jerarquía definida.
- c) *Estructura relacional.* Se estructura la información en forma de tablas, formadas por columnas con datos homogéneos. Las consultas (*vistas*) se realizan aplicando operadores del álgebra relacional a la información contenida en columnas y tablas.
- d) *Basadas en objetos.* Integran las técnicas de las bases de datos tradicionales, con los lenguajes de programación orientados a objetos, y las interfaces de usuario.

Existen diversos productos en el mercado que permiten :

- describir la estructura de la base de datos
- construir la base de datos
- consultar la base de datos
- generar informes

A nivel de microordenadores existen diversos estándares DBASE y SQL. A nivel de miniordenadores el estándar es SQL. Todos ellos son sistemas de gestión de bases de datos relacionales.

También se pueden construir aplicaciones que conecten con las bases de datos y que estén escritas en un lenguaje de alto nivel (Pascal, C, C++, COBOL, etc...)

• **Programas de comunicaciones**

La comunicación es el proceso de enviar y recibir información. En Informática, este termino se aplica a la conexión entre ordenadores, para lograr un intercambio de información entre ellos, o entre ordenadores y otros dispositivos.

Las comunicaciones se pueden realizar entre ordenadores personales y un gran ordenador central, actuando los ordenadores personales como terminales (se dice en este caso que son terminales inteligentes). La comunicación se puede realizar directamente mediante tarjetas o interfaces tipo RS-232c, RS-422, o también mediante línea telefónica, siguiendo diversos protocolos X.25, etc...

Existen dos sistemas básicos para transferir datos entre dos puntos:

a) **Trasmisión de datos en paralelo.** Todos los bits de una palabra binaria son enviados simultáneamente de un punto a otro. Se puede ver en el esquema de la figura 1.31.

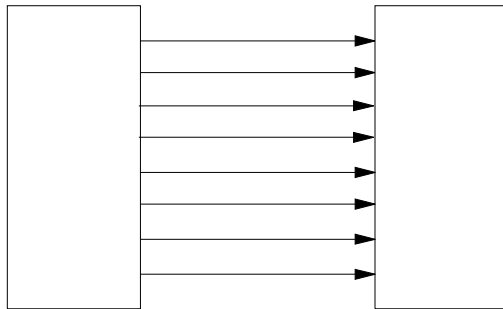


Figura 1.31 Transmisión de datos en paralelo

La ventaja de la transmisión en paralelo es que permite una mayor velocidad, pues al transmitirse todos los bits de una palabra al mismo tiempo, la velocidad de transmisión sólo está limitada por los circuitos digitales que la controlan; la transferencia de una palabra puede realizarse en nanosegundos (10^{-9} segundos). El inconveniente reside en la necesidad de disponer de una línea para cada bit.

La mayoría de los ordenadores personales utilizan una comunicación en paralelo, para conectar el ordenador con la impresora.

b) **Trasmisión en serie.** Los bits de una palabra binaria son transferidos de uno en uno, según una secuencia que circula por una línea de datos única. La figura 1.32 es un esquema de lo explicado anteriormente.

TRANSMISION EN SERIE

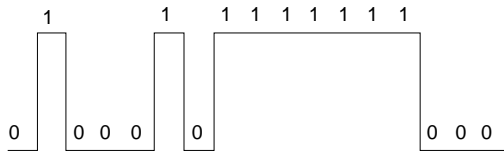


Figura 1.32 Transmisión de datos en serie

En el esquema anterior se representa la transmisión de una palabra de datos como una serie de impulsos eléctricos, que materializan los ceros y unos de la palabra binaria transmitida. La tensión correspondiente a cada bit se mantiene durante un intervalo de tiempo t , fijo para cada bit. La longitud de este intervalo de tiempo, es la que determina la velocidad de transmisión.

El **baudio** es la unidad de medida de transmisión de datos serie. Un baudio representa, aproximadamente, un bit por segundo. Las velocidades típicas de transmisión son: 50, 1.200, 2.400, 4.800, 9.600, 19.200, 38.400, 57.600, y 115.200 baudios. Dividiendo por 10 la velocidad en baudios, tenemos aproximadamente la velocidad de transmisión expresada en caracteres por segundo.

Los ordenadores personales suelen disponer de un adaptador de comunicaciones asíncronas, que convierte los datos transmitidos en paralelo por el bus de datos del ordenador, en un tren de impulsos en serie, o viceversa.

El término **asíncrono** indica "no igual en el tiempo", y se utiliza para designar procesos que no están regulados por una frecuencia determinada, sino que están regulados por una señal de comienzo y otra de parada. Así en una transmisión asíncrona, los datos se transmiten bit a bit, encabezados por un bit de comienzo y terminando por dos bits de parada.

El término opuesto es **síncrono**, que indica que se produce según una frecuencia determinada. Una transmisión síncrona no necesita bits de arranque y de cierre, sino que se efectúa con una frecuencia determinada.

La salida del adaptador de comunicaciones asíncronas puede tener varias configuraciones, la más utilizada es el interfaz EIA RS-232-C. Se trata de un estándar de la Electronics Industry Association que define los tipos de señales de entrada, sus niveles lógicos, y la configuración del conector (figura 1.33).

Otra terminología usada en comunicaciones se refiere a los tipos de circuitos empleados para las comunicaciones:

- a) circuito *simplex*: permite que los datos fluyan en una sola dirección.

SOFTWARE O SOPORTE LOGICO

b) circuito *half-duplex*: permite recibir o enviar datos de forma alterna.

c) circuito *full-duplex*: para transmitir y recibir en forma simultánea.

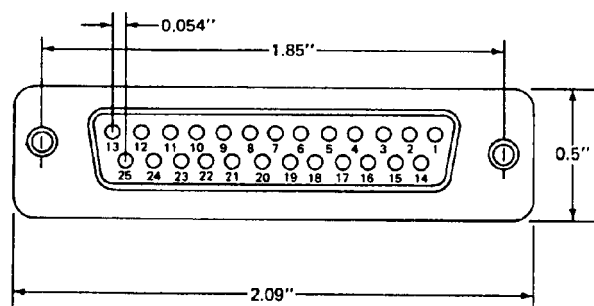


Figura 1.33 Conector RS-232-C de 25 pines

Algunos conceptos utilizados en comunicación de ordenadores, son los siguientes:

- ▣ *Buffer*. Area de memoria intermedia, que almacena temporalmente la información enviada o recibida.
- ▣ *Echo*. Proceso por el cual se visualizan los datos transmitidos en una comunicación.
- ▣ *Host*. Término empleado para describir el ordenador remoto.

• Paquetes integrados

Se pueden combinar varios paquetes en uno. A este tipo de programas se le suele llamar software integrado. Los paquetes integrados suelen constar como mínimo de un procesador de textos, una hoja de cálculo, una base de datos, programas de comunicación, programas de gráficos. Su principal característica es un interface de usuario común para todos los programas, y posibilidad de intercambio de datos entre ellos, sin ningún proceso especial.

Ejemplos de paquetes integrados son SYMPHONY, LOTUS, Open Access, Framework, ...

• Entornos operativos

Los entornos operativos incorporan las ventajas de los sistemas operativos, y además añaden un manejo fácil y cómodo de los recursos del sistemas y el manejo de las aplicaciones, por medio de una interface de usuario, habitualmente gráfico e intuitivo. El interface de usuario suele estar compuesto de pequeños símbolos denominados iconos, con una funcionalidad asociada a ellos (equivalente a un comando de un sistema operativo tradicional).

Ejemplos de este tipo de entornos son el GEM, Windows, TopView, y DesqView en el sistema operativo DOS, X-Windows en el sistema operativo UNIX, MOTIF en el sistema operativo OSF/1, y System 7 en Macintosh.

La forma habitual de trabajo con estos paquetes es mediante el uso de iconos, ratón y ventanas.

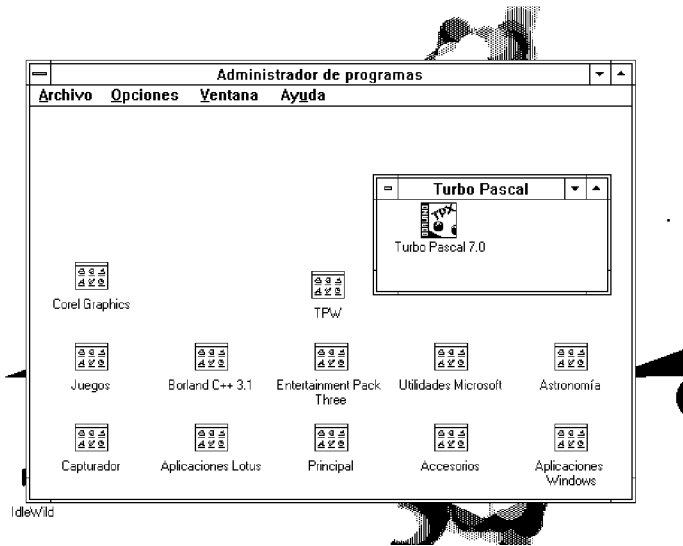


Figura 1.34 Entorno Windows

• **Diseño asistido por ordenador**

En primer lugar vamos a definir un conjunto de términos o siglas muy utilizados:

- CAD *Computer Aided Design*. Diseño asistido por ordenador.
- CADD *Computer Aided Design Drawing*. Dibujo diseñado con ayuda de ordenador.
- CAE *Computer Aided Engineering*. Ingeniería asistida por ordenador.
- CAM *Computer Aided Manufacturing*. Fabricación asistida por ordenador.
- CIM *Computer Integrated Manufacturing*. Fabricación integrada por ordenador.

Como puede observarse, se parecen todas las siglas mucho, pero su significado es diferente.

A nivel de informática personal, la mayor parte de los productos disponibles en el mercado son paquetes de CADD, es decir utilizar los ordenadores personales para realizar dibujos, planos, esquemas, ... ; guardarlos en una unidad de almacenamiento; modificarlos; y representarlos por medio de un dispositivo, tal como un plotter. También es habitual el uso de programas de diseño gráfico, con inclinaciones más artísticas.

SOFTWARE O SOPORTE LOGICO

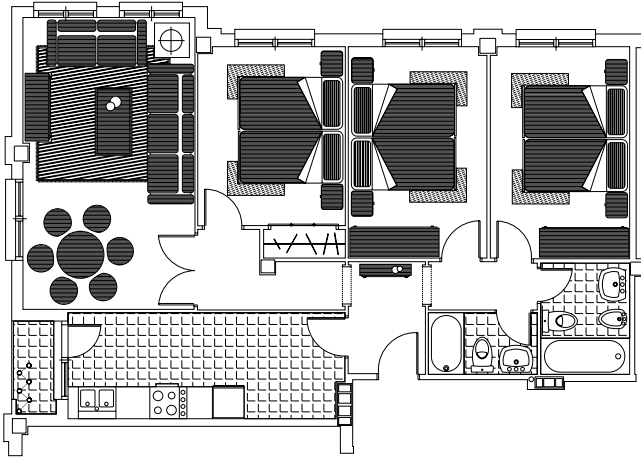


Figura 1.35 Diseño de viviendas

El CAD, comprende no solamente el dibujo, sino la modelización, cálculo y análisis. Es decir, si se diseña una pieza mecánica, se ha de hacer su representación gráfica, realizar el análisis de tensiones y esfuerzos a los que puede someterse la pieza y evaluar su grado de deterioro.

El CAM utilizaría los datos suministrados por el CAD, y los comunicaría a una máquina herramienta, para construir una pieza mecánica. Las máquinas herramientas están dirigidas por programas denominados de *control numérico*.

Cuando se logra una integración completa del proceso de CAD y CAM, se tiene lo que se llama el CIM.

Todos los procesos anteriores se pueden agrupar bajo el nombre de CAE.

En el mercado de los ordenadores personales existen diversos productos que permiten una realización completa del CADD. Ejemplo de estos productos son: AUTOCAD, ROBOCAD, MicroStation, etc... Ejemplo de programas de diseño gráfico son: COREL Draw, Micrographx, Freelance, ...

Las fases de trabajo con un paquete de CADD, se pueden resumir :

- ✘ *Introducción de los planos o gráficos.* En esta fase se ha de representar en el ordenador toda la información. Se puede utilizar la ayuda de diversos periféricos, tales como tabletas digitalizadoras, ratón, scanner, ... También se pueden utilizar bibliotecas de gráficos o de símbolos de una determinada especialidad (electrónica, mecánica, topografía, arquitectura, etc...).
- ✘ *Almacenamiento de la información.* Se puede realizar en disquetes o en disco duro.

- ✘ *Recuperación de la información.* Se puede extraer la información de las unidades de almacenamiento, para volver a modificarla o para reutilizarla. Un concepto importante en esta fase es el de parametrización, que consiste en guardar los esquemas en función de unos parámetros modificables. Así por ejemplo un tornillo, con unos parámetros determinados, se puede modificar para representar toda una familia.

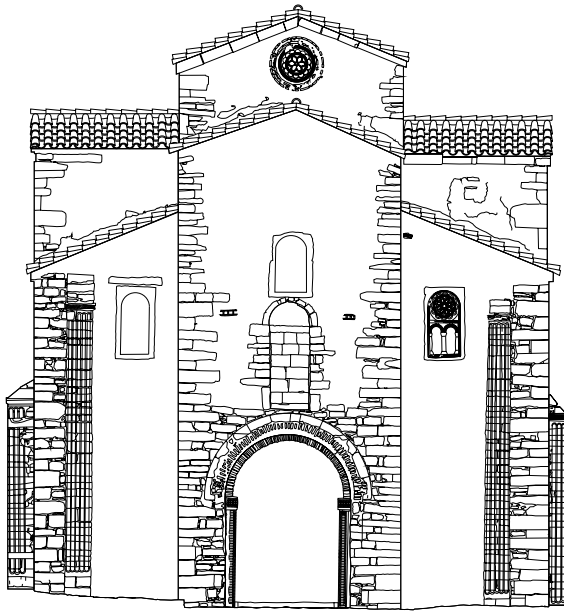


Figura 1.36 Reconstrucción de monumentos histórico-artísticos: San Miguel de Lillo (Oviedo)

- ✘ *Manipulación de los gráficos e información* para realizar nuevos diseños, modelado sólido, y animación de imágenes.
- ✘ *Representación de la información.*
 - Gráficamente. Mediante plotter o impresora.
 - Mediante listados a impresora de componentes. Por ejemplo número de elementos de una característica, listados en ficheros para tratar por otros programas, etc...
 - Mediante salidas a video, diapositiva, etc...

Las aplicaciones del CAD son múltiples Arquitectura, Ingenierías, Reconstrucción de monumentos y Arqueología, diseño, etc...

Los formatos para intercambio de ficheros gráficos más utilizados son los siguientes: DXF, IGES, Metafiles (CGM) y HPGL. Todos ellos son formatos vectoriales.

SOFTWARE O SOPORTE LOGICO

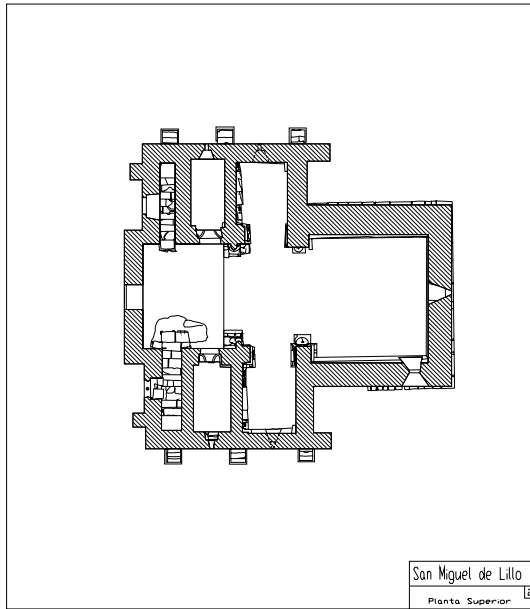


Figura 1.37 Reconstrucción de monumentos histórico-artísticos. Planta de San Miguel de Lillo

• Sistemas de Información Geográfica (GIS)

La representación de mapas, modelos digitales del terreno, topografía, ordenación del territorio son algunas de las aplicaciones de los Sistemas de Información Geográfica. Se basan en combinar la potencia de los programas de representación de gráficos con sistemas de gestión de bases de datos. Los GIS mezclan tanto información raster (introducida por scanner), como vectorial, ambas georreferenciadas, es decir referida a coordenadas espaciales.

Los más utilizados son ARC/INFO, TIGRIS, MAP/INFO, Microstation GIS, etc...



Figura 1.38 Ejemplo de consulta en un GIS

• Enseñanza Asistida por Ordenador

Existen gran cantidad de programas dedicados a enseñar con ordenador, estos programas se agrupan bajo las siguientes denominaciones.

- CAI, *Computer Aided Instruction*. Enseñanza asistida por ordenador (EAO).
- CAL, *Computer Aided Learning*. Aprendizaje asistido por ordenador.

• Multimedia

Multimedia es una disciplina que integra las distintas técnicas informáticas con la imagen y el sonido para crear productos informáticos (programas de juegos, enciclopedias, documentales, etc...) o hacia otros medios (cine, video, televisión, música, etc...). Existe una gran cantidad de *software multimedia* para el manejo de los periféricos de sonido y video, tanto para su generación externa, como para la construcción de programas con sonido e imágenes (video).

• Sistemas expertos

Un sistema experto intenta "imitar" el razonamiento de un experto humano.

Un sistema experto trabaja a partir de los datos suministrados por la base de conocimientos y por la base de hechos, que resulta de su trabajo con problemas concretos.

Los sistemas expertos más famosos son el DENDRAL (reconocimiento de fórmulas químicas), el MEYCIN (diagnóstico médico), PROSPECTOR (búsqueda de yacimientos) y otros de diagnósticos de averías en redes eléctricas o de ordenadores.

También existen herramientas de desarrollo como el KEE, OPS5, Personal Consultant, etc...

CUESTIONES

1.5 CUESTIONES

- 1.1 ¿Cuál es la unidad mínima de proceso de información?
- 1.2 Definir byte.
- 1.3 ¿Qué es un Kbyte?
- 1.4 Indicar las diferencias entre memoria RAM y ROM
- 1.5 Enumerar tres periféricos de entrada de datos.
- 1.6 Citar tres periféricos de salida.
- 1.7 ¿Cuál es la principal tarea del sistema operativo?
- 1.8 ¿Qué es un sistema operativo multitarea?
- 1.9 Clasificar los lenguajes de programación desde el punto de vista de independencia de la máquina en que se ejecutan.
- 1.10 Explicar la diferencia entre un compilador y un intérprete.
- 1.11 Definir tiempo de compilación y tiempo de ejecución.
- 1.12 Citar cinco tipos de software de aplicación.

1.6 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Este capítulo tan sólo pretende ofrecer una visión muy superficial de algunos conceptos elementales de Informática, para una mayor profundización en aspectos generales puede consultarse la obra de *A. Prieto, A. Lloris y J.C. Torres* titulada *Informática general*, publicada por *McGraw-Hill* (1989), que contiene una perspectiva amplia y profunda de los conceptos generales de Informática. Otras obras generales son las de *E. Pardo, Informática General, Ed. Jucar* (1984); *Sanders, Informática: presente y futuro*, en la editorial McGraw-Hill (1983); y *LL. Guilera, Introducción a la informática, Ed. Edunsa* (1988).

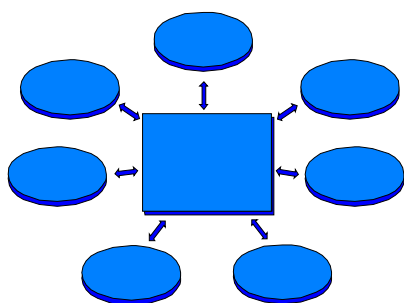
Sobre aspectos concretos de ordenadores personales pueden consultarse la obra de *P. Norton, Inside the IBM PC and PS/2* (1992), que existe una traducción de una edición más antigua en la editorial Anaya, titulada *El IBM PC a fondo* (1987). También hay una obra de divulgación titulada *Así funciona un ordenador por dentro*, de *W. Ron* (1993) que contiene gran número de gráficos en color explicando el funcionamiento del ordenador y sus periféricos.

Sobre el sistema operativo DOS puede consultarse las obras *DOS 5: manual del usuario* (1992), y *Novedades MS-DOS 6* (1993), de *J.Yraolagoitia* en la editorial Paraninfo.

Si el lector desea profundizar en temas relacionados con lenguajes de programación puede consultar la obra de *R. Sethi, Lenguajes de programación: Conceptos y constructores*, Ed. *Addison-Wesley* (1989, versión castellana 1992), que hace un recorrido sobre los lenguajes, sus

INTRODUCCION A LA INFORMATICA

tipos, y los distintos paradigmas. En el aspecto más concreto de diseño y construcción de compiladores la obra más aceptada y profunda, es el famoso libro del dragón de *A. Aho, R. Sethi y J.D. Ullman*, titulado *Compiladores: principios, técnicas y herramientas*, Ed. Addison-Wesley (1986, versión castellana 1990). Para una introducción elemental puede consultarse la obra de *J.M. Cueva*, *Conceptos básicos de traductores, compiladores e intérpretes*, Cuaderno didáctico nº9. Dto. de Matemáticas de la Universidad de Oviedo (1992).



CAPITULO 2

CONSTRUCCION DE PROGRAMAS

CONTENIDOS

- 2.1 Introducción
- 2.2 Las fases del proceso de programación
- 2.3 Análisis del problema
- 2.4 Desarrollo de la solución
- 2.5 Técnicas de descripción de algoritmos
- 2.6 Construcción de la solución en forma de programa
- 2.7 Prueba de programas
- 2.8 Documentación de programas
- 2.9 Mantenimiento de programas
- 2.10 Ejercicios resueltos
- 2.11 Ejercicios propuestos
- 2.12 Ampliaciones y notas bibliográficas

2.1 INTRODUCCION

Este capítulo pretende dar una visión general sobre como se diseñan los programas; quizá sea demasiado abstracto en una primera lectura, pero deberá volverse a él según se avanza los distintos capítulos del libro y se tenga una perspectiva más amplia.

LAS FASES DEL PROCESO DE PROGRAMACION

El objetivo de la programación de ordenadores es la construcción de programas, que deben reunir las siguientes características: *correctos, eficientes, y fácilmente modificables*.

- a) *correctos*: un programa que obtenga resultados erróneos es inútil.
- b) *eficientes*: si los programas son poco eficientes, su ejecución puede ser costosa en tiempo, almacenamiento y claridad.
- c) *fácilmente modificables*: el mantenimiento del programa, es decir, su ajuste a las necesidades, siempre cambiantes, de los usuarios, consume la mayor parte del tiempo de los programadores.

Evidentemente la primera característica (corrección) es la más importante, y decisiva, aunque no hay que olvidar que el incumplimiento de las otras dos características puede conducir a un programa al fracaso. En este libro de texto se pretende que el lector aprenda a programar cumpliendo estas tres características.

2.2 LAS FASES DEL PROCESO DE PROGRAMACION

Desarrollar una solución informática para las necesidades de un usuario es un proceso que se puede descomponer en las fases: *análisis del problema, desarrollo de la solución, descripción de los algoritmos, construcción del programa, prueba del programa y mantenimiento del programa*.

2.3 ANALISIS DEL PROBLEMA

Consiste en conocer las necesidades del usuario y las especificaciones que debe de cumplir el programa. En el caso de aplicaciones muy especializadas el programador recibe un informe detallado de un especialista en el tema, con él que colabora estrechamente hasta que se clarifiquen todas las necesidades. Los errores que se cometen en esta fase son los peores de remediar en etapas posteriores, y pueden echar por tierra todo el trabajo.

2.4 DESARROLLO DE LA SOLUCION

En esta fase se diseña el algoritmo que resuelve el problema planteado.

La metodología del desarrollo de algoritmos debe de sistematizarse al máximo, y no debe de improvisarse. Existe la desafortunada tendencia de muchos programadores a sentarse delante de la máquina antes de que el problema haya sido resuelto realmente. Se estudiarán dos metodologías: *diseño descendente y diseño orientado a objetos*.

DISEÑO DESCENDENTE

El diseño descendente (*top-down*) es una técnica para el desarrollo sistemático de algoritmos, que se basa en la máxima "divide y vencerás". Aunque también recibe otros nombres, según distintos autores:

- § Refinamiento por pasos o refinamiento progresivo (*Wirth*)
- § Modelado iterativo multinivel (*Zurcher y Randell*)
- § Programación jerárquica (*Dijkstra*)

El método consiste en resolver el problema separándolo en distintos niveles de abstracción. No se debe de intentar pasar directamente del problema a instrucciones de ordenador.

Se plantea el problema a resolver con términos del problema mismo (nivel 1 de abstracción).

Se descompone en varios subproblemas, que serán lo más independientes entre sí que sea posible, enunciados en términos del problema (nivel 2 de abstracción).

Se vuelven a descomponer los subproblemas en otros subproblemas (nivel 3 de abstracción).

Se repite el proceso hasta que se llega a subproblemas que son muy concretos, en contra de los enunciados tan abstractos que se hacían en los primeros niveles. Esta es la esencia del diseño descendente. Se trabaja a partir de una solución muy abstracta (**nivel 1-top**) hasta llegar a una solución final (**down**), mediante una serie de refinamientos sucesivos. En el capítulo 7 se explica como cada uno de estos subproblemas se corresponde con un subprograma.

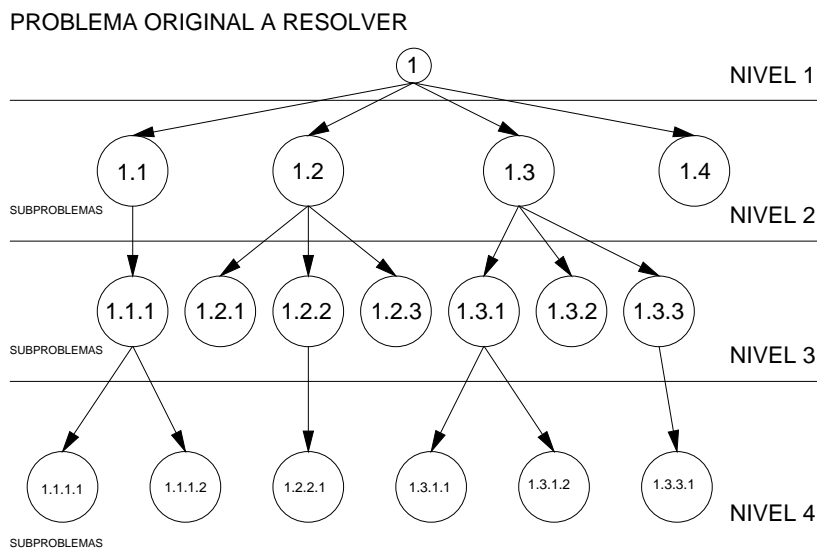


Figura 2.1 Esquema de análisis descendente

DESARROLLO DE LA SOLUCION

Ejemplo 2.1

El problema consiste en leer un número N y dar la lista de los cuadrados perfectos entre 1 y N (se supone N entero y positivo). Un cuadrado perfecto es un número cuya raíz cuadrada es un número entero. Por ejemplo los cuadrados perfectos que hay entre 1 y 30 son 1,4, 9, 16 y 25.

Solución. El diseño descendente del problema planteado se muestra en la figura 2.2. Obsérvese que en cada nivel, el problema debe de quedar completamente resuelto. Los cuatro rectángulos, a partir de los cuales no salen líneas inferiores (también llamados **terminales**), pueden considerarse como los puntos de partida para la construcción del algoritmo. Se marcan con un recuadro doble. Los otros rectángulos (también llamados **no-terminales**), definen estructuras de decisión de más alto nivel que describen como se ha desarrollado el trabajo; pueden aparecer en el algoritmo final en forma de subprogramas muy generales o tal vez sólo como comentarios.

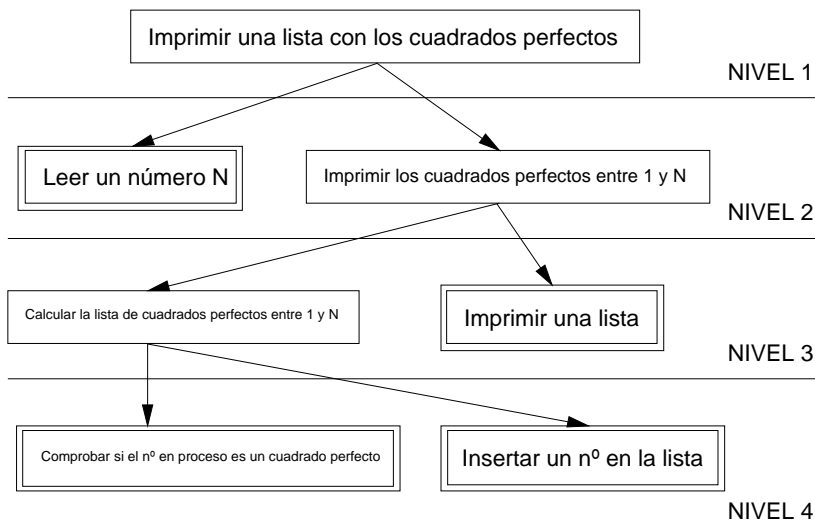


Figura 2.2 Ejemplo de análisis descendente

DISEÑO ORIENTADO A OBJETOS

El diseño orientado a objetos consta de cinco pasos fundamentales:

- 1º Identificación y definición de objetos
- 2º Definición de las clases (tipos *object*)
- 3º Diseño de la jerarquía de clases (tipos *object*)
- 4º Construcción de bibliotecas de clases.

5° Desarrollo de marcos de trabajo reutilizables (*frameworks*).

Al revés que el diseño descendente, el diseño orientado a objetos es *ascendente*, se pretende pasar de los problemas concretos modelados como objetos del mundo real a una jerarquía de clases (tipos *object*), en la cual los objetos concretos del problema a resolver se describen como una especialización de la jerarquía de objetos diseñada. En el capítulo 13 se profundiza con ejemplos concretos de objetos y clases. Turbo Pascal denomina a las clases como tipos objeto o *object*.

• Identificación y definición de objetos

El diseño orientado a objetos comienza por encontrar los objetos. Por una parte pretende ser un diseño intuitivo, dado que el mundo real está compuesto por objetos, que tienen unas propiedades y pueden realizar unas acciones. En teoría con la observación se pueden identificar los objetos. Las propiedades se pueden definir con *datos* y las acciones serán definidas con *métodos*. Los datos y los métodos juntos definen un objeto, que será un caso particular de la *clase*.

La clase define la forma general de un conjunto de objetos. Los objetos también se denominan *instancias* de una clase. En Turbo Pascal se denomina tipo *object* a las clases.

Existen metodologías para la identificación de objetos, una de ellas es la de *Booch*, que utiliza el siguiente método gramatical:

- a) El diseñador relata en prosa (lenguaje natural) la definición del problema y la descripción de la solución.
- b) Los nombres del texto pueden ser identificadores potenciales de clases de objetos
- c) Los verbos identifican los métodos o acciones que se realizan con las clases
- d) La lista resultante de clases (nombres) y métodos (verbos) se utilizará para comenzar el proceso de diseño

Ejemplo 2.2

- *Definición del problema*: Desarrollar un programa que determine la situación y calcule el coste de quesos de bola, de barra y de rosca.
- *Descripción de la solución*: Se muestra la situación con tres coordenadas. Dados los volúmenes del queso de bola, de barra y de rosca; los costes de queso por unidad de volumen; y los costes fijos, se calcula el coste.

Solución. La identificación y definición de objetos por medio del análisis gramatical sería:

Se muestra la situación con tres coordenadas. Dados los volúmenes del queso de bola, de barra, y de rosca; los costes de queso por unidad de volumen; y los costes fijos, se calcula el coste.

Se han subrayado los objetos seleccionados: *coordenadas*, *bola*, *barra*, *rosca*, y *queso*. El método o acciones que se realizan sobre los objetos es *calcular* coste.

DESARROLLO DE LA SOLUCION

• Definición y organización de relaciones entre clases

Una vez definidos los objetos, el paso siguiente es reunir las definiciones comunes de objetos en una abstracción de clase. La abstracción es la tarea continua del diseñador orientado a objetos. Una abstracción útil es el resultado de la organización inteligente de la descripción del problema en elementos independientes e intuitivamente correctos. Las abstracciones útiles deben ser fáciles de comprender, de especializar, y de convertirse en elementos de una librería⁵ genérica.

No existen reglas estrictas para identificar clases, pero se pueden definir las siguientes directrices generales:

- Modelar con clases las entidades que aparecen de forma natural en el problema.
- Diseñar métodos genéricos de finalidad única
- Evitar métodos extensos
- El diseñador debe pensar más en hacer clases genéricas, que en resolver el problema concreto.

Los errores más frecuentes son: la creación de clases innecesarias y la declaración de clases que no lo son. Existen muchas clases en potencia, tantas como objetos o más; las innecesarias no son fundamentales para la resolución del problema.

Los diseñadores siempre deben de ver las clases ya existentes en primer lugar, tanto las suministradas en los lenguajes y en los entorno de desarrollo, como las creadas por los equipos de programación. Siempre se debe dedicar tiempo adicional de diseño a generalizar las clases ya existentes.

Ejemplo 2.3

Continuando con el ejemplo 2.2, se pueden diseñar las clases *ubicación*, *esfera*, *cilindro*, y *toroide*. Se eligen cuerpos geométricos como una abstracción de las formas de los quesos.

• Diseño de la jerarquía de clases

Las clases se pueden definir como un caso particular de otras clases, mediante el mecanismo de herencia. Con la herencia una clase se puede definir como una especialización de otra clase, añadiendo datos y métodos a los de otra clase ya existente, o redefiniendo los métodos ya existentes en la clase de la cual se hereda. Si se utilizan unos métodos denominados *virtuales* no se puede saber en tiempo de compilación cual es el método de la jerarquía de clases que se va a aplicar, sin embargo se aplicará en tiempo de ejecución el método adecuado (polimorfismo).

⁵ El concepto de librería o biblioteca aparece en el capítulo 7, como un conjunto de subprogramas. Se profundiza el mismo concepto en el capítulo 11.

Aunque no existen metodologías formales para diseñar jerarquías de clases, se pueden definir tres reglas:

- Definición de métodos con nombres y comportamientos estándar, de forma que la jerarquía de clases los comparta sin lugar a equívocos.
- Reducir el tamaño de los métodos. Cuando un método crece demasiado, es menos probable que sea heredado, por el contrario los métodos más reducidos, pueden ser heredados, y por tanto perfeccionados o ignorados de forma selectiva.
- Construcción de clases abstractas. Las clases abstractas están en las zonas más altas de la jerarquía de clases, no se utilizan directamente, y deben de contener los métodos comunes a todas las clases que heredan de ellas.

Ejemplo 2.4

Continuando con el ejemplo 2.3, se diseña la jerarquía de clases que se muestra en la figura 2.3, donde se ve como se crea una clase abstracta denominada *solido*, de la cual heredan *esfera*, *cilindro*, y *toroide*. La clase *solido* hereda a su vez de *ubicación*. La solución completa se desarrolla en el capítulo 13.

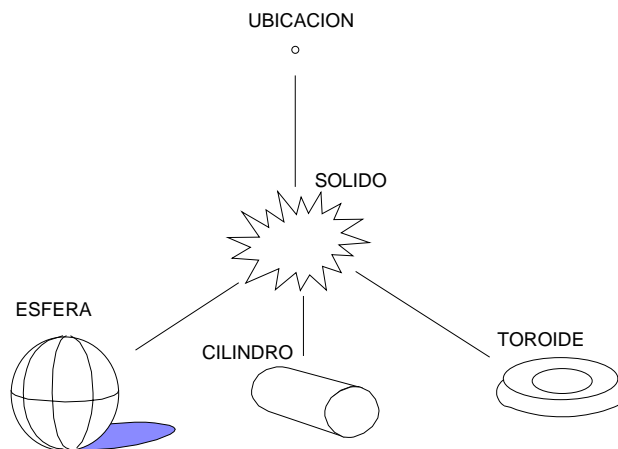


Figura 2.3 Ejemplo de jerarquía de clases

• Construcción de bibliotecas de clases

Una biblioteca de clases es un conjunto de clases desarrolladas para un determinado programa o conjunto de programas. El objetivo de las bibliotecas de clases es reutilizar al máximo los

TECNICAS DE DESCRIPCION DE ALGORITMOS

elementos comunes de una aplicación. Una buena biblioteca de clases debe de ser profunda y estrecha, con varios niveles de subclasificación. Las jerarquías anchas y poco profundas sugieren la necesidad de mejorar el diseño.

• Desarrollo de marcos de trabajo reutilizables (*frameworks*)

Los marcos de trabajo son el objetivo último del diseño orientado a objetos, ya que ellos representan el nivel más alto de abstracción.

Los marcos de trabajo reutilizables (*frameworks*) son bibliotecas de clases genéricas, es decir se pueden utilizar por cualquier programa, pero a su vez especializadas, por ejemplo en manejo de interfaces gráficos de usuario. Un ejemplo de marcos de trabajo reutilizables son *Turbo Vision* y *ObjectWindows*⁶.

Los marcos de trabajo proporcionan a los programadores una capacidad mayor que las bibliotecas de clases, al permitir la reutilización de un diseño completo orientado a objetos.

2.5 TECNICAS DE DESCRIPCION DE ALGORITMOS

Las técnicas para describir los algoritmos que se explicarán a continuación son: *lenguaje natural*, *organigramas*, *pseudocódigo*, y *lenguaje algorítmico*.

LENGUAJE NATURAL

Se entiende por lenguaje natural, aquel que hablan los humanos, y por contraposición los lenguajes de programación los que sirven para comunicarse con los ordenadores.

El método aparentemente más sencillo de describir un algoritmo es relatarlo verbalmente. Sin embargo el lenguaje natural es en muchas ocasiones farragoso e impreciso, y con frecuencia un vehículo poco fiable para transmitir información tan exacta. Puede existir el peligro de malinterpretar o perder información. Otro problema del lenguaje natural es que resulta poco conciso en la descripción de los algoritmos.

Por estas razones se han buscado mejores métodos de descripción de algoritmos.

ORGANIGRAMAS

Una forma de describir algoritmos es mediante gráficos. Un organigrama es una representación gráfica de la lógica de un algoritmo. A continuación se estudiarán dos tipos de organigramas: *los diagramas de flujo*, y *los diagramas estructurados de Nassi-Shneiderman o de Chapin*.

⁶ Se estudiarán en los capítulos 14 y 15.

• **Diagramas de flujo**

Para la confección de diagramas de flujo el Instituto de Normalización Americano ANSI (*American National Standards Institute*) ha homologado un conjunto de símbolos y signos, algunos de los cuales se muestran en la figura 2.4.

Los signos se unen con líneas y flechas indicando la dirección de flujo del programa. El programa comienza en un símbolo ovalado con la palabra *inicio*, y termina con otro símbolo ovalado con la palabra *fin*. Las líneas de flujo o flechas indican la secuencia de las operaciones del algoritmo. Los rectángulos indican algún tipo de proceso, que se describe en su interior. Los rombos indican que existe una bifurcación en función de la condición expresada en su interior, si la condición es cierta se dirige el flujo por un camino, en caso de que sea falsa por otro. La repetición de una parte del algoritmo un determinado número de veces se simboliza con un hexágono, se produce lo que se denomina un *bucle*. Los bucles o estructuras de control repetitivas también pueden estar controlados por una condición al principio o al final del bucle.

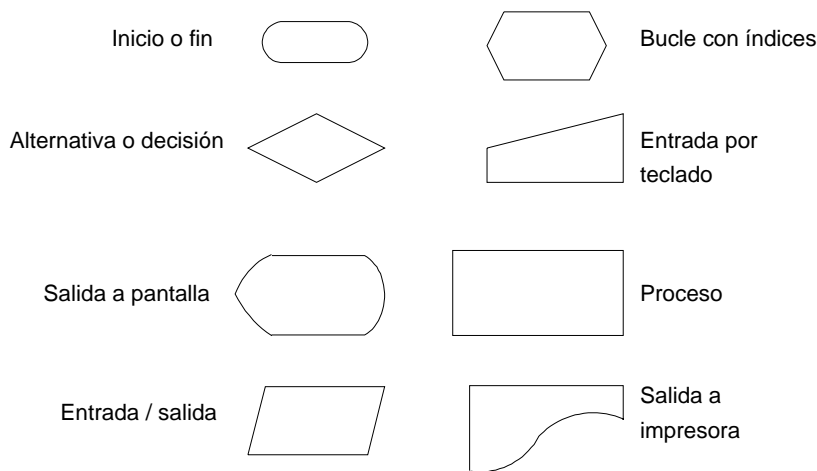


Figura 2.4 Símbolos de los diagramas de flujo

Uno de los objetivos de este libro es enseñar al lector a programar de forma estructurada. Aunque los diagramas de flujo permitan todo tipo de bifurcaciones, sin ningún tipo de condición, la programación estructurada se basa en la descripción de los algoritmos mediante estructuras de control de flujo de tres tipos: *secuenciales*, *alternativas*, y *repetitivas*. Las estructuras de control de programación estructurada se caracterizan por tener una sola entrada y una única salida. La representación de los bloques básicos de las distintas estructuras de control de la programación estructurada se muestra en la figura 2.5. En la figura 2.5 sólo se ha representado un tipo de estructura de control repetitiva y otro de tipo alternativa, el resto se representan en las figuras 2.6 y 2.7. Las estructuras de control de flujo se estudiarán en el capítulo 6.

TECNICAS DE DESCRIPCION DE ALGORITMOS

La *estructura de control secuencial* ejecuta las acciones sucesivamente unas a continuación de otras, sin posibilidad de omitir ninguna acción y sin poder hacer bifurcaciones. Tiene por tanto una sola entrada y una única salida.

Una *estructura de control alternativa* bifurca el flujo de un algoritmo según se cumplan una o varias condiciones. Las estructuras alternativas pueden ser: simples, dobles o múltiples. La *estructura alternativa simple* ejecuta una acción si la condición es cierta, en caso contrario se la salta. La *estructura alternativa doble* permite la elección entre dos acciones según la condición sea cierta o falsa. La *estructura alternativa múltiple* permite la elección entre varias acciones según los valores de una variable selector. En la figura 2.6 se representan las estructuras de control alternativas, véase como debe de conservar una sola entrada y una sola salida.

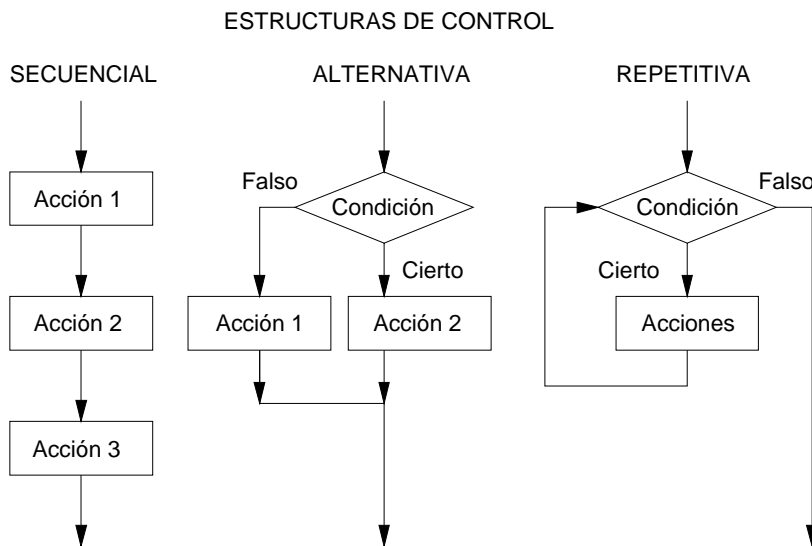


Figura 2.5 Estructuras de control

Una *estructura de control repetitiva* permite ejecutar las acciones un número de veces que puede estar definido *a priori*, o indefinido hasta que se cumpla una determinada condición. Se denomina *bucle* o *lazo* al conjunto de las acciones repetidas. Las estructuras de control repetitivas en Pascal pueden ser de tres tipos: *estructura FOR*, *estructura WHILE*, y *estructura REPEAT*.

En la *estructura FOR* el número de repeticiones se conoce antes de realizar el bucle, por medio de sus índices. La *estructura WHILE* repite las acciones mientras la condición de control del bucle sea cierta, esta condición está colocada al principio del bucle. La *estructura REPEAT* repite las acciones mientras la condición de control del bucle sea falsa, esta condición está colocada al final del bucle.

ESTRUCTURAS DE CONTROL ALTERNATIVAS

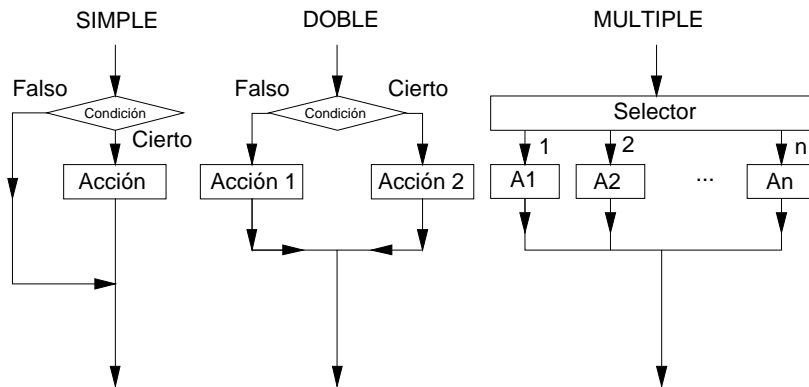


Figura 2.6 Estructuras de control alternativas

En 1966 *Böhm* y *Jacopini* demostraron el **teorema de la programación estructurada**, que dice: *todo algoritmo puede ser descrito utilizando solamente tres tipos de estructuras de control: secuencial, alternativa y repetitiva*. No se necesitan bifurcaciones incondicionales para la descripción de los algoritmos.

ESTRUCTURAS DE CONTROL REPETITIVAS

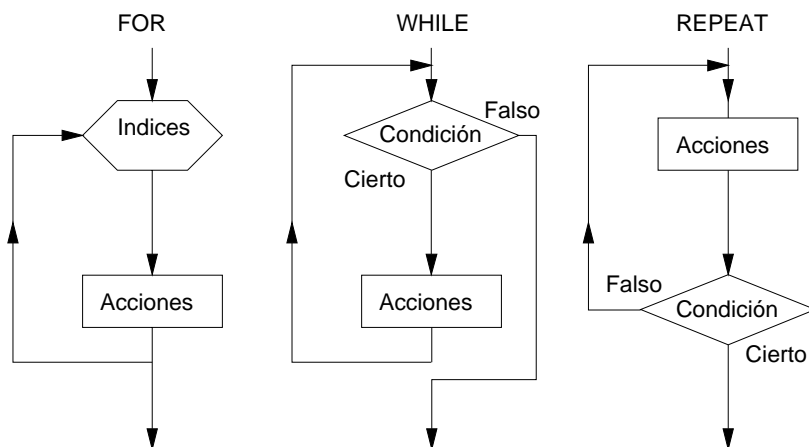


Figura 2.7 Estructuras de control repetitivas

Ejemplo 2.5

Realizar el diagrama de flujo del algoritmo de imprimir los cuadrados perfectos entre 1 y N.

Solución. Es el diagrama de la figura 2.8, sería una solución en forma de algoritmo del diseño descendente del ejemplo 2.1. En el diagrama de flujo se ha empleado la función *Sqrt()* para calcular raíces cuadradas, y la función *Trunc()* para realizar truncamientos de la parte decimal de un número real. Para indicar la asignación, es decir que una variable toma un valor se utiliza el símbolo \leftarrow precedido de la variable y seguido por el valor a tomar. Se puede leer como *c* toma el valor 0. En los bucles con índices (estructura de control FOR) se indica el nombre del índice, seguido por un signo \leftarrow , y los valores extremos que toma el índice. También se emplea un vector *v*, para almacenar los cuadrados perfectos, entre paréntesis se coloca el subíndice.

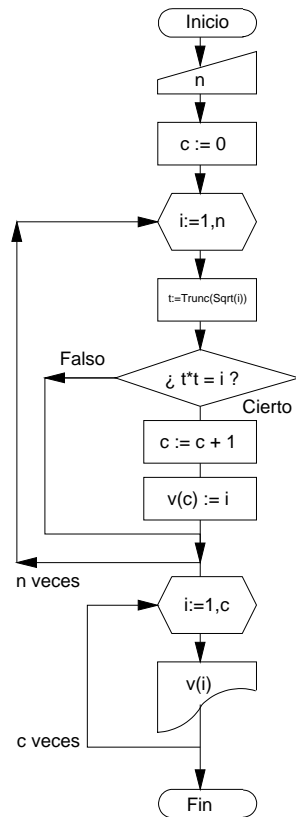


Figura 2.8 Ejemplo de diagrama de flujo

Inconvenientes de los diagramas de flujo

- § Los diagramas de flujo de algoritmos complejos y detallados son muy laboriosos de realizar.
- § Las acciones a seguir después de un símbolo de decisión, pueden ser difíciles de encontrar debido a la complejidad de los caminos.
- § No existen normas fijas en la elaboración de los diagramas de flujo, que permitan introducir todos los detalles que el usuario desee.
- § La facilidad de hacer bifurcaciones puede crear malos hábitos de programación, es decir se pueden construir programas poco estructurados.
- § Los diagramas de flujo muestran la lógica de un algoritmo pero oscurecen su estructura.

Se utilizarán diagramas de flujo para explicar como se programan las estructuras de control en el capítulo 6.

Normalmente se utilizan los diagramas de flujo para describir pequeños algoritmos, cuando se está comenzando a programar, o para mostrar esquemas muy generales de una aplicación informática.

• Diagramas estructurados de Nassi-Shneiderman o de Chapin

Es como un diagrama de flujo con las flechas omitidas y con bloques contiguos. Con esto se evitan las bifurcaciones incondicionales de los diagramas de flujo, con lo que se consigue que la programación sea estructurada. Los diagramas de las estructuras de control se representan en la figura 2.9.

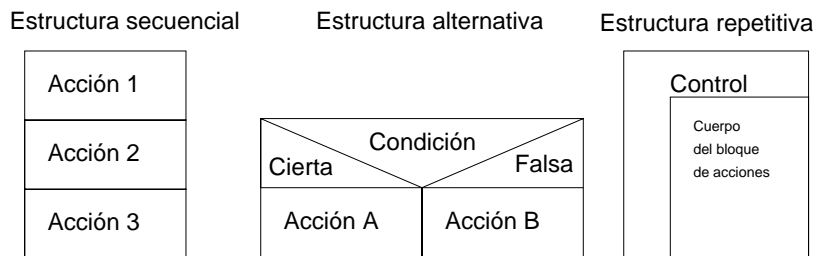


Figura 2.9 Diagramas de Nassi/Shneiderman o de Chapin

Ejemplo 2.6

Realizar el diagrama de Chapin del algoritmo de impresión de los cuadrados perfectos entre 1 y N.

Solución. Usando una notación similar al ejemplo 2.5, el diagrama estructurado se representa en la figura 2.10.

Ventajas de los organigramas estructurados

- § Permiten ver mejor la estructura y enlace entre los módulos.
- § Son más compactos, aunque no indican exactamente los medios por los cuales se realizan las entradas y salidas (pantalla, impresora,...).

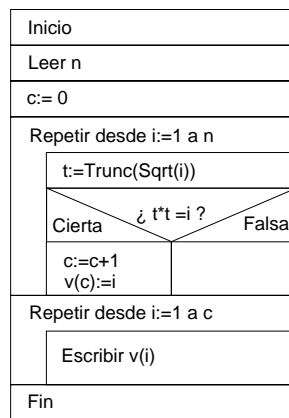


Figura 2.10 Ejemplo con los diagramas de Nasi/Shneiderman o de Chapin

Inconvenientes de los organigramas estructurados

El principal inconveniente es la laboriosidad que se necesita para realizar la descripción de los algoritmos, aunque existen herramientas gráficas para facilitar esta labor.

PSEUDOCODIGO

Otra forma de describir los algoritmos es mediante un pseudocódigo. Pseudo significa "falso", "imitación" y código se refiere a las instrucciones escritas en un lenguaje de programación. Entonces pseudocódigo no es realmente un código, sino una versión abreviada de las instrucciones que desempeñan las estructuras de control.

El pseudocódigo es una técnica para expresar en lenguaje natural la estructura de un programa.

El pseudocódigo no es un lenguaje de programación sino un modo de plantear un proceso de forma que su traducción a un lenguaje de alto nivel sea sencillo para un programador.

CONSTRUCCION DE PROGRAMAS

En general se utilizan traducciones al castellano de las estructuras de control, no existe una normativa estándar, cada programador sigue la suya propia. La *estructura de control secuencial* en pseudocódigo es la siguiente:

```
acción 1
acción 2
acción 3
```

La *estructura alternativa simple*:

```
SI (condición) ENTONCES
    acción
```

La *estructura alternativa doble*:

```
SI (condición) ENTONCES
    acción 1
SINO
    acción 2
```

La *estructura alternativa múltiple*:

```
CASO (selector) igual a
    1: acción 1
    2: acción 2
    ...
    n: acción n
```

La *estructura repetitiva FOR*:

```
PARA i:=1 HASTA n
    acción
```

La *estructura repetitiva WHILE*:

```
MIENTRAS (condición) HACER
    acción
```

La *estructura repetitiva REPEAT*:

```
REPETIR
    acción
HASTA (condición)
```

Para indicar que una instrucción afecta a un grupo de sentencias se colocan éstas alineadas entre sí, es decir se utiliza el interespaciado y los espacios en blanco para indicar los niveles de anidamiento de las instrucciones.

También se definen algunos tipos de acciones: *Leer* (para entrada de datos) y *Escribir* (para salida de resultados). Para asignar un valor o una expresión a una variable se usa el símbolo de asignación :=. El principio del algoritmo se indica con la palabra *INICIO*, y el final con la palabra *FIN*.

Ejemplo 2.7

Realizar el pseudocódigo del algoritmo de impresión de los cuadrados perfectos comprendidos entre 1 y N.

```

INICIO
Leer n
contador:=0
PARA i:=1 HASTA n
    t:=Trunc(Sqrt(i))
    SI (t*t = i)
        ENTONCES
            contador := contador+1
            v(contador) :=i
PARA i:=1 HASTA contador
    Escribir v(i)
FIN
    
```

NOTACION ALGORITMICA O LENGUAJE ALGORITMICO

Nace con la idea de aproximar más la descripción del algoritmo a la máquina, pero desvinculado de cualquier lenguaje de programación.

Cada autor utiliza su propia notación, aunque tienen características comunes. En este libro la notación algorítmica se irá definiendo en paralelo a la introducción de cada una de las estructuras de control, y estructuras de datos que se vayan explicando en los capítulos sucesivos.

La diferencia entre pseudocódigo y notación algorítmica es que el pseudocódigo se utilizará para descripciones muy generales del problema a resolver, mientras que la notación algorítmica profundizará en los algoritmos hasta el detalle. Aunque en los ejemplos y ejercicios de este capítulo el pseudocódigo tendrá mucho detalle, debido a que se resuelven problemas elementales.

Ejemplo 2.8

Se mostrará una notación algorítmica para la solución del problema de los cuadrados perfectos, ya explicado en el ejemplo 2.7. Este algoritmo lee un entero positivo N, e imprime la lista de cuadrados perfectos entre 1 y N. El vector V (con el índice contador) se usa para almacenar los cuadrados mientras se espera su impresión. T es una variable entera.

```

INICIO
1.[ENTRADA]
  Leer(N)
2.[SE CALCULA EL VECTOR CUADRADOS PERFECTOS.]
  contador := 0
  DESDE I := 1 HASTA N HACER
    2.1. [Se calcula el valor truncado de la raíz cuadrada del número en proceso]
      T := TRUNC (SQRT (I))
      2.2. [¿Es I un cuadrado perfecto?]
        SI T*T = I ENTONCES
          contador := contador + 1
          V(contador) := I
  FIN_SI
    
```

CONSTRUCCION DE PROGRAMAS

```
FIN_DESDE
3.[SE IMPRIMEN LOS NUMEROS CONTENIDOS EN EL VECTOR]
  DESDE I := 1 HASTA contador HACER
    Escribir (V(I))
  FIN_DESDE
4.[FINAL]
FIN      ^Z
```

Aquí FIN denota el final lógico del algoritmo, y ^Z el final físico.

2.6 CONSTRUCCION DE LA SOLUCION EN FORMA DE PROGRAMA

Dado que en las etapas anteriores la solución ya se ha definido, este paso es totalmente mecánico, solamente han de traducirse las reglas anteriormente descritas a un lenguaje de programación concreto.

De esta forma el programa ya nace con un estilo y una estructura muy determinada, y no como un conjunto de ideas agregadas desordenadamente.

El lenguaje Pascal desde su nacimiento trata de ser una notación algorítmica en lengua inglesa, de ahí que este paso se convierte en una traducción de la notación algorítmica en castellano al lenguaje Pascal en inglés.

Una vez que el lector tenga práctica en el diseño de algoritmos, no será necesario que especifique el pseudocódigo con tanta profundidad como se ha realizado en los ejemplos anteriores, para eso ya se escribe en Pascal directamente. Normalmente el pseudocódigo indica las líneas generales del algoritmo, prescindiendo de detalles muy concretos de la implementación.

Ejemplo 2.9

Codificación en lenguaje Pascal del algoritmo mostrado en los ejemplos 2.5, 2.6, 2.7 y 2.8.

```
PROGRAM CuadradosPerfectos (input, output);
CONST m = 100
TYPE indice = 1..m;
vector = ARRAY [indice] OF integer;
VAR t, contador: integer;
i: indice;
v: vector;
BEGIN
Write('Dame el valor de N');
Readln(n);
contador:=0;
FOR i:=1 TO n DO
BEGIN
t:=Trunc(Sqrt(i));
IF (t*t)=i
THEN
BEGIN
contador:=contador+1;
```

PRUEBA DE PROGRAMAS

```
        v[contador]:=i;
    END;
END;
FOR i:=1 TO contador DO
    Writeln('Cuadrado perfecto', i:3, '=', V [i]: 4);
END.
```

IMPLEMENTACION DE PROGRAMAS EN LENGUAJE PASCAL

A continuación se indican algunas recomendaciones para la implementación de programas en general, y en particular para el caso del lenguaje Pascal.

- § *Presentación del programa.* El lenguaje Pascal es bastante claro en sí para mostrar la estructura de los programas, pero puede mejorarse la legibilidad y claridad de éstos con el uso de comentarios y de interespaciado. La presentación de programas puede jugar un papel importante de cara a evitar errores o para subsanarlos.
- § *Uso de comentarios.* Los comentarios deben servir para explicar las intenciones y supuestos de una sección del programa. Los comentarios aclaran el programa, pero no pueden mejorar un programa muy mal estructurado. El uso de comentarios en exceso puede oscurecer el programa. A veces se cambia el código del programa y se olvida cambiar los comentarios; hay que tener cuidado con esto.
- § *Uso de interespaciados.* El interespaciado, es decir, el uso juicioso de espacios en blanco, es otro elemento importante en la presentación de programas.
- § *Uso de indentaciones.* La indentación (uso de márgenes escalonados) es conveniente cuando se emplea anidamiento de sentencias y cláusulas compuestas. Es importante alinear cada sentencia BEGIN con su correspondiente END.
- § *Uso de variables.* El lenguaje Pascal ofrece mucha flexibilidad al elegir los nombres de las variables, para que los programas estén autodocumentados tanto como sea posible. Es decir los nombres de las variables han de escogerse de forma que reflejen el propósito de la variable en el programa, lo que representa una gran influencia en la legibilidad del código y también facilita la labor al realizar modificaciones.

2.7 PRUEBA DE PROGRAMAS

En esta fase se comprueba si el programa está correctamente instalado en la máquina. No todo se deja para probar en esta etapa, sino que hasta este momento los programas se habían probado en la mente de cada programador, simulando mentalmente la correcta ejecución de cada módulo del programa.

CONSTRUCCION DE PROGRAMAS

La prueba de programas (figura 2.11) comienza por su escritura en un editor en el ordenador, posteriormente se compilan, pudiendo darse errores de compilación debidos a errores sintácticos (por ejemplo falta un ;) o semánticos (asignación entre variables de tipos no compatibles). Entonces se debe de volver al editor para subsanarlos. Una vez que el compilador no indica errores, se puede ejecutar el programa. También pueden ocurrir errores de ejecución (por ejemplo una división por cero, o la raíz cuadrada de un número negativo), pero los errores de ejecución habituales son debidos a que el programa no resuelve adecuadamente el problema propuesto. En estos casos ha de revisarse el programa, y corregirlo con el editor. Los compiladores comerciales (por ejemplo Turbo Pascal) suelen incorporar opciones de depuración (*debug*) en el entorno integrado de desarrollo (IDE). El depurador permite la visualización paso a paso de la ejecución del programa, visualizándose simultáneamente la instrucción que se ejecuta, el valor de las variables, y el resultado de la ejecución.

La prueba de un programa nunca es sencilla, pues aunque puede probar la presencia de errores, nunca se puede demostrar la ausencia de ellos.

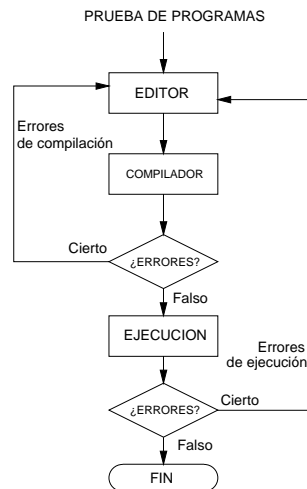


Figura 2.11 Prueba de programas

Una prueba con éxito, sólo significa que no se detectaron errores bajo las circunstancias especiales de dicha prueba, esto no significa nada frente a otras circunstancias. En teoría, la única manera en que las pruebas pueden demostrar que un programa es correcto es que examinen todos los casos posibles (esta prueba recibe el nombre de prueba exhaustiva). Una prueba exhaustiva debe tener en cuenta todas las combinaciones posibles que se pueden dar en un programa. Esto es técnicamente imposible en la mayoría de los programas.

DOCUMENTACION DE PROGRAMAS

La frase anterior no significa que las pruebas sean inútiles. Se puede reducir mucho el número de casos a probar a partir del número requerido por una prueba exhaustiva. Se puede hacer una prueba razonable con un número relativamente pequeño de casos bien elegidos.

Para realizar las pruebas hay que olvidarse de como es la estructura del programa, y volver al primer paso donde se nos indican las especificaciones del programa.

El proceso de prueba es un proceso creativo, en el cual hay que buscarle las "*cosquillas*" al programa o a la aplicación informática. En muchas ocasiones se realiza por personas diferentes a las del equipo de desarrollo de la aplicación.

2.8 DOCUMENTACION DE PROGRAMAS

Toda aplicación informática tiene una documentación. La documentación puede clasificarse en manual del usuario y manual de implementación.

El manual del usuario incluye toda la información sobre: *requerimientos de hardware y software de la aplicación, instalación de la aplicación, y manejo de la aplicación.*

El manual de implementación contiene toda la información necesaria para desarrollarla y mantener la aplicación informática. Los documentos que componen este manual son: *definición de requisitos, análisis, diseño, algoritmos de cada módulo, código de los programas, pruebas realizadas, e historia del mantenimiento de la aplicación.*

2.9 MANTENIMIENTO DE PROGRAMAS

Cuando un estudiante programa, muy rara vez realiza mantenimiento de programas. Sin embargo esta etapa ocupa del orden del 50% o más del tiempo de un programador en la vida real. Es clásica la representación gráfica de la figura 2.12.

El mantenimiento de un programa no se refiere a la reparación o cambio de las partes deterioradas como sucede con el hardware, sino a las modificaciones que deben hacerse a los defectos del diseño, lo cual suele suponer el desarrollo de nuevas funciones para cubrir necesidades adicionales que surgen.

El mantenimiento es inevitable, ya que los usuarios siempre piden más a un programa. Por lo tanto los programas deben de realizarse lo más abiertos posibles de cara a un futuro mantenimiento. También es muy importante la claridad.

En la figura 2.13 se muestra el ciclo de vida de una aplicación informática, donde se muestran las relaciones entre las distintas fases, y se observa que es un ciclo sin fin.

CONSTRUCCION DE PROGRAMAS

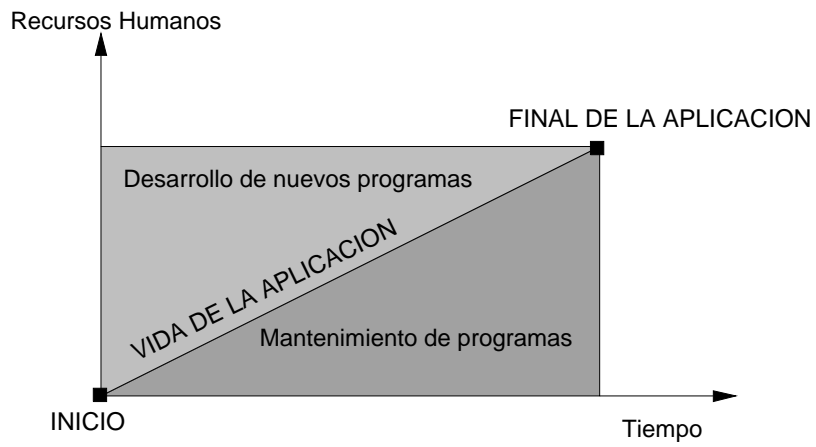


Figura 2.12 Relación entre desarrollo y mantenimiento en la vida de una aplicación informática

CICLO DE VIDA DEL SOFTWARE

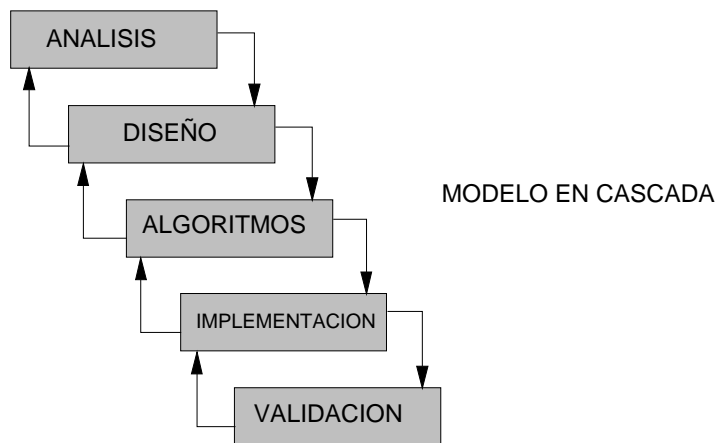


Figura 2.13 Ciclo de vida de una aplicación informática

EJERCICIOS RESUELTOS

2.10 EJERCICIOS RESUELTOS

2.1 Escribir el pseudocódigo de un algoritmo que determine el mayor de dos números.

```
INICIO
Leer a, b
SI a>b ENTONCES
    Escribir a, 'Es el mayor'
SINO
    Escribir b, 'Es el mayor'
FIN
```

2.2 Escribir el pseudocódigo de un algoritmo que determine el mayor de tres números.

```
INICIO
Leer a,b,c
mayor:=a
SI b>mayor ENTONCES
    mayor:=b
SI c>mayor ENTONCES
    mayor:=c
Escribir mayor
FIN
```

2.3 Escribir el pseudocódigo de un algoritmo que determine el mayor de n números.

```
INICIO
Leer n
Leer x
mayor:=x
PARA i:=2 HASTA n
    Leer x
    SI x>mayor ENTONCES mayor:=x
Escribir mayor
FIN
```

2.4 Escribir el pseudocódigo de un algoritmo que determine el máximo y el mínimo de un número de valores no conocido *a priori*, se introducirá -999 para indicar que se finaliza la introducción de datos.

```
INICIO
Leer x
maximo:=x
minimo:=x
MIENTRAS x<>-999 HACER
    Leer x
    SI x>maximo ENTONCES maximo:=x
    SI x<minimo ENTONCES minimo:=x
Escribir maximo, minimo
FIN
```

- 2.5** Escribir el mismo algoritmo del ejercicio 2.4, pero utilizando la estructura repetitiva REPEAT.

```

INICIO
maximo:=-10000
minimo:=10000
REPETIR
    Leer x
    SI x>maximo ENTONCES maximo:=x
    SI x<minimo ENTONCES minimo:=x
HASTA x=-999
Escribir maximo, minimo
FIN

```

2.11 EJERCICIOS PROPUESTOS

- 2.6** Se desea construir un algoritmo que determine los números primos entre 1 y un valor N leído por teclado. Escribir el diseño descendente, el diagrama de flujo, el diagrama estructurado de Chapin, y el pseudocódigo.
- 2.7** Se desea escribir un algoritmo que genere los n primeros términos de la sucesión de Fibonacci. La sucesión de Fibonacci es 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,... el término general se define como $F_{n+2}=F_{n+1}+F_n$ para $n \geq 0$, con $F_0=0$, y $F_1=1$. Escribir el diseño descendente y el pseudocódigo.
- 2.8** Escribir de tres formas diferentes el pseudocódigo de un algoritmo que determine el factorial de un número. Pueden usarse las tres estructuras de control repetitivas.
- 2.9** Escribir el pseudocódigo de un algoritmo que determine el valor medio de un conjunto de n números.
- 2.10** Modificar el algoritmo del ejercicio 2.9, para cuando n no se conoce *a priori*.
- 2.11** Modificar el algoritmo del ejercicio 2.10, para que además de la media calcule la desviación típica.
- 2.12** Escribir un algoritmo que escriba una tabla de multiplicar.
- 2.13** Escribir el pseudocódigo de un algoritmo que escriba los meses del año en función de una variable selector que es el número del mes.

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

- 2.14** Escribir el pseudocódigo de un algoritmo que resuelva ecuaciones de segundo grado.
- 2.15** Escribir el pseudocódigo de un algoritmo que dados n pares de coordenadas cartesianas (x,y) , determine qué puntos están dentro del círculo limitado por la circunferencia de ecuación $x^2+y^2=5$.
- 2.16** Realizar el análisis orientado a objetos del problema de calcular los costes de consumo de gasolina de distintos tipos de vehículos (automóviles, camiones, furgonetas, y autobuses).

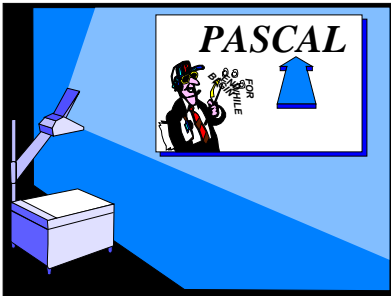
2.12 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Sobre metodologías de diseño e ingeniería del software existen varias obras generales: *Ingeniería del Software, un enfoque práctico* de R.S. Pressman (McGraw-Hill, 1993); e *Ingeniería del Software* de I. Sommerville (Addison-Wesley Iberoamericana, 1988).

Para profundizar sobre el diseño descendente puede consultarse el libro de N. Wirth, titulado *Introducción a la programación sistemática*, publicado en castellano por la editorial *El Ateneo* (1984).

Para una introducción al diseño orientado a objetos puede consultarse la obra *Software orientado a objetos* de A.L. Winblad, S.D. Edwards, D.R. Kingn (Addison-Wesley/Díaz de Santos, 1993). Para una profundización consultar *Object Oriented Design with applications* de G. Booch (Benjamin Cummings, 1991).

La primera referencia contra el uso indiscriminado de las instrucciones de salto incondicional (GOTO), fue el artículo de E.W. Dijkstra titulado "*Goto statement considered harmful*" publicado en la revista *Communications of the ACM* (Vol 11, núm. 3, marzo 1968). El concepto de programación estructurada fue desarrollado en el libro titulado *Structured Programming*, de los autores O.J. Dahl, E.W. Dijkstra, y C.A.R. Hoare, publicado por la editorial *Academic Press* en 1972.



CAPITULO 3

INTRODUCCION AL LENGUAJE PASCAL

CONTENIDOS

- 3.1 El lenguaje Pascal
- 3.2 Los datos y sus tipos
- 3.3 Identificadores, constantes y variables
- 3.4 Definiciones y declaraciones
- 3.5 La sentencia de asignación
- 3.6 Características de Turbo Pascal
- 3.7 Palabras reservadas de Turbo Pascal
- 3.8 Directivas estándar de turbo Pascal
- 3.9 Cuestiones resueltas
- 3.10 Cuestiones propuestas
- 3.11 Ampliaciones y notas bibliográficas

3.1 EL LENGUAJE PASCAL

El lenguaje de programación Pascal fue diseñado por Niklaus Wirth en 1968, en la Universidad Técnica de Zurich (Suiza). Deriva del lenguaje ALGOL-60. Su nombre se debe al matemático francés Blaise Pascal (1623-1662), inventor de la primera máquina de calcular mecánica de la Historia.

La idea inicial de Wirth fue desarrollar un lenguaje de alto nivel muy disciplinado, para enseñar programación estructurada. Sin embargo hoy en día, se usa tanto como lenguaje de enseñanza, como lenguaje de propósito general para una gran variedad de aplicaciones. El uso del Pascal va en aumento debido principalmente a las siguientes características:

- **claridad:** Los programas escritos en Pascal son fáciles de comprender, debido al énfasis particular que impone su programación. De esta forma se facilita la labor del programador para entender los programas realizados por otras personas, o por él mismo al cabo de un período largo de tiempo. Esta característica también contribuye a una mayor facilidad en la depuración de los errores.
- **modularidad:** En el lenguaje Pascal la mayor parte de las tareas se dividen en módulos separados, que constituyen los procedimientos o funciones como se verá en el capítulo 7. El uso de una estructura modular potencia la precisión y claridad de un programa, facilitando las futuras modificaciones del mismo.

3.2 LOS DATOS Y SUS TIPOS

Los programas de ordenador realizan operaciones con distintos tipos de datos. Un tipo de datos es el conjunto de valores que puede tomar una constante, variable o expresión. Una de las características más importantes e interesantes del lenguaje Pascal es la capacidad que tiene para admitir muchos tipos diferentes de datos. A continuación se presenta un resumen de los distintos tipos de datos.

• DATOS ESTATICOS

(1) Datos de tipo simple

(I) Tipos de datos predefinidos o estándar:

- a) INTEGER (entero)
- b) REAL (real)
- c) CHAR (literal o carácter)
- d) BOOLEAN (lógico o booleano)

(II) Tipos de dato definidos por el usuario:

- a) enumerado
- b) subrango

(2) Datos de tipo estructurado

- (I) ARRAY (Matrices)
- (II) RECORD (Registros)
- (III) FILE (Ficheros)
- (IV) SET (Conjuntos)

• DATOS DINAMICOS

Se almacenan en estructuras dinámicas de datos (listas, pilas, colas, árboles, anillos, grafos,...) realizadas con punteros. En estas estructuras, los elementos se van creando y eliminando en tiempo de ejecución, por tanto, la memoria ocupada por datos dinámicos es gestionada en tiempo de ejecución a diferencia de los datos estáticos cuya memoria se gestiona en tiempo de compilación. Las estructuras dinámicas de datos serán estudiadas en profundidad en el capítulo 12.

• TIPOS ABSTRACTOS DE DATOS

Son un modelo matemático sobre el cual se pueden efectuar diversas operaciones definidas por el programador. Es decir, operaciones definidas por una estructura de datos y procesos que pueden utilizar esas operaciones de forma coordinada para que no interfieran entre sí. Esto facilita la *abstracción*, la *ocultación* y el *encapsulamiento* de datos como se verá en el capítulo 13.

DATOS DE TIPO SIMPLE PREDEFINIDOS

Los datos de tipo simple son elementos individuales (números, caracteres ...). El lenguaje Pascal tiene cuatro tipos simples predefinidos, que se muestran a continuación:

Tipo INTEGER

El tipo *INTEGER* está constituido por los números enteros.

Ejemplo 3.1

+333 -45 +1 35 60

Cuando se omite el signo se asume que el entero es positivo.

El valor máximo y mínimo de un número entero depende del hardware de la máquina y del compilador. El valor máximo en las microcomputadoras es +32767 y el mínimo es -32768, que corresponde a los valores que pueden ser memorizados en una palabra de memoria de 16 bits ($2^n - 1$ donde $n=16$). Se necesitan dos bytes para almacenar un número entero.

Tipo REAL

El tipo *REAL* está constituido por los números reales racionales. En el lenguaje Pascal los números reales deben contener un punto decimal o un exponente (o ambos). Si se incluye el punto decimal, éste debe aparecer entre dos dígitos. En consecuencia, un número real no puede comenzar o acabar con un punto decimal.

Ejemplo 3.2

Números reales válidos como tipo *REAL*.

99.9E+8	0.066E14	0.000	7.234	-0.1	+23.2
+0.1234567E12	2E-9	5E+8	4E10	3E3	-1E-1

Ejemplo 3.3

Números reales no válidos como tipo *REAL*, por las razones que se exponen:

- | | |
|----------|---|
| 35. | Debe existir un dígito a cada lado del punto decimal ⁷ . |
| 1.000,05 | No se admiten comas. |
| .6666666 | Debe existir un dígito a cada lado del punto decimal. |
| 100 | Debe haber un punto decimal o un exponente. |
| 903.E+11 | Debe existir un dígito a cada lado del punto decimal. |
| 8E2.77 | El exponente debe ser entero. |
| 9999.E-9 | Debe existir un dígito a cada lado del punto decimal. |
| 3E 10 | No se permiten espacios en blanco entre la E y los dígitos del exponente. |

Los números reales tienen un rango mucho mayor que los números enteros. Dicho rango no depende del lenguaje Pascal sino del ordenador y del compilador, en los microordenadores oscila desde un valor de 1E-38 hasta 1E+38. El número de cifras significativas puede variar de unas versiones a otras del compilador.

El límite de precisión de los números reales hace que los errores de redondeo puedan afectar a la exactitud de los resultados:

$$1/3 + 1/3 + 1/3 = 0.33333333 + 0.33333333 + 0.33333333 = 0.99999999$$

en vez de 1 resulta 0.99999999 .

Tipo CHAR

El tipo de datos *CHAR* o carácter describe los caracteres simples encerrados entre apóstrofes.

⁷ El compilador Turbo Pascal tiene una extensión al Pascal estándar, que permite constantes reales con un punto decimal a su comienzo o a su final.

Ejemplo 3.4

Datos de tipo *CHAR*.

'r' 'q' '&' '§' '%' '!' 'a' 'b' 'B'

El tipo *CHAR* solo admite un carácter.

CADENAS DE CARACTERES

Una cadena de caracteres es una sucesión de caracteres simples encerrados entre apóstrofes:

'lalala' 'Pedro' 'MARIA'

El número máximo de caracteres que pueden incluirse en una cadena varía de una versión de Pascal a otra. La mayoría de las versiones permiten longitudes máximas de al menos 255 caracteres, suficientes para la mayoría de las necesidades.

Si una cadena incluye un apóstrofe, éste debe escribirse dos veces. Sin embargo sólo aparecerá una vez cuando la cadena se visualice o se imprima. Por tanto un sólo apóstrofe se interpreta por el lenguaje como delimitador de cadena, mientras que un apóstrofe repetido se interpreta como un único apóstrofe dentro de la cadena.

Ejemplo 3.5

'L' 'equipe' 'Pedro' 's'

El compilador de Pascal necesita los apóstrofes para diferenciar entre el entero 8 y el char '8', o entre el real 3.33 y la cadena '3.33'.

Veasé la sección correspondiente a la representación de la información del capítulo 1 para mayor información sobre el conjunto de caracteres alfanuméricos diferentes que posee cada máquina.

El tipo BOOLEAN

El tipo *BOOLEAN* o lógico es un tipo con sólo dos valores *TRUE* (cierto) y *FALSE* (falso). El nombre de este tipo de datos se debe al matemático inglés George Boole (1815-1864). Los datos de tipo *BOOLEAN* no pueden leerse, pero si imprimirse.

El resto de los tipos de datos se irán introduciendo en los sucesivos capítulos. Tan sólo indicar que los datos de tipo estructurado se componen de múltiples elementos relacionados entre sí de alguna manera especificada.

3.3 IDENTIFICADORES, CONSTANTES Y VARIABLES

Un *identificador* es un nombre dado a un elemento de un programa (nombre de una variable, una constante, ...). Los identificadores están formados por letras o dígitos en cualquier orden, pero el primer carácter debe ser una letra como se verá más adelante. *Se permiten tanto letras mayúsculas como minúsculas*, que se consideran indistinguibles. Aprovechando esta posibilidad, existen unas normas para la escritura de los diferentes identificadores que, aunque no son obligatorias, son aceptadas por la mayoría de los programadores puesto que facilitan la lectura de los programas, y serán aplicadas a lo largo de este texto.

Normativa para la escritura de identificadores⁸

- Las palabras reservadas del lenguaje (tabla 3.1) se escribirán en mayúsculas.

Ejemplo 3.6

```
BEGIN  END
```

- Los identificadores de tipos, constantes y variables se escribirán en minúsculas; y caso de que formasen una palabra compuesta, se intercalarán letras mayúsculas para mejorar su legibilidad.

Ejemplo 3.7

```
nombreCompuesto
```

- El nombre del programa y los nombres de procedimientos y funciones (subprogramas), se escribirán también en minúsculas pero comenzando por letra mayúscula.

Ejemplos 3.8

```
Write
FuncionMagica
NombreDelPrograma
```

Obsérvese que en los ejemplos anteriores no se han utilizado acentos ni las letras ñ y Ñ. Únicamente se admiten las letras del alfabeto inglés.

Las **palabras reservadas** no pueden utilizarse como identificadores y tienen un propósito específico en el lenguaje de programación, tal y como se irá viendo a lo largo de los capítulos siguientes (tabla 3.1).

⁸ El entorno integrado de desarrollo (IDE) del compilador Turbo Pascal, facilita la tarea al resaltar con diferentes colores: los identificadores, las palabras reservadas, los comentarios, las cadenas de caracteres, etc...

Un identificador puede ser arbitrariamente largo, aunque en la mayoría de las implementaciones del Pascal sólo se reconocen los 8 primeros caracteres. Un identificador debe tener un número de caracteres suficiente para indicar su significado. Por otra parte debe evitarse un número excesivo de caracteres.

and	end	nil	set
array	file	not	then
begin	for	of	to
case	function	or	type
const	goto	packed	until
div	if	procedure	var
do	in	program	while
downto	label	record	with
else	mod	repeat	

Tabla 3.1 Palabras reservadas del lenguaje Pascal

METALENGUAJES

Son herramientas útiles para la descripción formal de la sintaxis de un lenguaje de programación, facilitando así la comprensión del mismo. Se estudiarán dos tipos: los **diagramas sintácticos** y la **notación EBNF** (*Extended Backus Naur Form*).

Diagramas Sintácticos

Constan de una serie de cajas o símbolos geométricos conectados por flechas donde se introducen los símbolos del lenguaje que se dividen en:

- **Símbolos terminales:** Son los que forman las sentencias del lenguaje y se introducen dentro de círculos o cajas de bordes redondeados.
- **Símbolos no terminales:** Son introducidos como elementos auxiliares y no figuran en las sentencias del lenguaje. Se representan por su nombre encerrado en un rectángulo o cuadrado.

Notación EBNF

Utiliza los siguientes metasímbolos:

< > encierra conceptos definidos o por definir. Se utiliza para los símbolos no terminales.

::= sirve para definir o indicar equivalencia.

| separa las distintas alternativas.

IDENTIFICADORES, CONSTANTES Y VARIABLES

{ } indica que lo que aparece entre llaves puede repetirse cero o más veces. En algunos casos se indica con subíndices y superíndices el intervalo de repeticiones.

" " indica que el metасímbolo que aparece entre comillas es un caracter que forma parte de la sintaxis del lenguaje.

() se permite el uso de paréntesis para hacer agrupaciones.

Se comenzará viendo la notación *EBNF* y el diagrama sintáctico de un *IDENTIFICADOR*.

```
<identificador> ::= <letra> { <alfanumérico> }
<alfanumérico> ::= <letra> | <dígito>
<letra>         ::= a | b | c | ... | y | z | A | B | ... | Y | Z
<dígito>       ::= 0 | 1 | 2 | ... | 8 | 9
```

El diagrama siguiente indica que un identificador debe comenzar por una letra y puede estar seguido indistintamente por una letra o un dígito.

Ejemplos 3.9

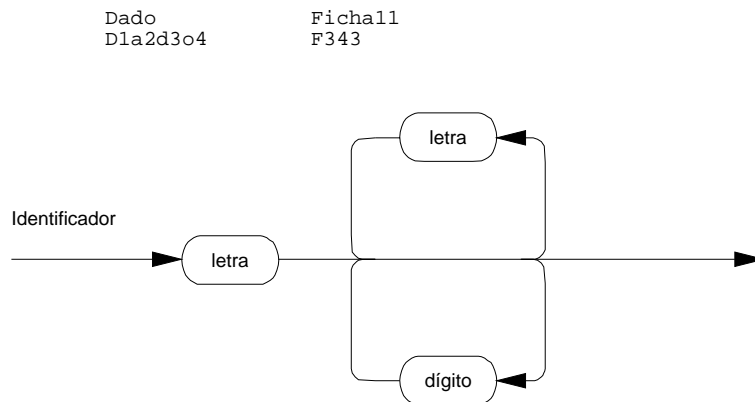


Figura 3.1 Diagrama sintáctico de un identificador

Los datos se almacenan en la memoria del ordenador. La memoria del ordenador puede representarse como un gran casillero donde se almacenan los datos. Evidentemente las casillas no son iguales pues cada tipo de dato ocupa distinta memoria. Cada una de las casillas está localizada por su posición. Los programadores en lenguaje máquina indican estas posiciones con un código, por ejemplo, de la forma 1011100110. Los lenguajes de alto nivel ofrecen la posibilidad de usar identificadores para señalar la posición de memoria que ocupan los datos.

Los identificadores pueden ser nombres de variables o de constantes, es decir, pueden indicar localizaciones de memoria cuyo valor puede cambiarse o debe de permanecer constante.

Variable es un identificador cuyo valor puede cambiar a lo largo de la ejecución de un programa.

Constante es un identificador cuyo valor **no** puede variar a lo largo del programa.

3.4 DEFINICIONES Y DECLARACIONES

Las constantes deben estar **definidas** antes de que aparezcan en una sentencia de Pascal. Esta definición cubre dos propósitos: establece que el identificador es una constante y asocia un valor a la constante. El tipo de constante viene determinado implícitamente por el dato asignado (real, entero, ...).

La forma de definición de constantes es la siguiente:

```
CONST
    nombre = valor;
```

donde:

nombre es un identificador que representa el nombre de la constante.
 valor es el dato efectivo que se asigna al nombre.

Ejemplo 3.10

```
CONST
    fraccion = 0.6666667;
    ndatos = 100;
    letra = 'h';
```

La sintaxis en notación *EBNF* es la siguiente:

```
<parte definición de constante> ::= CONST <definición de constante>
                                   { ; <definición de constante> } ;
<definición de constante> ::= <identificador> = <constante>
<constante> ::= <número sin signo> | <signo> <número sin signo> | <identificador de constante> | <signo> <identificador de constante> | <cadena de caracteres>
```

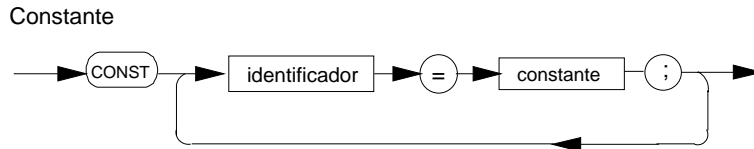


Figura 3.2 Diagrama sintáctico para definir constantes

Las definiciones de las constantes deben preceder a la declaración de las variables.

Cada variable debe ser **declarada** antes de que aparezca en el programa. La declaración de variable indica al compilador que el identificador es una variable y especifica el tipo de la variable. A diferencia de la definición de constante no se le asocia ningún dato concreto a la variable.

La forma general de declaración de una variable es:

```
VAR
    nombre : tipo;
```

LA SENTENCIA DE ASIGNACION

o si hay varias variables del mismo tipo

```
VAR
    nombre1, nombre2, ... , nombre-n : tipo
```

donde *nombre*, *nombre1*, *nombre2*, ... , *nombre-n* son identificadores que representan nombres de variables individuales y *tipo* es el tipo de datos de las variables.

Ejemplo 3.11

```
VAR
    ini, fini : integer;
    sueldo : real;
    piropi : char;
    decide : boolean;
    notaPasc : real;
    notaFort : real;
    nombre, apelli1, apelli2 : char;
    estoyAprobadoEnPascal : boolean;
```

La sintaxis en notación *EBNF* es la siguiente:

```
<parte de declaración de variables> ::= VAR <declaración de variables>{ ;
<declaración de variable> ::= <identificador>{ , <identifica-
dor> } : <tipo>
```

Variable

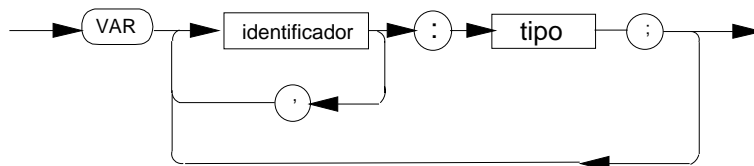


Figura 3.3 Diagrama sintáctico para declarar variables

3.5 LA SENTENCIA DE ASIGNACION

La asignación inicial o el cambio de valor de una variable se realiza mediante una sentencia de asignación.

En lenguaje Pascal todos los identificadores deben declararse o definirse antes de usarse, por lo tanto antes de utilizar las sentencias de asignación, las variables y constantes deben estar definidas.

Ejemplo 3.12

```
CONST
    pi = 3.14159;
    verdadero = true;
    casos = 27;
    nombre = 'N';
```

```
VAR
    a, e : integer;
    b, f : boolean;
    c, g : real;
    d, h : char ;
```

Si se declaran las variables y constantes anteriores las siguientes asignaciones son válidas:

```
a := 1000;
b := true;
c := 9.99;
e := casos;
f := verdadero;
g := pi;
h := nombre;
```

Teniendo en cuenta las definiciones y declaraciones anteriores las sentencias de asignación siguientes no son correctas:

```
a := 9.999;    La variable a es entera y 9.999 es un real
f := 7654;     La variable f es booleana y 7654 es un entero
k := 4444;     k no está definido o declarado
```

La sintaxis en notación *EBNF* es la siguiente:

<sentencia de asignación> ::= <variable> := <expresión>

Expresión



Figura 3.4 Diagrama sintáctico de la sentencia de asignación

El símbolo de asignación es := y se puede leer como *toma*. Entonces el diagrama sintáctico puede leerse como: *la variable toma el valor de la expresión*.

Una expresión es una serie de operandos (números, constantes, variables) enlazados por operadores, que serán numéricos o booleanos dando lugar a expresiones numéricas o expresiones booleanas. Una expresión numérica representa un valor numérico, mientras que una expresión booleana representa un valor booleano.

Las expresiones booleanas también se llaman expresiones lógicas.

Ejemplo 3.13

Sea la expresión numérica:

$$3 * (a - b + 7 * c) / a * b * c$$

CARACTERISTICAS DE TURBO PASCAL

donde *a*, *b*, y *c* son identificadores. A los números 3 y 7 se les llama operandos, y los símbolos +, -, *, y / son operadores, que representan adición, sustracción, multiplicación y división, respectivamente. Esta expresión completa representa un número, así si *a*, *b*, y *c* valen 1, la expresión vale 21.

Ejemplo 3.14

```
edad = 30
```

Para que esta expresión booleana sea correcta la variable *edad* debe ser de tipo entero, pues 30 es un número entero. *edad* y 30 son los operandos y = es un operador lógico.

En los capítulos siguientes se especificarán todos los tipos de operadores lógicos y numéricos.

3.6 CARACTERISTICAS DE TURBO PASCAL

- Un identificador puede tener cualquier longitud pero solamente son significativos 63 caracteres y puede comenzar por letra o el carácter de subrayado (_).
- Se pueden utilizar las *constantes con tipo* que a diferencia de las *constantes sin tipo*, especifican el tipo y el valor de las constantes.

Las constantes con tipo pueden utilizarse igual que las variables del mismo tipo, y pueden aparecer a la izquierda de una sentencia de asignación.

En el ejemplo siguiente se puede ver un fragmento de programa que utiliza constantes con tipo.

Ejemplo 3.15

```
...
VAR      mayor:integer;
         menor:real;
CONST   minimo:integer = 2;
         maximo:integer = 10;
...
```

Toda constante con tipo es una variable con un valor constante y no se puede intercambiar con constantes ordinarias.

- Las definiciones de constantes no deben preceder *obligatoriamente* a la declaración de las variables.
- El entorno integrado de desarrollo (IDE) de Turbo Pascal, usa diferentes colores para distinguir los identificadores, palabras reservadas, comentarios, cadenas, etc...

3.7 PALABRAS RESERVADAS DE TURBO PASCAL

Además de las palabras reservadas citadas en la tabla 3.1, incluye las siguientes:

asm	inherited	shl
constructor	inline	shr
destructor	interface	string
implementation	object	unit

3.8 DIRECTIVAS ESTANDAR DE TURBO PASCAL

Las directivas indican al compilador cómo debe comportarse ante ciertas situaciones. Turbo Pascal incluye las siguientes:

absolute	forward	public
assembler	interrupt	virtual
external	near	
far	private	

Public y *private* actúan como directivas excepto cuando aparecen en declaraciones de tipos objetos, como se verá en el capítulo 13.

3.9 CUESTIONES RESUELTAS

3.1 Determinar cuáles de las siguientes sentencias de asignación son correctas si todas las variables que aparecen son del tipo real.

```
r1 := 1;
r2 := r1 + 3;
r3 := r1 + r2;
```

Solución

Son todas correctas.

3.2 Dependiendo del código utilizado, ¿Qué diferencia al carácter 'A'?

Solución

El número decimal asociado que en el código ASCII es el 65 y en el EBCDIC es el 193.

CUESTIONES PROPUESTAS

3.10 CUESTIONES PROPUESTAS

3.3 Determinar si los siguientes identificadores son válidos.

puñeta a12 21.a alfa+1

3.4 Determinar si las siguientes asignaciones son correctas, siendo las variables de tipo integer.

```
y:=1;  
i:='1';  
calculo:=y;  
valor:=4.5;
```

3.5 ¿En qué se diferencia el tipo integer del tipo real?

3.6 Enumerar los distintos tipos de datos del lenguaje Pascal.

3.7 Indicar las diferencias entre las constantes y las variables.

3.8 Determinar si las siguientes sentencias de asignación son correctas teniendo en cuenta las declaraciones:

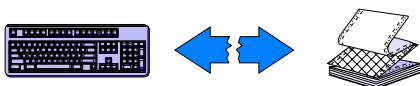
```
...  
VAR  
  a,b:char;  
  i,j,k:integre;  
  z:real;  
...  
a:='1';  
i:=33;  
j:=a;  
b:='cd';  
...
```

3.9 ¿ Cuantos caracteres se incluyen dentro del código ASCII ?

3.10 ¿ Existe semejanza entre el tipo char y el tipo boolean ?

3.11 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

En los anexos II y III se encuentra la sintaxis completa del lenguaje Pascal estándar descrita en notación EBNF y mediante diagramas sintácticos. Si se quiere conocer la normativa estándar del lenguaje en profundidad, el libro indicado es *User manual and report ISO Pascal Standard* de Kathleen Jensen, Niklaus Wirth (Springer-Verlang, 4ª edición 1991). Existe una traducción al castellano de la segunda edición en la *Editorial Ateneo* (Buenos Aires, 1984), con el título *PASCAL: Manual del usuario e informe*.



CAPITULO 4

ENTRADA / SALIDA

CONTENIDOS

- 4.1. Entrada estándar
- 4.2. Salida estándar
- 4.3. Estructura completa de un programa
- 4.4. Ejercicios resueltos
- 4.5. Extensiones del compilador Turbo Pascal
- 4.6. Ejercicios propuestos

4.1 ENTRADA ESTANDAR

Los programas suelen necesitar datos de entrada, con los cuales operan y producen los resultados. La entrada de datos puede hacerse de múltiples maneras, siendo las más corrientes desde el teclado o desde un fichero en disco o en cinta. Los ficheros se presentan en el capítulo 11.

Mientras tanto, en todos los ejemplos y programas que se realicen, se supone que los datos de entrada se introducen por el teclado del ordenador, y que esta es la única comunicación del programa con el exterior en cuanto a lectura.

ENTRADA ESTANDAR

Las sentencias⁹ de lectura del lenguaje Pascal son *Read* y *Readln*.

La sentencia *Read* se utiliza para leer datos de un fichero de entrada y asignarlos a variables de un tipo determinado declarado anteriormente. El fichero estándar de entrada se denomina *INPUT* y normalmente está asignado al teclado. No todos los tipos de variables del lenguaje Pascal pueden leerse mediante esta sentencia. Por ejemplo, las variables de tipo booleano no pueden incluirse en la lista de variables de entrada.

Las sentencias *Read* y *Readln* tienen el siguiente diagrama sintáctico:

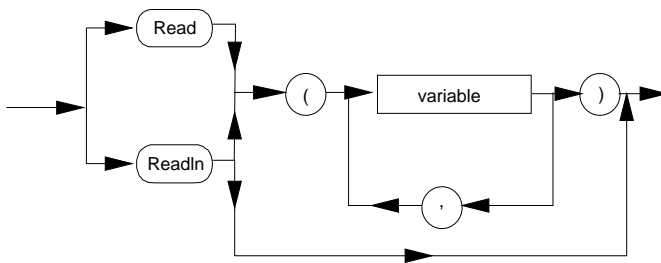


Figura 4.1 Diagrama sintáctico de *Read* y *Readln*

las variables se representan por sus *identificadores*. Los datos que se introduzcan tienen que coincidir con el tipo de las variables.

La sintaxis de estas sentencias en notación EBNF es:

```
<sentencia Read> ::= Read ( <variable> { , <variable> } )  
<sentencia Readln> ::= Readln <argumentos>  
<argumentos> ::= <vacío> | ( <variable> { , <variable> } )
```

La sentencia *Read* se encarga de leer el dato de un fichero y de asignarlo a la variable que se indica. Hasta que se estudien los ficheros, la entrada se realizará desde *INPUT*, fichero de entrada estándar, generalmente el teclado.

Ejemplo 4.1

```
Read (nCasos, nDias, mes );
```

Aquí la sentencia *Read* lee tres variables, que deben estar declaradas anteriormente con un determinado tipo, que no puede ser booleano. Por el teclado se deben de introducir los valores correspondientes a dichas variables separados por uno o más espacios en blanco. La variable *nDias* no tiene acentuada la *i* dado que no se pueden usar acentos en los identificadores.

⁹ Estrictamente son *procedimientos estándar de entrada y salida*, pero dado que el concepto de procedimiento no se introduce hasta el capítulo 7, se mantendrá hasta ese capítulo la notación de sentencia.

Ejemplo 4.2

Si se declaran las variables de la siguiente forma:

```
VAR
  nAprobados, nSuspendos, nPresentados, dia, mes, anio: integer;
```

entonces las sentencias de lectura se ejecutan así :

Sentencia	Datos	Contenidos en memoria después de leer
Read (nAprobados)	50	nAprobados:=50
Read (dia,mes,anio)	2 10 1989	dia :=2 mes :=10 anio:=1989
Read (nSuspendos)	0.5432	!!! ERROR !!! ¹⁰ La variable es de tipo entero, y se ha introducido un número real.
Read (nPresentados)	150	nPresentados:=150
Read (dia,mes,anio)	20 10 1989	dia:=20 mes:=10 anio:=1989

Tal como se observa en el ejemplo anterior el programa lee los valores de las tres variables, tanto si están en la misma línea como si no.

Los datos de entrada del fichero *input* están organizados en líneas. Desde un punto de vista lógico, cada línea está separada de la siguiente por un carácter especial (o grupo de caracteres) denominado *marca de fin de línea*. De este modo se consigue independizar el proceso de lectura del dispositivo físico al que esté asignado el fichero *input*. Así, en la época en que se desarrollaba el Pascal, la mayoría de los ordenadores realizaban la entrada de datos a través de tarjetas perforadas. Cada tarjeta contenía una línea de datos y la *marca de fin de línea* se producía al alcanzar el final físico de la tarjeta. Actualmente el fichero *input* suele estar asignado al teclado y la *marca de fin de línea* se produce al pulsar la tecla **RETURN**, también denominada *intro*.

El comportamiento de la sentencia *Read* depende del tipo de variable que se esté leyendo. Solo se pueden leer variables de tipo *char*, *integer* o *real*.

Con una variable de tipo *char*, se lee un solo carácter de la entrada y se asigna a la variable. Aquí todos los caracteres son válidos, incluso el espacio en blanco, que es un carácter como otro cualquiera. Un caso especial se produce cuando el carácter leído es una *marca de fin de línea*. En esta situación el Pascal estándar asigna un carácter blanco a la variable que se está leyendo.

10 Consultar la sección 4.5 de este capítulo: *Extensiones del compilador Turbo Pascal, sentencia Read*

ENTRADA ESTANDAR

Con variables de tipo numérico (*integer* o *real*), la sentencia *Read* espera una secuencia de caracteres que se ajuste a la sintaxis del tipo de variable que se está leyendo. Todos los espacios en blanco, tabuladores y marcas de fin de línea que pudieran preceder a la secuencia numérica, se saltan. Si los datos introducidos no se ajustan al formato numérico correspondiente se produce un error de entrada/salida.

LA SENTENCIA *Readln*

La sentencia *Readln* sin parámetros salta al principio de la línea siguiente; es decir, avanza el puntero de lectura desde la posición actual hasta la posición inmediatamente siguiente a la próxima *marca de fin de línea* que se encuentre, con lo cual si existían datos aún no leídos en la línea actual son ignorados, y la próxima sentencia de lectura comenzará desde el principio de la línea siguiente.

La sentencia *Readln* también puede tener parámetros, en cuyo caso se comporta como una sentencia *Read*, con la diferencia de que una vez leído el último parámetro, salta al principio de la línea siguiente. En resumen, tenemos que

```
Readln(a,b,c,d);
```

es equivalente a la secuencia de sentencias

```
Read(a,b,c,d);  
Readln;
```

Ejemplo 4.3

```
Read (dia);  
Readln (mes);  
Read(anio);
```

si se teclean las dos líneas siguientes:

```
5 7 1111  
1993
```

las variables toman los valores:

```
dia:=5  
mes:=7  
anio:=1993
```

los 11111 son ignorados, ya que la sentencia *Readln* una vez leída la línea actual, avanza al principio de la línea siguiente, obligando a las siguientes sentencias de lectura a leer una nueva línea. Así se tiene que la variable *anio* toma el valor 1993.

El lenguaje Pascal tiene una función estándar *Eoln* que devuelve el valor lógico *true* si se ha detectado una indicación fin de línea en la línea que se está leyendo. En caso contrario, devuelve el valor *false*.

Cada línea de entrada tiene un carácter fin de línea que se puede denotar como *<eoln>*, que indica al ordenador donde acaba una línea y empieza la siguiente. Las sentencias *Read* y *Readln* atravesarán los límites de línea para encontrar tantos valores como identificadores haya en la lista de variables.

Ejemplo 4.4

Si se declaran las variables de la siguiente forma:

```
VAR
  nAprobados, nSuspendos, nPresentados, dia, mes, anio: integer;
```

entonces las sentencias de lectura se ejecutan así :

Sentencia	Datos	Contenidos en memoria después de leer
Readln (nAprobados)	50	nAprobados:=50
Readln (dia,mes,anio)	2 10 1989	dia :=2 mes :=10 anio:=1989
Readln (nSuspendos)	0.5432	!!! ERROR !!! ¹¹ La variable es de tipo entero, y se ha introducido un número real
Readln (nPresentados)	150	nPresentados:=150
Readln (dia,mes,anio)	20 10 1989	dia:=20 mes:=10 anio:=1989

Ejemplo 4.5

Sean tres líneas de entrada :

```
111 222 333<eoln>
444 555 666<eoln>
777 888 999<eoln>
```

y las variables declaradas de la siguiente forma:

```
VAR
  nCasos, nAlumnos, nAlumnas,nProfes: integer;
```

A continuación se indica para unas sentencias de lectura dadas, los valores asignados y la posición del indicador o puntero después de cada sentencia de lectura (^).

11 Consultar la sección 4.5 de este capítulo: *Extensiones del compilador Turbo Pascal, sentencia Read*

SALIDA ESTANDAR

Sentencias de lectura	Valores asignados	Posición del puntero (^) después de leer
Readln (nCasos);	nCasos:= 111	111 222 333<eoln> 444 555 666<eoln> ^
Readln (nAlumnos);	nAlumnos:= 444	777 888 999<eoln> ^
Read(nAlumnas,nProfes);	nAlumnas:=777 nProfes:=888	777 888 999<eoln> ^

Los datos de tipo *real* se tratan de la misma forma que los de tipo *integer*. Sin embargo con los datos de tipo *char* ocurre que, como cada variable de tipo *char* sólo almacena un carácter alfanumérico, cuando se leen valores en variables declaradas de tipo *char*, sólo se lee un carácter.

4.2 SALIDA ESTANDAR

Se realiza mediante los procedimientos estándar *Write* y *Writeln* siendo su diagrama sintáctico:

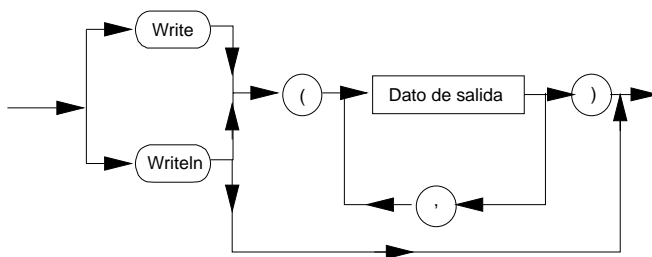


Figura 4.2 Diagrama sintáctico de Write y Writeln

Los datos de salida pueden ser:

- cadenas entre apóstrofes (ejemplo: 'x =')
- constantes
- variables (valores de variables)
- expresiones (resultado de evaluar la expresión)
- funciones (resultado de evaluar la función)

La sintaxis de estas sentencias en notación EBNF es:

```

<sentencia Write> ::= Write ( <dato de salida> { , <dato de salida> } )
<sentencia Writeln> ::= Writeln <argumentos>
<argumentos> ::= ( <dato de salida> { , <dato de salida> } )
<dato de salida> ::= <cadena de caracteres> | <constante> | <variable> |
<expresión> | <designador de función>
    
```

Hasta que estudiemos los ficheros (capítulo 11), la salida de nuestros programas se dirigirá siempre al fichero estándar de salida, denominado *output*, asociado generalmente a la pantalla.

Ejemplo 4.6

Sean las sentencias:

```
x:=123.456;
Write ('x=', x);
```

El resultado de su ejecución es:

```
x= 1.2345600E+2
```

Ejemplo 4.7

```
a:= 3; b:= -1;
Write ('Suma= ', a+b);
```

El resultado es:

```
Suma = 2
```

Ejemplo 4.8

```
Write ('Hola ');
Write ('no ');
Write ('pasa nada');
```

El resultado sale en una misma línea:

```
Hola no pasa nada^
```

y la siguiente escritura de datos se produciría en el lugar indicado por el puntero.

LA SENTENCIA *Writeln*

Hace que la siguiente salida salga en otra línea, es decir genera un carácter fin de línea después de escribir sus argumentos.

Ejemplo 4.9

```
Writeln ('Hola');
Writeln ('no');
Writeln ('pasa nada');
```

El resultado es:

ESTRUCTURA COMPLETA DE UN PROGRAMA

```
Hola
no
pasa nada
^
```

y la siguiente salida se produciría en una nueva línea, en el lugar indicado por el puntero.

Usada sin argumentos, la sentencia:

```
Writeln;
```

hace que se genere un carácter fin de línea; en el caso de que esté después de otra sentencia *Writeln* producirá una línea en blanco.

Ejemplo 4.10

```
Writeln ('Hola');
Writeln ('A continuación una línea en blanco');
Writeln ;
Writeln ('Estamos después de la línea en blanco');
```

El resultado es:

```
Hola
A continuación una línea en blanco
Estamos después de la línea en blanco
```

En definitiva se puede decir que la secuencia de sentencias:

```
Write (argumentos);
Writeln;
```

es equivalente a:

```
Writeln (argumentos);
```

4.3 ESTRUCTURA COMPLETA DE UN PROGRAMA

A continuación se presenta la estructura completa de un programa Pascal, aunque algunas de las partes que se citan serán desarrolladas en los capítulos sucesivos. Puede verse como la sección de declaraciones¹², precede al cuerpo de sentencias.

¹² También hay otras declaraciones, denominadas locales, que se estudiarán en el capítulo 7 (Subprogramas).

```

PROGRAM NombrePrograma (parámetros);

LABEL
{ Declaración de etiquetas }

CONST
{ Declaración de constantes }

TYPE
{ Declaración de tipos definidos por el usuario }

VAR
{ Declaración de variables }

PROCEDURE Nombre1 ...
{ Procedimientos y funciones }

FUNCTION Nombre2 ...
...

BEGIN { Comienzo del programa }
sentencial;
sentencia2;
...

sentenciaN;
END. { Fin del programa }

```

Los parámetros del programa se refieren a los ficheros que este va a utilizar. Por el momento, nos basta saber que deberemos referirnos a los ficheros estándar *input* y/o *output* según que nuestro programa vaya a realizar entradas (*Read*) y/o salidas (*Write*) respectivamente.

Si un programa va a necesitar ambas cosas, pondremos:

```
PROGRAM Nombre (input, output);
```

y si sólo va a producir salida de datos (por ejemplo para calcular el número *e* con 1000 cifras decimales), basta con poner:

```
PROGRAM NumeroE (output)
```

ya que no es necesario en este caso introducir ningún dato.

No siempre tienen por qué estar presentes todas las secciones del programa señaladas anteriormente, pero caso de que sean necesarias deberán aparecer en ese orden; es decir: la definición de constantes debe aparecer antes que la de variables, los procedimientos y/o funciones después etc.

Notas sobre la sintaxis de programas en Pascal

- Las *declaraciones* de un programa deben de colocarse en el orden que señalan los diagramas sintácticos, obligatoriamente.
- El esquema general de un programa en Pascal es:

EJERCICIOS RESUELTOS

1) CABECERA

2) BLOQUE (DECLARACIONES Y SENTENCIAS)

- El ; (punto y coma) se usa como separador entre sentencias o entre declaraciones sucesivas.
- *BEGIN* y *END* no van seguidas de ;, pues hay que interpretarlas como llaves que indican *PRINCIPIO* y *FIN* del programa o de una sentencia compuesta. Si se coloca ; después de un *END* es para separar esta sentencia compuesta de otra.
- Un punto y coma (;) antes de un *END*, se interpreta como una sentencia nula, es decir una sentencia vacía. Por lo tanto no es preciso colocar un ; antes de un *END*, aunque sí es aconsejable.
- Todo programa completo debe de terminar en un punto .

Comentarios

Un comentario no es una sentencia. Es un texto que es ignorado por el compilador, y su misión es ayudar a la comprensión del programa cuando es leído por el programador.

El nombre del autor, fecha del programa, el pseudocódigo etc., pueden introducirse en el programa en Pascal como comentarios.

En Pascal los comentarios se introducen entre llaves:

```
{ Esto es un comentario }
```

o entre (* y *):

```
(* Esto es otro comentario *)  
(* Esto no vale )  
{ Y esto tampoco },
```

aunque esto último sí es válido en algunos compiladores (no es el caso de Turbo Pascal).

Fundamentalmente se utilizarán comentarios, para aclarar aspectos del programa.

4.4 EJERCICIOS RESUELTOS

Una vez estudiadas las sentencias más elementales y la estructura completa de un programa, vamos a ver algunos ejemplos de desarrollo de programas completos. Detallaremos en estos primeros ejercicios resueltos las fases en que hemos dividido el proceso de programación, presentadas en el capítulo 2.

4.1 Cálculo del área de un rectángulo. Se plantea un problema elemental para determinar la superficie de un rectángulo, a partir de su base y altura.

- *Análisis*

La solución se obtiene aplicando la formula:

$$\text{Superficie} = \text{base} \times \text{altura}$$

- *Algoritmo (pseudocódigo)*

Inicio

 Escribir 'Dame la base :'
 Leer la base
 Escribir 'Dame altura :'
 Leer la altura
 Superficie := base * altura
 Escribir 'Superficie=', Superficie

Fin

- *Codificación en Pascal*

```
PROGRAM rectangulo (input, output);
VAR
  base, altura, superficie: integer;
BEGIN
  Write('Dame la base: ');
  Readln(base);
  Write('Dame la altura: ');
  Readln(altura);
  superficie := base * altura;
  Writeln('Superficie = ',superficie);
  Write('Pulse <Intro> para volver al editor');
  Readln; (* Para retener en pantalla los resultados *)
           (* Probar el efecto al eliminar esta sentencia *)
END.
```

- *Compilación*

Se corrigen los posibles errores de sintaxis, hasta que el compilador emite un mensaje indicando que se ha realizado la compilación con éxito.

- *Ejecución*

```
Dame la base : 3
Dame la altura : 2
Superficie = 6
Pulse <Intro> para volver al editor
```

4.2 El departamento de publicidad de unos grandes almacenes ha realizado el siguiente anuncio:

GRANDES TARTAS "PIN"
CAMPAÑA DE TARTAS A MEDIDA

EJERCICIOS RESUELTOS

Nos da el radio de la tarta en centímetros y su espesor, y automáticamente le decimos el precio de su tarta de almendra

Y encarga un programa al departamento de informática para agilizar la entrega de presupuestos. El departamento de informática elabora el programa paso a paso en las siguientes fases:

- *Análisis*

$$\text{Volumen} = \pi \cdot (\text{radio})^2 \cdot \text{espesor}$$

$$\text{Peso} = \text{Volumen} \cdot \text{densidad}$$

$$\text{Precio} = (\text{peso en kg.}) \cdot (\text{precio costo por kg.}) + \text{beneficios}$$

$$\text{Precio costo} = \text{precio componentes por Kg.}$$

- *Algoritmo*

Inicio

Leer el radio de la tarta en cm.

Leer el espesor de la tarta en cm.

Cálculo del precio de costo por kg.

Añadir los beneficios

Cálculo del volumen

Cálculo del precio total (volumen * densidad * precio de 1 Kg)

Escribir precio

Fin

- *Codificación en Pascal*

```
Program Tarta (input,output);
CONST
  pi=3.14159 ;
  levadura=100;
  harina=60;
  azucar=160;
  almendra=800;
  huevo=10;
  beneficiosPorCiento=20;
(* la densidad se obtiene experimentalmente y se mide en kg/dm3 *)
  densidad=2.5;
VAR
  radio,espesor :integer;
  precioTotal,preciolkg,volumen :real;
```

ENTRADA / SALIDA

```
BEGIN
Write('Introduzca el radio de la tarta en centímetros: ');
Readln (radio);
Write('Introduzca el espesor de la tarta en centímetros: ');
Readln (espesor);
(* fórmula secreta del repostero/a *)
preciokg:=levadura/100+harina/4+azucar/4+almendra/2+huevo*4;
(* Beneficios de la confitería *)
preciokg:=preciokg*(100+beneficiosPorCiento)/100;
volumen:=pi*sqr(radio)*espesor;
precioTotal:=volumen*densidad/1000*preciokg;
Writeln;
Writeln('El precio de una tarta de radio= ',radio,
        ' y espesor= ',espesor,' es ',precioTotal,' pts ');
Writeln ('Pulse <Return> para volver al Editor');
Readln;
END.
```

- *Compilación*

Se corrigen los posibles errores de sintaxis, hasta que el compilador emite un mensaje indicando que se ha realizado la compilación con éxito.

- *Ejecución*

```
Introduzca el radio de la tarta en centímetros: 50
Introduzca el espesor de la tarta en centímetros: 10
El precio de una tarta de radio= 50 y espesor= 10 es
1.1686714800E+05 pts
Pulse <Return> para volver al Editor
```

- *Prueba y Mantenimiento*

Es necesario probar el programa para distintos casos. Probar los casos particulares, los casos más frecuentes, etc. En este caso, se puede comprobar que el precio de la tarta resulta un poco elevado (incluso para valores pequeños del radio y el espesor, por lo que sería conveniente confirmar que las fórmulas utilizadas son las correctas, antes de dar por concluido el programa. También se puede probar la tarta, para verificar la fórmula del repostero.

Respecto al mantenimiento, es posible que el cliente esté interesado en hacer algo similar con otros tipos de tarta o incluso con otros productos, si los resultados de la campaña publicitaria son optimistas. Si el programa está bien estructurado, se facilitará la tarea de adaptar el programa a otros casos distintos de la tarta de almendra.

4.5 EXTENSIONES DEL COMPILADOR TURBO PASCAL

Sentencia *Read*

En Turbo Pascal, la sentencia *Read* tiene alguna diferencia respecto al estándar. Cuando una sentencia *Read* encuentra un *<Eoln>* al leer una variable de tipo *char*, asigna un retorno de carro (o carácter de fin de línea) a la variable. En Pascal estándar se asignaría un espacio en blanco.

Con variables numéricas, el Turbo Pascal obliga a que la secuencia numérica de entrada se termine con un espacio en blanco, tabulador o marca de fin de línea. En el estándar no existe esta restricción. La norma no especifica cómo debe finalizar la secuencia numérica de entrada, pero por razones prácticas, la mayoría de los compiladores admiten como marcas de fin de secuencia el espacio en blanco y la marca de fin de línea.

Por esta razón, en los ejemplos 4.2 y 4.4, aunque la variable `nSuspensos` es de tipo entero, si nos atenemos a la norma la sentencia de lectura no tiene por qué producir un error de ejecución, sino que puede tomarse el carácter '.' como final de secuencia numérica de entrada, en cuyo caso la variable `nSuspensos` tomará el valor 0, según se indica a continuación:

Sentencia	Datos	Contenidos en memoria después de leer
<code>Read (nSuspensos)</code>	0.5432	<code>nSuspensos := 0</code>
<code>Readln (nSuspensos)</code>	0.5432	<code>nSuspensos := 0</code>

Ni en Pascal estándar ni en Turbo Pascal se permite la asignación de un valor real a una variable de tipo entero. Generalmente, una situación como la presentada en los ejemplos 4.2 y 4.4 es debida a un error no intencionado en la secuencia numérica de entrada, o a una confusión al definir el tipo de la variable `nSuspensos`. Es por ello que, con fines didácticos, consideramos preferible no admitir como válida la sentencia anterior.

Estructura de un programa completo en Turbo Pascal

Un programa en Turbo Pascal tiene que adaptarse a la estructura de la figura 4.3.

El programa consta de una cabecera, la cláusula opcional *Uses*, y el bloque del programa principal. Dentro del bloque del programa principal pueden existir bloques menores de procedimientos y funciones (subprogramas). Aunque no se indica en la figura, dentro de estos bloques menores pueden anidarse otros procedimientos y funciones.

Un *token* es una unidad sintáctica del lenguaje (palabras reservadas, identificadores, operadores, separadores, etc.). Las *expresiones* se forman combinando *tokens* con otros *tokens* y espacios en blanco. Combinando *expresiones* se forman *sentencias*. Combinando *sentencias* con *declaraciones* formamos *bloques*, ya sean de programas principales o de subprogramas.

En Turbo Pascal se puede omitir la cabecera del programa, que incluye el nombre y la lista de parámetros usados.

Respecto a la utilización de la cláusula *Uses*, consultar las secciones 7.9 (Programación gráfica) y 7.12 (Subprogramas externos con Turbo Pascal) del capítulo siete, *Subprogramas*.

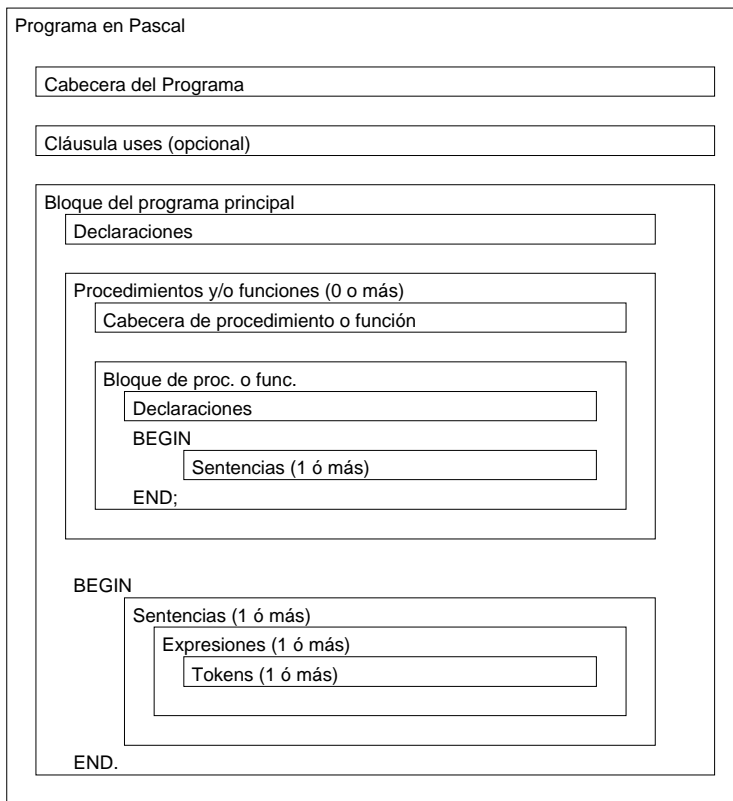


Figura 4.3 Estructura de un programa en Turbo Pascal

Procedimientos y Funciones para Entrada/Salida

Además de los procedimientos estándar *Read*, *Readln*, *Write*, *Writeln*, la función *Eoln*, y los que estudiaremos en el capítulo de ficheros: *Reset*, *Rewrite*, y la función *Eof*, Turbo Pascal incorpora los procedimientos y funciones de la tabla 4.1. Se incluyen también los de manejo ficheros, aunque se utilizarán más adelante (capítulo 11).

Si desea más información sobre estos subprogramas, consultar el manual del compilador, o la utilidad *Help*, mientras se está utilizando el programa.

EXTENSIONES DEL COMPILADOR TURBO PASCAL

Procedimiento o función	Descripción
<i>Append</i>	Abre un fichero de texto ya existente, para añadirle información.
<i>Assign</i>	Asocia el nombre de un fichero externo a una variable fichero.
<i>BlockRead</i>	Lee uno o más registros de un fichero sin tipo.
<i>BlockWrite</i>	Escribe uno o más registros en un fichero sin tipo.
<i>ChDir</i>	Cambia el directorio actual.
<i>Close</i>	Cierra un fichero abierto.
<i>Erase</i>	Borra un fichero externo.
<i>FilePos</i>	Devuelve la posición actual de lectura/escritura en un fichero.
<i>FileSize</i>	Devuelve el tamaño actual de un fichero (no de texto).
<i>Flush</i>	Vacía el buffer de un fichero de texto para salida.
<i>GetDir</i>	Devuelve el directorio actual de la unidad especificada.
<i>IOResult</i>	Devuelve un valor entero que representa el resultado de la última operación de entrada/salida realizada.
<i>MkDir</i>	Crea un subdirectorio.
<i>Rename</i>	Cambia de nombre un fichero externo.
<i>RmDir</i>	Borra un subdirectorio vacío.
<i>Seek</i>	Posiciona el indicador de lectura/escritura en el elemento especificado en un fichero (no de texto).
<i>SeekEof</i>	Devuelve el estado del fin de fichero en un fichero de texto.
<i>SeekEoln</i>	Devuelve el estado del fin de línea en un fichero de texto.
<i>SetTextBuf</i>	Asigna un buffer de entrada/salida a un fichero de texto.
<i>Truncate</i>	Trunca un fichero en la posición actual del cursor de lectura/escritura dentro del fichero.

Tabla 4.1 Procedimientos y funciones de Turbo Pascal para entrada/salida

Entrada/Salida usando la *unit Crt*

La *unit Crt* de Turbo Pascal contiene una serie de procedimientos y funciones para facilitar la entrada y salida de datos a través de teclado y pantalla. Se usan mucho para mejorar la presentación en pantalla de los resultados. Para utilizarla, hay que incluir en la cabecera del programa la siguiente cláusula *Uses*:

```
Uses Crt;
```

Cuando se escribe en un dispositivo utilizando la *unit Crt*, los caracteres de control siguientes tienen los significados especiales de la tabla 4.2.

<i>Carácter</i>	<i>Nombre</i>	<i>Descripción</i>
#7	BELL	Emite un pitido
#8	BS	Retrocede un espacio
#10	LF	Avanza una línea
#13	CR	Posiciona el cursor en la esquina izquierda de la ventana actual

Tabla 4.2 Caracteres de control especiales de la *unit Crt*

Los procedimientos y funciones incluidos en la *unit Crt* se detallan a continuación. Para cada subprograma se indica en primer lugar su declaración y a continuación se explica su funcionamiento. Para poder utilizarlos hasta que estudiemos subprogramas (capítulo 7), adelantaremos que los procedimientos (procedures) se utilizan como si fuesen sentencias (Read y Write son procedimientos estándar del lenguaje), y las funciones se utilizan igual que las funciones estándar ya estudiadas.

Procedure AssignCrt(VAR F:Text);

Asocia un fichero de texto con la ventana *Crt*. Funciona igual que el procedimiento *Assign*, con la diferencia de que no se especifica el nombre físico del fichero. Nos facilita y agiliza la salida (y entrada).

Ejemplo 4.11

```
PROGRAM EjemploAssignCrt(input, output);
Uses Crt;
VAR F: text;
    resp:char;
BEGIN
  Write('¿Salida hacia pantalla o impresora (P/I)? ');
  Readln(resp);
```

EXTENSIONES DEL COMPILADOR TURBO PASCAL

```
IF (resp='i') OR (resp='I')      (* Impresora *)
  THEN Assign(F, 'PRN')
  ELSE AssignCrt(F);             (* Pantalla *)
Rewrite(F);
Writeln(F, 'Salida rápida vía subprogramas de la Crt');
Close(F);
END.
```

Procedure ClrEol;

Borra los caracteres desde el cursor al final de línea, sin mover el cursor. Si está definida una ventana (con el procedimiento *Window*), es relativo a la ventana actual. Es decir, borraría los caracteres hasta el borde derecho de la ventana. Mantiene los atributos de texto en las posiciones de los caracteres borrados (por ejemplo, el color de fondo definido con *TextBackGround*).

Procedure ClrScr;

Borra la pantalla y posiciona el cursor en la esquina superior izquierda. Mantiene los atributos de texto definidos. Por ejemplo, si tenemos definido un color de fondo con *TextBackGround*, la pantalla entera quedará de dicho color. Si tenemos definida una ventana con *Window*, es relativo a la ventana actual, se borraría la ventana actual y el cursor se posicionaría en la esquina superior izquierda de dicha ventana (que puede no coincidir con la misma esquina de la pantalla)

Procedure Delay(Ms:word);

Detiene la ejecución (retarda) el número de milisegundos especificado. El periodo de espera es aproximado, puede no coincidir exactamente con Ms milisegundos. Se utiliza para dar tiempo al usuario para leer mensajes u observar resultados durante la ejecución del programa.

Procedure DelLine;

Borra la línea que contiene el cursor y mueve todas las inferiores una línea hacia arriba, añadiendo una línea nueva al final. Se mantienen los atributos de texto definidos. Por ejemplo, si tenemos definido un color de fondo con *TextBackGround*, la línea nueva quedará de dicho color. Si tenemos definida una ventana con *Window*, es relativo a la ventana actual.

Procedure GotoXY(X, Y: Byte);

Posiciona el cursor en la columna X, línea Y. Se considera que las coordenadas de la esquina superior izquierda son (1,1). Es relativo a la ventana actual. Si las coordenadas X, Y, no son válidas, no se produce un error de ejecución, sino que la llamada es ignorada.

Procedure HighVideo;

Selecciona caracteres de alta intensidad. Las siguientes sentencias de salida escribirán caracteres de alta intensidad. El color de fondo no cambia.

Procedure InsLine;

Inserta una línea vacía en la posición del cursor. Todas las líneas posteriores se mueven una línea hacia abajo, desapareciendo la línea inferior de la pantalla. La nueva línea tendrá los atributos de texto actualmente definidos. Esto es, si se ha definido un color de fondo con `TextBackGround`, aparecerá de ese color. Es relativo a la ventana actual.

Function KeyPressed: Boolean;

Devuelve *true* al ser pulsada cualquier tecla del teclado, *false* en otro caso. El carácter correspondiente a la tecla pulsada queda en el buffer del teclado. *KeyPressed* no detecta teclas de intercambio como *Shift*, *Alt*, *NumLock*, etc.

Procedure LowVideo;

Funciona igual que *HighVideo*, pero selecciona caracteres de baja intensidad.

Procedure NormVideo;

Restablece el atributo de intensidad de caracteres, a partir de la posición del cursor, al valor original que tenía al arrancar el programa.

Procedure NoSound;

Desactiva el micrófono interno.

Function Readkey:char;

Devuelve un carácter leído de teclado. El carácter leído no aparece en pantalla. Si *KeyPressed* era *True* antes de la llamada a *ReadKey*, el carácter se devuelve inmediatamente. En caso contrario, *ReadKey* espera la pulsación de una tecla.

Procedure Sound(Hz: word);

Activa el micrófono interno, emitiendo un sonido de frecuencia Hz. Para interrumpir el sonido hay que utilizar *NoSound*.

Ejemplo 4.12

```
PROGRAM EjemploSonido(output);
uses Crt;
BEGIN
  Sound(220);
  Delay(200);
  NoSound;
END.
```

Procedure TextBackGround(Color: byte);

Selecciona el color del fondo. El parámetro *Color* es una expresión entera en el rango 0..7, o la constante de color correspondiente (ver manual). Después de su utilización, los caracteres se escriben con el color de fondo seleccionado.

Procedure TextColor(Color: byte);

Selecciona el color de los caracteres, del texto escrito posteriormente a su utilización. El parámetro *Color* es una expresión entera en el rango 0..15, o la constante de color correspondiente. Se pueden hacer parpadear los caracteres, en el color deseado, sumando 128 o la constante *blink* al valor deseado.

Ejemplo 4.13

```
TextColor(Green);           (* caracteres verdes *)
TextColor(LightRed + Blink); (* caracteres rojos resaltados parpadeando *)
TextColor(14);              (* caracteres amarillos *)
```

Procedure TextMode(Mode: Word);

Selecciona un modo de texto específico. Consultar en el manual las constantes de modos *Crt*. Cuando se llama a *TextMode*, la ventana actual es restaurada a la pantalla completa, y las variables de la *Crt* que almacenan atributos de texto y vídeo son restauradas a sus valores por defecto.

Especificar *TextMode(LastMode)* selecciona el último modo de texto activo. Es útil para regresar a modo texto después de usar una *unit* gráfica como *Graph* o *Graph3*.

Procedure Window(X1, Y1, X2, Y2: byte);

Define una ventana de texto en la pantalla. *X1, Y1* son las coordenadas de la esquina superior izquierda de la ventana, y *X2, Y2* las de la esquina inferior derecha. *X* representa columnas e *Y* líneas. La esquina superior izquierda de la pantalla tiene coordenadas (1, 1). Si las coordenadas no son válidas, la llamada a *Window* se ignora. El tamaño mínimo de una ventana es una columna

por una línea.

Muchos procedimientos y funciones de la unit *Crt* son relativos a ventanas de texto: *ClrEol*, *ClrScr*, *DelLine*, *GotoXY*, *InsLine*, *WhereX*, *WhereY*, *Read*, *Readln*, *Write*, *Writeln*.

Function WhereX: Byte;

Devuelve la coordenada *X* (columna) de la posición actual del cursor, relativa a la ventana activa.

Function WhereY: Byte;

Devuelve la coordenada *Y* (fila) de la posición actual del cursor, relativa a la ventana actual.

Para ver un ejemplo de utilización de la unit *Crt* en la salida a pantalla, consultar el ejercicio resuelto 7.1 del capítulo 7 (Subprogramas).

Salida a impresora

Si queremos que todas o parte de las sentencias de salida de un programa se dirijan a la impresora en lugar de hacerlo a la pantalla, procederemos como se indica a continuación.

En primer lugar, hay que incluir en la cláusula *Uses* la unit *Printer*, para indicar al compilador Turbo Pascal que vamos a utilizar la impresora:

```
Uses Printer;
```

A continuación, en las sentencias de salida que queramos dirigir a la impresora, antepondremos el identificador *lst* a la lista de variables de salida:

```
Write(lst, ...);
Writeln(lst, ...);
```

lst es el nombre de una variable de tipo fichero de texto (se estudiarán en el capítulo 11), que el compilador asocia al dispositivo LPT1 o PRN (impresora).

Ejemplo 4.14

Veamos un ejemplo sencillo de programa con salida a impresora.

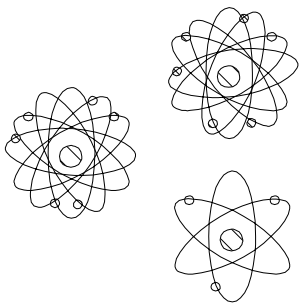
```
PROGRAM Impresora(output);
Uses Printer;
VAR ch:char;
BEGIN
  Writeln('Imprimiendo en pantalla:');
  Write('EN PANTALLA YA SABEMOS ESCRIBIR MENSAJES ');
  Writeln;
  Writeln('PROBEMOS AHORA A ESCRIBIR CON LA IMPRESORA');
  Writeln(lst, 'Imprimiendo en impresora:');
  Write(lst, 'Dirigir la salida a la impresora ');
  Write(lst, 'es muy sencillo. ');
  Writeln(lst);
```

EJERCICIOS PROPUESTOS

```
Write('Introduzca un caracter para imprimir...');
Readln(ch);
Writeln(1st,'Imprimimos tu caracter...',ch);
Write(1st, 'AHORA TAMBIEN SABEMOS ESCRIBIR EN LA IMPRESORA');
Writeln;
Write('Pulse <Intro> para volver al editor...');
Readln;
END.
```

4.6 EJERCICIOS PROPUESTOS

- 4.1 Hacer un programa que pregunte las iniciales del nombre y apellidos al usuario y los vuelva a escribir por pantalla.
- 4.2 Escribir un programa que lea dos números por teclado y escriba su suma, diferencia, producto y cociente.
- 4.3 Realizar un programa que lea un radio, r , por teclado y determine:
 - La longitud de la circunferencia de radio r .
 - El área del círculo de radio r .
 - El volumen de una esfera de radio r .
- 4.4 Escribir un programa que calcule el volumen y la superficie lateral de un cono, a partir del radio de su base y el semiángulo.
- 4.5 Basándose en el ejercicio anterior de la tarta, resuelto en este capítulo, escribir uno similar para el cálculo del precio de pizzas. Consultar la fórmula en la pizzería más próxima.



CAPITULO 5

TIPOS DE DATOS SIMPLES

CONTENIDOS

- 5.1 Introducción
- 5.2 Tipo entero
- 5.3 Tipo boolean
- 5.4 Tipo carácter
- 5.5 Tipo real
- 5.6 Tipos definidos por el usuario
- 5.7 Ampliaciones del Pascal estándar con Turbo Pascal
- 5.8 Cuestiones y ejercicios resueltos
- 5.9 Cuestiones y ejercicios propuestos
- 5.10 Ampliaciones y notas bibliográficas

5.1 INTRODUCCION

En el capítulo tercero se estudiaron los tipos de *datos simples predefinidos*. Otros tipos de datos muy utilizados son los *datos definidos por el usuario* pero antes de abordar estos tipos de datos se hará un estudio más detallado de los datos simples.

TIPO ENTERO

5.2 TIPO ENTERO

En el tipo entero se engloban las constantes enteras, variables enteras, expresiones enteras y funciones de tipo entero.

CONSTANTES ENTERAS

Tal y como se vio en el capítulo tercero, una constante entera se define de la siguiente forma:

```
CONST
  nombre1 = valor1 ;
  nombre2 = valor2 ;
  . . .
```

donde `nombre1`, `nombre2`, ... son los identificadores que representan los nombres de cada constante, y `valor1`, `valor2`, ... son los valores respectivos de dichas constantes enteras.

La sintaxis en notación EBNF es la siguiente:

```
<constante entera> ::= <signo> <entero sin signo> | <entero sin signo>
<entero sin signo> ::= <dígito> {<dígito>}
```

Ejemplo 5.1

```
CONST
  numCasos = 250 ;
  numAlumnos = 150 ;
  numOrdenadores = 10 ;
```

El lenguaje Pascal incluye una constante entera estándar que se denota con el identificador **maxint**, y especifica el mayor valor que puede ser asumido por una cantidad de tipo entero.

El rango de los enteros permitidos va desde :

- maxint hasta maxint

siendo su rango generalmente entre -32768 y 32767.

VARIABLES ENTERAS

Las variables de tipo entero, se declaran como ya se vió en el capítulo tercero, de la siguiente forma:

```
VAR
  nombre1, nombre2, ... : integer ;
```

donde `nombre1`, `nombre2`, ... son los identificadores que representan el nombre de las variables.

Ejemplo 5.2

```
VAR
  edad, numPersonas, indica : integer ;
```

EXPRESIONES ENTERAS

Una expresión entera está formada por una serie de operandos (números, constantes, variables y funciones enteras) enlazados por operadores aritméticos. Existen seis operadores aritméticos que pueden utilizarse con operandos de tipo entero.

Los operadores aritméticos y sus características son los que se muestran en la tabla 5.1. Solamente citar que el operador - si es unario contiene solamente un operando, el resto de los operadores son binarios.

Operador aritmético	Efecto	Tipo de operandos	Tipo de resultado
- (unario)	Cambio de signo	Entero	Entero
+	Adición	Entero	Entero
-	Sustracción	Entero	Entero
*	Multiplicación	Entero	Entero
DIV	División Entera	Entero	Entero
MOD	Módulo (resto)	Entero	Entero

Tabla 5.1 Operadores aritméticos con operandos enteros

Las normas para aplicar los operadores aritméticos para formar expresiones enteras son:

- El resultado será positivo si ambos operandos son del mismo signo. En caso contrario será negativo.
- Los operadores DIV y MOD requieren que el segundo operando no sea nulo.
- Dos operadores no pueden ir seguidos, a no ser que estén separados por un paréntesis.

Cuando en una expresión aparece más de un operador, la secuencia de evaluación de la expresión depende de la precedencia de los operadores. El operador con mayor precedencia se evaluará primero. Cuando dos o más operadores tienen el mismo nivel de precedencia, la evaluación procede de izquierda a derecha. Los niveles de precedencia para los operadores aritméticos son:

- 1º) - (unario)
- 2º) *, DIV, MOD
- 3º) +, -

Los paréntesis alteran la precedencia de los operadores forzando al programa a evaluar primero la expresión que está dentro del paréntesis.

TIPO ENTERO

Ejemplo 5.3

Expresión	Resultado
-(+2)	-2
3+7	10
3-7	-4
3*7	21
3 DIV 7	0
10 DIV 4	2
13 DIV 4	3
13 DIV 0	**ERROR**
3 MOD 2	1
3 MOD 3	0
3 MOD 5	3
3 MOD 0	**ERROR**
0 MOD 3	0
0 MOD 4	0
27 MOD 5	2

Ejemplo 5.4

$$77 * (23 + 77)$$

aquí se evalúa antes la suma que la multiplicación.

En los paréntesis anidados, es decir paréntesis dentro de paréntesis, se evaluará primero la expresión más interior.

Ejemplo 5.5

$$3 * 7 + 100 * (((3 + 5) * 2 DIV 8) + 2) DIV 2 + 1)$$

La expresión se calcula por el siguiente orden:

- 1º) Se multiplica 3×7 y da 21.
- 2º) El paréntesis más interior es $(3 + 5)$, que vale 8.
- 3º) El siguiente paréntesis se evalúa de izquierda a derecha pues los operadores * y DIV tienen el mismo nivel de precedencia, es decir se calcula primero 8×2 , resultando 16, y se divide por 8, resultando 2.
- 4º) Se suma 2, resultando 4.
- 5º) Se divide por 2, resultando 2.

6º) Se añade 1, resultando 3.

7º) Se multiplica por 100, resultando 300.

8º) Se añade a 21 el valor 300, y se obtiene 321.

FUNCIONES DE TIPO ENTERO

El lenguaje Pascal tiene funciones predefinidas o estándar, que también se denominan funciones *intrínsecas* o *internas*. Algunas de estas funciones aceptan un tipo de parámetro y devuelven un valor del mismo tipo, mientras que otras devuelven un valor de tipo diferente al aceptado. A continuación se presentan las funciones internas que aceptan y devuelven valores de tipo entero exceptuando la función Sqrt(x).

Función	Efecto	Tipo de parámetro (x)	Tipo de resultado
Abs (x)	Calcula el valor absoluto de x	Entero	Entero
Sqr (x)	Calcula el cuadrado de x	Entero	Entero
Sqrt (x)	Calcula la raíz cuadrada de x	Entero	Real
Pred (x)	Determina el predecesor de x	Entero	Entero
Succ (x)	Determina el sucesor de x	Entero	Entero

Tabla 5.2 Funciones internas de tipo entero

Ejemplo 5.6

Valor de x	Función	Notación	Resultado
-3	Abs (x)	Abs (-3)	3
-5	Sqr (x)	Sqr (-5)	25
2	Sqrt (x)	Sqrt (2)	1.4142135
27	Pred (x)	Pred (27)	26
27	Succ (x)	Succ (27)	28
5	Abs (x)	Abs (5)	5
-2	Pred (x)	Pred (-2)	-3
-2	Succ (x)	Succ (-2)	-1

Los valores de tipo entero están ordenados de menor a mayor:

-maxint , ... , -3 , -2 , -1 , 0 , 1 , 2 , 3 , ... , maxint

TIPO BOOLEAN

y se les puede aplicar las funciones intrínsecas $\text{Pred}(x)$ y $\text{Succ}(x)$. Obsérvese que $\text{Succ}(\text{maxint})$ y $\text{Pred}(-\text{maxint})$ no están definidos y producirán error.

5.3 TIPO BOOLEAN

Antes de presentar el tipo de datos "boolean" en Pascal, se describen a continuación los fundamentos del *Algebra de Boole*. Solo se presentarán las definiciones y leyes básicas, sin entrar en el rigor que entraña una definición matemática, si bien se mantienen algunos aspectos matemáticos formales, pensando que la comprensión del vocabulario booleano permitirá al lector formular, calcular y en definitiva entender mejor las expresiones booleanas que aparezcan en los programas.

Se llaman variables booleanas aquellas que pueden tomar únicamente dos valores: *verdadero* o *falso*; que en lo sucesivo denotaremos por **true** y **false** respectivamente, siguiendo la nomenclatura adoptada por el lenguaje Pascal.

Los tres operadores lógicos más importantes son:

- La unión lógica, representada por OR.
- La conjunción lógica AND.
- El operador negación representado por NOT.

Existen otros operadores (or exclusivo, equivalencia, ...), pero con los tres citados se pueden representar todas las combinaciones posibles de cualquier número de variables booleanas.

Los operadores AND y OR son binarios (se aplican a dos operandos), mientras que el operador NOT es unario. Veamos la tabla de verdad correspondiente a cada uno de ellos:

p	q	p AND q	p OR q	NOT p
F	F	F	F	T
F	T	F	T	T
T	F	F	T	F
T	T	T	T	F

donde:

$T \rightarrow \text{true}$

$F \rightarrow \text{false}$

Tabla 5.3 Tabla de verdad de los operadores AND, OR y NOT

Su interpretación es sencilla:

- La operación AND es cierta únicamente en el caso de que ambas variables **p** y **q** sean ciertas.
- La operación OR es cierta si **p** o **q** (o ambas) son ciertas.
- Con la operación NOT si la variable a la que afecta es cierta, el resultado es falso. Si es falsa, el resultado es cierto.

En la tabla anterior se aplica sobre la variable **p**.

Veamos algunas propiedades del Algebra de Boole:

1) Las operaciones AND y OR son conmutativas:

$$p \text{ AND } q = q \text{ AND } p$$

$$p \text{ OR } q = q \text{ OR } p$$

2) Elementos neutros:

true para la operación AND
false para la OR.

$$p \text{ AND } \text{true} = p$$

$$p \text{ OR } \text{false} = p$$

3) Cada operación es distributiva respecto a la otra:

$$p \text{ AND } (q \text{ OR } r) = (p \text{ AND } q) \text{ OR } (p \text{ AND } r)$$

$$p \text{ OR } (q \text{ AND } r) = (p \text{ OR } q) \text{ AND } (p \text{ OR } r)$$

4) Doble negación:

$$\text{NOT} (\text{NOT } p) = p$$

5) Leyes de Morgan:

$$\text{NOT} (p \text{ AND } q) = (\text{NOT } p) \text{ OR } (\text{NOT } q)$$

$$\text{NOT} (p \text{ OR } q) = (\text{NOT } p) \text{ AND } (\text{NOT } q)$$

6) Leyes de redundancia:

$$p \text{ AND } \text{false} = \text{false}$$

$$p \text{ OR } \text{true} = \text{true}$$

$$p \text{ AND } p = p$$

$$p \text{ OR } p = p$$

Mediante estas reglas se puede escribir cualquier expresión de Boole en su forma más simplificada. Se debe comprender perfectamente el significado de las operaciones AND, OR y NOT, de esta forma resultará muy fácil razonar las propiedades antes descritas.

DATOS DE TIPO BOOLEAN EN PASCAL

El Pascal, a diferencia de otros lenguajes, incorpora el tipo de datos booleano descrito anteriormente. El tipo *boolean* es de gran utilidad para hacer preguntas y verificar ciertas condiciones a lo largo de un programa como se verá en el apartado siguiente; pero, antes de nada, veamos como se define este tipo de datos en Pascal.

Los identificadores estándar que representan los dos únicos valores de este tipo de datos son *true* y *false*. Y estos son los valores que podrán tomar las variables booleanas.

Si se desea declarar las variables *a*, *b* y *test* como booleanas, se hará lo siguiente:

TIPO BOOLEAN

```
VAR
  a, b, test :boolean;
```

es decir, indicando que su tipo es *boolean*.

Las variables booleanas no pueden leerse. Escribir *Read(a)* en un programa sería incorrecto. Sus valores deben ser asignados por programa.

Las sentencias de asignación:

```
  a := true;
  test := false;
```

ponen a *true* la variable *a* y *false* a *test*.

Aunque no es muy usual, sus valores pueden escribirse mediante una sentencia *Write* o *Writeln*.

Las únicas constantes booleanas válidas son *true* y *false*.

EXPRESIONES BOOLEANAS

En Pascal las preguntas se hacen mediante expresiones booleanas que al evaluarlas darán un valor *true* si la respuesta a la pregunta es afirmativa, y *false* en caso contrario.

Una expresión booleana puede ser:

- una constante (*true* o *false*),
- una variable booleana,
- una función booleana

o una combinación de ellas unidas mediante los operadores *OR*, *AND* y *NOT*; o una relación o comparación de datos numéricos o caracteres.

La jerarquía de estos operadores es la siguiente:

NOT	mayor jerarquía
AND	jerarquía media
OR	menor jerarquía

Al igual que ocurre con las expresiones de tipo *integer* y *real*, se pueden utilizar paréntesis, los cuales alteran el orden de precedencia de todos los operadores:

a OR b AND test

es lo mismo que

a OR (b AND test)

pero es diferente de

(a OR b) AND test

Algunos ejemplos de asignaciones válidas son:

TIPOS DE DATOS SIMPLES

```
a := true;
b := NOT a;
test := NOT (a AND b);
b := 5 > 7;
c := 'd' < 'f'
```

tras la ejecución de estas tres sentencias, *a* tendrá el valor *true*, *b* será *false* y *test* tomará el valor *true*.

FUNCIONES INCORPORADAS

El Pascal tiene tres funciones estándar que devuelven un resultado Booleano al ser llamadas:

Odd(x): el parámetro *x* debe ser un *integer*. La función devuelve *true* si *x* es un número impar, y *false* si es par.

Eof(f): devuelve *true* si se ha alcanzado el fin de fichero *f* que se le pasa como parámetro.

Eoln(f): devuelve *true* si se ha alcanzado el fin de una línea en el fichero de texto *f*.

Succ(x): devuelve *true* si *x* es *false* y da error si *x* es *true*.

Pred(x): devuelve *false* si *x* es *true* y da error si *x* es *false*.

Estas dos últimas funciones se estudiarán más adelante.

Veamos, a través de un ejemplo, cómo se evalúa una expresión en la que intervienen funciones:

```
VAR
  a, b : boolean;
  x : integer;
BEGIN
  x := 4;
  a := false;
  b := NOT Odd(x+1) OR a;
  ...
END.
```

El orden de evaluación de la última sentencia es el siguiente:

- 1º) Se calcula el valor $x+1$. El resultado es 5
- 2º) Este valor se pasa como parámetro a *odd*, que devolverá el valor *true* pues 5 es número impar.
- 3º) El operador *NOT* actúa sobre dicho valor convirtiéndolo en *false*.
- 4º) Ahora se efectúa la operación *OR*, cuyo resultado final será *false*, y se le asigna este valor a la variable *b*.

RELACIONES O COMPARACIONES

También se obtiene un resultado booleano al evaluar cualquier relación o comparación. Una relación se define como:

TIPO BOOLEAN

<expresión> <operador-de-relación> <expresión>

Las expresiones a comparar deben ser del mismo tipo: números con números (integer o real); caracteres con caracteres; etc.

Los operadores de relación que se pueden utilizar son:

OPERADOR	SIGNIFICADO
=	igual a
<>	distinto de
<	menor que
>	mayor que
<=	menor o igual
>=	mayor o igual

Tabla 5.4 Operadores de relación

Por ejemplo, si x vale 5, y z vale 6; las siguientes comparaciones son todas *true*.

```
x <> z
x < z
x = z-1
x >= 2.5
```

También darían un resultado *true* las comparaciones:

```
true = true
false = false
false < true
```

Si tenemos declaradas las variables:

```
VAR
  x, z : integer;
  b : boolean;
```

los siguientes son algunos ejemplos de asignaciones válidas:

```
b := x = z+1;
```

pondrá b a *true* si el valor de la variable x es una unidad superior al de la variable z .

```
b := ( x < z ) AND ( z < 16 )
```

pondrá b a *true* si el valor de x es menor que el de z , y además éste es a su vez menor que 16.

Ejemplo 5.7

Si queremos leer por teclado la hora, minuto y segundo, habrá que verificar que esos datos que leemos corresponden a una hora válida. Alguien podría introducir los siguientes datos: 28

horas, 77 minutos y 15 segundos; que evidentemente no son válidos. ¿Cómo preguntaríamos en Pascal si la hora introducida es válida?. Pues bien, mediante una expresión booleana que sea *true* en ese caso, y *false* en caso contrario.

Una posible codificación en Pascal sería:

```
PROGRAM Reloj (input, output);
VAR
  hora, minuto, segundo : integer;
  horaValida : boolean;

BEGIN (* programa principal *)
  Writeln('Introduzca hora, minuto y segundo:');
  Readln (hora, minuto, segundo);
  horaValida := (hora >= 0) AND (hora < 24) AND (minuto >= 0)
  AND (minuto < 60) AND (segundo >= 0)
  AND (segundo < 60);
  Writeln(horaValida)
END.
```

5.4 TIPO CARACTER

Antes de pasar a definir las constantes y variables de tipo carácter, se recordará la definición de los conjuntos de caracteres.

CONJUNTOS DE CARACTERES

Actualmente los ordenadores han dejado de ser considerados únicamente como potentes máquinas de "calcular", como ocurría en los primeros tiempos. Sus campos de aplicación van mucho más allá del cálculo numérico, saliendo de los entornos puramente matemáticos.

El procesamiento de información en forma escrita (no numérica) es en la actualidad de capital importancia. Sin ir más lejos, la comunicación hombre-máquina se realiza casi exclusivamente mediante cadenas de caracteres.

Cada máquina trabaja con un conjunto de caracteres alfanuméricos que pueden ser representados en ella. Este conjunto de caracteres varía de una máquina a otra, siendo los dos más importantes el **ASCII** (*American Standard Code for Information Interchange*) y el **EBCDIC** (*Extended Binary Code Decimal for Information Coded*), como ya se mostró en el capítulo 1 (véase anexo I).

El primero de ellos consta de 128 caracteres, y el segundo de 256. En el Anexo I se muestran ambos repertorios de caracteres. Como se puede apreciar, ambos incluyen letras mayúsculas y minúsculas, dígitos, signos de puntuación,... en general todos aquellos que pueden encontrarse en el teclado de una máquina de escribir además de una serie de caracteres de control no imprimibles. Las características de los dos códigos son las que a continuación se detallan.

•ASCII (*American Standard Code for Information Interchange*)

Cada carácter individual está codificado numéricamente por una combinación única de 7 bits (por tanto tiene $2^7 = 128$ caracteres diferentes).

TIPO CARACTER

En la tabla del Anexo I puede verse como los caracteres además de *codificados* están *ordenados*. Los dígitos están ordenados según su propia secuencia numérica (de 0 a 9), y las letras están colocadas consecutivamente según su propio orden alfabético. Esto permite aplicar reglas de sucesión y precedencia al ser un conjunto ordenado.

Ejemplo 5.8

La 'K' se codifica con el 75 mientras que la 'L' se codifica con el 76. Se dice que 'K' precede a 'L' o que 'L' sucede a 'K'.

•EBCDIC (*Extended Binary-Coded Decimal Interchange Code*)

Se utiliza en los ordenadores IBM (a excepción de la gama personal). Se trata de un esquema de codificación de 8 bits y tiene por lo tanto 256 caracteres ($2^8 = 256$). El conjunto de caracteres EBCDIC es independiente e incompatible con el código ASCII.

El código EBCDIC también es un *conjunto ordenado*.

CONSTANTES Y VARIABLES DE TIPO CARACTER

Según se ha indicado antes, el proceso de la información que está en forma de caracteres es muy importante. Para este fin el Pascal define el tipo de datos *char*. Los valores de las constantes y variables de este tipo de datos serán caracteres aislados del conjunto de caracteres de la máquina.

Una *constante* de tipo *carácter* se representa encerrando el carácter en cuestión entre comillas simples:

'A' '5' '='

el compilador necesita las comillas para diferenciar, por ejemplo, el carácter '5' y el número entero 5 o el carácter 'A' y el identificador A.

El carácter '5' (o cualquier otro dígito) no tiene ningún significado numérico. No se pueden sumar '5' + '4' pues son caracteres. También hay que recordar que el carácter *blanco* es un carácter como otro cualquiera. Su representación es: ' '.

Aparece un problema: si los caracteres se representan encerrados entre comillas, ¿cómo se representa el carácter comilla sin ambigüedad? Pues bien, en este caso se duplica la comilla.

Se pueden definir constantes de tipo carácter en la sección CONST, de la manera que se muestra a continuación:

Ejemplo 5.9

```
CONST
blanco = ' ';
cruz = ' + ';
asterisco = ' * ';
siete = '7';
```

TIPOS DE DATOS SIMPLES

```
zeta= 'z ' ;
abrirInterrogacion = '¿' ;
cerrarInterrogacion = '?' ;
afirmativo = 'S' ;
negativo = 'N' ;
letraE = 'E' ;
comilla = ' ' ; (* caso especial mencionado antes *)
```

Los identificadores blanco, cruz, asterisco, siete, zeta, abrirInterrogacion, cerrarInterrogacion, afirmativo, negativo, letraE y comilla representan a ' ', '+', '*', '7', 'z', '¿', '?', 'S', 'N', 'E' y comilla simple.

Las *variables* de tipo *char* se declaran, igual que el resto de variables, en la sección VAR del programa:

```
VAR
  letra, ch : char ;
  digito : char ;
  signo : char ;
  punto : char ;
```

a las cuales se les pueden asignar valores mediante sentencias de asignación tales como:

```
letra := 'm' ;
ch := blanco;
```

siendo *blanco* la constante definida en el ejemplo anterior.

Cada variable carácter sólo puede contener un carácter en cada instante de la ejecución del programa.

También pueden tomar valores externos por medio de una sentencia de lectura *Read* o *Readln*. Así,

```
Read (ch);
```

leería un carácter del fichero *input* (teclado) y lo almacenaría en la variable *ch*.

Tanto las constantes como las variables de tipo *char* pueden imprimirse mediante las sentencias *Write* y *Writeln*.

```
Write (blanco);    escribiría un carácter blanco.
Write (ch);        escribiría el contenido de la variable ch.
```

También es posible formatear la salida, es decir, conseguir que los datos estén en determinadas columnas.

```
Write (ch:8);     escribiría el contenido de la variable ch
                  en un campo de longitud 8, es decir, precedida por 7 blancos.
```

EXPRESIONES Y FUNCIONES RELACIONADAS CON CARACTERES

El conjunto de caracteres de una máquina está ordenado como puede comprobarse observando el Apéndice I. Aunque el orden difiere de un código a otro, en particular se observa que

TIPO CARACTER

las letras se disponen en orden alfabético de manera que 'a' < 'b' < 'c' < etc... tanto en ASCII como en EBCDIC; y lo mismo ocurre en otros códigos. Eso siempre se verifica, sin embargo en algunos conjuntos de caracteres como el EBCDIC las letras no van contiguas. Hay caracteres de control no imprimibles intercalados entre ellas.

En los ejemplos de este texto supondremos que el código usado es el ASCII en el cual las letras sí van contiguas.

También sucede que '0' < '1' < '2' < ... < '9' en ambos códigos.

En virtud de la ordenación anterior, a cada carácter se le asigna un número de orden que constituye el *ordinal* de dicho carácter en el conjunto de caracteres.

Así pues, se pueden comparar entre sí constantes y/o variables del tipo *char* mediante los operadores de relación vistos anteriormente.

Las siguientes comparaciones darían un resultado *true*:

'a' < 'z' 'b' <= 'e'
'A' <> 'a' '?' = '?'

Es importante considerar no obstante, que una misma comparación en distintas máquinas pueden dar resultados diferentes.

FUNCIONES PREDEFINIDAS DE TIPO CARACTER

El lenguaje Pascal incorpora cuatro funciones relacionadas con caracteres: *Ord*, *Chr*, *Pred* y *Succ*.

Función	Acción	Tipo de Argumento(x)	Tipo de Resultado
Pred(x)	Determina el predecesor de x	char	char
Succ(x)	Determina el sucesor de x	char	char
Chr(x)	Determina el carácter cuyo código es x	integer	char
Ord(x)	Determina el entero que se usa para codificar el carácter x	char	integer

Tabla 5.5 Funciones de tipo carácter

Como ejemplo, (suponiendo que se trabaja en ASCII) se tiene que:

Llamada	Valor devuelto
Ord ('A')	65 (integer)
Chr (35)	'#' (char)
Pred ('g')	'f' (char)
Succ ('f')	'g' (char)

Las funciones *Ord* y *Chr* son inversas entre sí, de manera que:

Ord (Chr(15)) devuelve 15 (integer)
 Chr (Ord('a')) devuelve 'a' (char)

5.5 TIPO REAL

El tipo real como ya se estudió en el capítulo 3, está constituido por los números reales racionales. Para se estudio más detallado se comenzará viendo el concepto de rango y precisión.

RANGO

Sea un ordenador *simplificado*, tal que cada palabra tiene asignado un número de posiciones de memoria, de tal forma que un número entero se representa por un signo y cinco dígitos decimales (base 10). Se recuerda, que los ordenadores reales sólo trabajan en binario (no en base 10). En este caso el **rango** que se puede representar con cinco dígitos es:

+ 9 9 9 9 9 Mayor entero positivo
 + 0 0 0 0 0 Cero
 - 9 9 9 9 9 Menor entero negativo

PRECISION

En este ejemplo la precisión es de 5 dígitos, y todo entero que no tenga más de cinco dígitos se puede representar exactamente.

Todo esto es evidente cuando se manejan datos de tipo entero, pero veamos que sucede cuando se manejan datos de tipo real.

Supongamos que uno de los cinco dígitos del número, representa al exponente, por ejemplo el dígito situado más a la izquierda.

+ 2 7 8 9 3
 ↑
 exponente

Aquí tenemos representado el número

TIPO REAL

$$+ 7893 \times 10^2$$

Con esta nueva representación el *rango* de números que pueden representarse considerando:

$$\begin{array}{ll} + 9999 & \text{Mayor número real positivo} \\ - 9999 & \text{Menor número real negativo} \end{array}$$

es desde

$$-9999 \times 10^9 \quad \text{a} \quad 9999 \times 10^9$$

o escrito de otra forma

$$-9\,999\,000\,000\,000 \quad \text{a} \quad +9\,999\,000\,000\,000$$

El rango ha aumentado, pero la *precisión* ha disminuido, y ahora sólo es de 4 dígitos. Es decir sólo se pueden representar 4 dígitos que no sean ceros.

Con este tipo de representación sólo se muestra exactamente cualquier número con 4 dígitos no nulos.

¿Qué sucede con números de más de cuatro dígitos?

Los 4 primeros dígitos serán correctos, y el resto de los dígitos se sustituyen por ceros.

Ejemplo 5.10

número	representación	valor
84321	+18432	$+8432 \times 10^1$
973250	+29732	$+9732 \times 10^2$
-999999	-29999	-9999×10^2
-123456789	-51234	-1234×10^5
1000000	+31000	$+1000 \times 10^3$
-99999	-19999	-9999×10^1
-27272727	-42727	-2727×10^4

Continúa siendo el primer dígito de la representación el exponente.

En el ejemplo anterior se puede observar la diferencia entre el número a representar y el valor representado.

No siempre ocurre esto, en la siguiente tabla se representa exactamente 1 000 000 pero no el resto de los números.

número	valor representado
84321	+84320
973250	+973200
-999999	-999900
-123456789	-123400000
1000000	+1000000
-99999	-99990
-27272727	-27270000

En este ejemplo se ha supuesto un esquema de codificación que sólo tiene 4 dígitos significativos. Se dice que *la precisión es de cuatro dígitos*. Entonces se puede definir precisión.

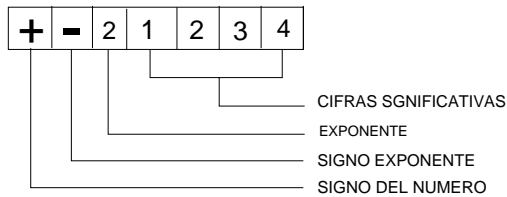
Precisión es el número máximo de dígitos significativos.

El esquema de codificación anterior se puede *extender*, para que se puedan representar números reales con exponente negativo.

Ejemplo 5.11

1234×10^{-2} su valor es 12.34
 77×10^{-2} su valor es 0.0077

Se puede proponer un *nuevo esquema de codificación* de la forma:



Con este nuevo esquema de codificación, se tiene que:

++9999 Mayor nº positivo = $+9999 \times 10^{+9}$
+-90001 Menor nº positivo = $+1 \times 10^{-9}$
--90001 Mayor nº negativo = -1×10^{-9}
-+99999 Menor nº negativo = $-9999 \times 10^{+9}$

Gracias a que hemos podido modificar el esquema de codificación, añadiendo exponentes negativos, se pueden representar *números fraccionarios*.

TIPO REAL

Ejemplo 5.12

número	codificación	valor
0.1234	+ - 4 1 2 3 4	+1234×10 ⁻⁴
0.12345	+ - 4 1 2 3 4	+1234×10 ⁻⁴
-0.005	-- 6 5 0 0 0	-5000×10 ⁻⁶
-5.3030	-- 3 5 3 0 3	-5303×10 ⁻³
7654.3210	+ - 0 7 6 5 4	+7654×10 ⁻⁰
123.4567	+ - 1 1 2 3 4	+1234×10 ⁻¹

Obsérvese que se sigue teniendo una precisión de 4 dígitos, por lo cual existe una diferencia entre el número a codificar y el valor en memoria. Así en el ejemplo anterior se tiene:

número	valor representado
0.1234	+0.1234
0.12345	+0.1234
-0.005	-0.005
-5.3030	-5.3030
123.4567	+123.4000
7654.3210	+76540000

ERROR REPRESENTACIONAL

En los ejemplos anteriores puede observarse que existe un **límite de magnitud** de los números reales para poder almacenarse en un ordenador con un esquema de codificación dado. En el ejemplo anterior los límites son:

$$+9999 \times 10^{+9} \quad \text{y} \quad -9999 \times 10^{+9}$$

También existe un **límite de precisión** de los números reales, para un esquema de codificación determinado. En el ejemplo anterior la precisión es de 4 dígitos.

En los ordenadores los números se codifican en binario (en base 2), pero el esquema de codificación es similar al que se acaba de ver. Así se ha estudiado en el capítulo correspondiente al tipo INTEGER, que en muchos ordenadores un número entero sin signo se almacena en una palabra de memoria de 16 bits (2 bytes).

BASE 2		BASE 10
0000000000000000	valor mínimo	0
1111111111111111	valor máximo	65535

TIPOS DE DATOS SIMPLES

Dado que los enteros se extienden entre $-MAXINT$ y $MAXINT$, su rango queda definido entre

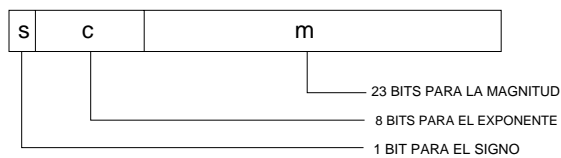
$$-32768 \quad \text{y} \quad 32767$$

Si los valores de tipo INTEGER se almacenan en una palabra de memoria de 32 bits (4 bytes o 4 octetos), el rango de los enteros queda definido entre

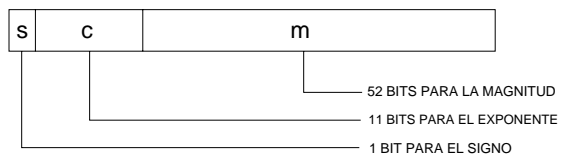
$$-2.147.483.648 \quad \text{y} \quad +2.147.483.647$$

Los números reales se almacenan en palabras de memoria de 32 bits (simple precisión) 64 bits (doble precisión) e incluso 80 bits (precisión ampliada).

El esquema de codificación de un número real en una palabra de 32 bits es el siguiente:



Si un número real se almacena en una palabra de 64 bits, el esquema de codificación es el siguiente:



donde:

- m** es la mantisa o magnitud
- c** es el exponente o característica
- s** es el signo

NORMALIZACION IEEE 754

Es una normalización elaborada por un comité de expertos para representar números reales. Sus puntos más destacados son:

1.- El exponente se ajusta de forma que el 1 más significativo de la mantisa binaria quede justo a la izquierda del punto decimal y no se almacena quedando implícito en la representación interna. De esta forma:

$$1 \leq m < 2$$

2.- Cuando el exponente toma su valor máximo ($c = 255$) se presentan dos casos:

TIPO REAL

- 2.1.- si $m = 0$ más o menos infinito (desbordamiento ...)
2.2.- si $m < 0$ resultado no válido o símbolo no numérico.

3.- Cuando el exponente toma su valor mínimo ($c = 0$), el 1 más significativo de la mantisa no se supone implícito.

Según esta normalización y siguiendo el esquema de codificación de un número real N en simple y doble precisión queda representado de la siguiente manera:

Simple precisión

- a) si $c = 255$ y $m < 0$ N no representa un número
b) si $c = 255$ y $m = 0$ $N = (-1)^s \times \text{Infinito}$
c) si $0 < c < 255$ $N = (-1)^s \times 2^{c-127} \times 1.m$
d) si $c = 0$ y $m < 0$ $N = (-1)^s \times 2^{-126} \times 0.m$
e) si $c = 0$ y $m = 0$ $N = (-1)^s \times 0$

Doble precisión

- a) si $c = 2047$ y $m < 0$ N no representa un número
b) si $c = 2047$ y $m = 0$ $N = (-1)^s \times \text{Infinito}$
c) si $0 < c < 2047$ $N = (-1)^s \times 2^{c-1023} \times 1.m$
d) si $c = 0$ y $m < 0$ $N = (-1)^s \times 2^{-1022} \times 0.m$
e) si $c = 0$ y $m = 0$ $N = (-1)^s \times 0$

Como resumen de todo lo visto anteriormente se puede concluir:

a) Existe un **límite de magnitud** de los números reales. Este límite es variable de un ordenador a otro y de una implementación del lenguaje Pascal a otra. El lenguaje de programación Pascal no impone ningún límite de magnitud.

b) Existe un **límite de precisión** de los números reales. Este límite depende del ordenador y de la implementación del lenguaje Pascal. El lenguaje de programación Pascal no impone ningún límite de precisión.

c) Con la misma ocupación de memoria, al aumentar la precisión disminuye el rango y viceversa.

d) De las dos conclusiones anteriores se deduce que existe un **error representacional**. Esto se debe a que muchos números no se pueden representar exactamente en la memoria del ordenador por:

- ser irracionales.
- ser racionales que no tienen representación binaria exacta.
- tener excesivos números significativos.

ERRORES COMPUTACIONALES

Cuando se utiliza la aritmética entera los resultados son exactos. Sin embargo la aritmética real no es tan exacta.

Ejemplo 5.13

Continuando con el ordenador de 4 dígitos de precisión, sean los siguientes tres números reales

$$X = -1234 \times 10^3$$

$$Y = 1235 \times 10^3$$

$$Z = 5432 \times 10^0$$

los sumamos de la siguiente forma:

$$X + Y = 1 \times 10^3 = 1000 \times 10^0$$

$$(X + Y) + Z = 1000 \times 10^0 + 5432 \times 10^0 = 6432 \times 10^0$$

pero si los sumamos de esta otra forma:

$$Y + Z = 1235 \times 10^3 + 5432 \times 10^0 = 1240432 \times 10^0$$

como nuestra precisión es de 4 dígitos el ordenador sólo almacena 1240×10^3

entonces $X + (Y + Z) = -1234 \times 10^3 + 1240 \times 10^3 = 6000 \times 10^3$

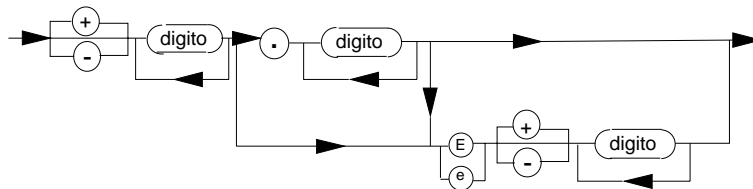
No se cumple aparentemente la propiedad asociativa de la suma. Los *errores representacionales* se propagan por *errores computacionales*.

EL TIPO REAL EN PASCAL

La sintaxis de un número real en notación EBNF se denota por:

```
<constante real> ::= <signo> <real sin signo> | <real sin signo>
<real sin signo> ::= <entero sin signo> . <dígito> {<dígito>} | <entero sin signo> . <dígito> ({<dígito>} | (<E | e> <constante entera>)) | <entero sin signo> (<E | e> <constante entera>)
```

Utilizando el diagrama sintáctico, la descripción formal es la siguiente:



El significado de E es "10 elevado a".

Ejemplos de números reales **correctamente** escritos en Pascal:

TIPO REAL

-41.0	0 E 0
57.0	2 E 3
27.3 E +7	73.1 E -5
1234 E 56	57.58
1234.56 E -7	58.57
1234.5 E 6	

Ejemplos de números reales **incorrectamente** escritos en lenguaje Pascal estándar:

21	es un entero
1. E12	es obligatorio escribir parte fraccionaria con punto
.2	es obligatorio escribir la parte entera y fraccionaria
2.3 E	falta el exponente
44165.	falta la parte fraccionaria
9.880,0	incorrección sintáctica

OPERADORES ARITMETICOS

Existen cinco operadores que pueden utilizarse con operandos de tipo real. Todos llevan dos operandos a excepción del operador que realiza el cambio de signo.

(-)	unario (cambio de signo)
+	adición
-	sustracción
*	multiplicación
/	división real.

Tabla 5.6 Operadores aritméticos con operandos reales

Los números reales no deben compararse para la igualdad ya que dos números reales es extraño que sean exactamente iguales. Se recurre a calcular la diferencia entre dos números y comparar el resultado con el operador < respecto a alguna cantidad muy pequeña. Por ejemplo $Abs(a - b) < 0.0000001$

JERARQUIA DE LOS OPERADORES ARITMETICOS

Indica los niveles de precedencia para los operadores aritméticos. Así con los operadores anteriores se establece la siguiente jerarquía:

- 1º) - unario
- 2º) * , /
- 3º) + , -

En el caso de que existan paréntesis se evalúan en primer lugar las expresiones incluidas en éstos.

COMPATIBILIDAD INTEGER-REAL

El valor de una expresión entera puede ser asignado a una variable *real*. Es una excepción de la correspondencia de tipos obligatoria en la sentencia de asignación.

DECLARACION DE CONSTANTES REALES

Se efectúa según el esquema siguiente:

```
CONST
    identificador = constante;
```

donde *constante* ha de ser un número *real* tal como se definió previamente el tipo *real*.

Ejemplo 5.14

```
CONST
    pi = 3.14159 ;
    raizDeDos = 1.4142135 ;
    numeron = 999.99 ;
    paston = 23 E 6 ;
    ridiculo = 1 E -9 ;
```

DECLARACION DE VARIABLES REALES

Se efectúa según el esquema siguiente:

```
VAR
    identificador : tipo;
```

Ejemplo 5.15

```
VAR    a, b, c, x1, x2 : real ;
       d, pr, pi : real ;

VAR    a, b, c, x1, x2, d, pr, pi : real ;
```

Observaciones:

- Las dos formas anteriores son equivalentes.
- El orden de declaración de las variables es indiferente, tanto si son del mismo tipo o de tipos diferentes:

TIPO REAL

Ejemplo 5.16

```
VAR
    a : real ;
    b : char ;
    c : integer ;
    d : boolean ;
    e : real ;
```

LECTURA DE VARIABLES REALES

Se realiza con las sentencias:

```
Read
Readln
```

de igual forma que para las variables de tipo entero.

Ejemplo 5.17

```
Readln ( a, b, c ) ;
```

si *a*, *b*, *c* son identificadores de variables reales se pueden teclear los siguientes valores separados por uno o más blancos:

```
7.0 5.7 -9E5
```

Observaciones:

- Puede darse un valor entero a una variable real, y ese entero se transforma en real, para ser almacenado en la variable.
- Una variable de tipo entero **no** puede almacenar un valor de tipo real.

ESCRITURA DE REALES

Las constantes, variables, funciones o expresiones de tipo real pueden aparecer como parámetros en las sentencias de escritura:

```
Write
Writeln
```

Escritura con formato

Los números reales se escriben con *formato exponencial*, siempre que no se especifique un formato determinado.

Ejemplo 5.18

```
a := 6.0;
Writeln (a);
```

La salida será:

```
6.00000 E + 00
```

es decir, el ordenador escribe antes del punto decimal un dígito, con o sin signo, seguido de tantas cifras decimales como precisión tenga la máquina, y el exponente.

Los números enteros también tienen posibilidad de ajustarse a distinto formato que el dado inicialmente por el ordenador.

Ejemplo 5.19

```

. . .
VAR
    a : integer;
. . .
a := 75;
. . .
Writeln (a);
. . .

```

La ejecución de este programa produce la salida en el formato inicial:

7	5
---	---

Sin embargo, si queremos que la salida ocupe **cuatro** espacios, se puede sustituir la sentencia de escritura por

```
Writeln (a:4)
```

Siendo la salida en este caso

	7	5
← 4 →		

La forma general de especificar el formato de variables, constantes o expresiones enteras es:

```
Write (variable1:m, variable2:m, . . . )
```

o

```
Writeln(variable1:m, variable2:m, . . . )
```

donde *m* es el nº total de espacios a ocupar.

Se puede fijar la forma de escritura de un número real mediante la *especificación del número de espacios* *m* en que se debe escribir el número real completo (parte entera y fraccionaria), y el número de espacios *n* que ocupa la parte decimal. La forma es la siguiente:

```
Write (variable1:m1:n1, variable2:m2:n2, . . . )
```

o

```
Writeln(variable1:m1:n1, variable2:m2:n2, . . . )
```


TIPO REAL

Ejemplo 5.20

```
PROGRAM Ejemplo (output) ;
CONST
  pi = 3.14159;
  po = 3.14 ;
VAR
  a, b : real ;
BEGIN
  a := 6 ;
  b := -2.4E -2 ;
  Writeln (pi:8:3, po:8:3, a * a: 10:2 );
  Writeln ( b:9:5 , 7.0);
  Writeln( b, ' ' , a)
END.
```

El resultado de la ejecución de este programa es:

3.142	3.140	36.00
-0.02400	7.0000000000E+00	
-2.4000000000E-02	6.0000000000E+00	

FUNCIONES REALES INCORPORADAS

El lenguaje Pascal dispone de las siguientes funciones estándar en las que el argumento x es del tipo real, pudiendo ocupar este lugar las constantes, variables, funciones y expresiones reales.

Función	Resultado
Abs(x)	El valor absoluto de x
Sqr(x)	El cuadrado de x
Sqrt(x)	La raíz cuadrada de x
Exp(x)	El número e elevado a x
Ln(x)	El logaritmo neperiano de x
Sin(x)	El seno de x (x en radianes)
Cos(x)	El coseno de x (x en radianes)
ArcTan(x)	El arcotangente de x , en radianes
Round(x)	El entero más cercano a x . Redondea. El resultado es un <i>entero</i> .
Trunc(x)	Trunca el número real x . El resultado es un <i>entero</i> .

Tabla 5.7 Funciones estándar

Ejemplo 5.21

Aplicación de la función	Resultado
Round (6.7)	7 (entero)
Round (-4.37)	-4 (entero)
Trunc (-4.7)	-4 (entero)
Round (-4.7)	-5 (entero)
Abs (-6.78)	6.78
Sqrt(9)	3.0
Sin (0.0)	0.0
Cos (0.0)	1.0
ArcTan (0.0)	0.0
ArcTan (1.0)	0.785
Ln (1.0)	0.0
Exp (0.0)	1.0
Sqr (2.0)	4.0

El lenguaje Pascal *no dispone* como función estándar de la *función potenciación*, pero se puede realizar de la siguiente forma:

$$z = a^b \rightarrow \text{Ln } z = b \text{ Ln } a \rightarrow z = e^{b \text{Ln } a}$$

que en lenguaje Pascal se puede expresar como:

$$\text{Exp} (b * \text{Ln} (a))$$

El lenguaje Pascal *no dispone* tampoco de la función estándar *logaritmo decimal*, pero se puede calcular por

$$y = \log_{10} x \rightarrow 10^y = x \rightarrow y \text{ Ln } 10 = \text{Ln } x \rightarrow y = \text{Ln}(x) / \text{Ln}(10)$$

en lenguaje Pascal:

$$\text{Ln} (x) / \text{Ln} (10)$$

El lenguaje Pascal no dispone tampoco de la función trigonométrica $\text{tg}(x)$. Evidentemente se puede obtener como:

$$\text{Sin} (x) / \text{Cos} (x)$$

5.6 TIPOS DEFINIDOS POR EL USUARIO

Además de los tipos predefinidos citados en las secciones anteriores, existen otros tipos definidos: *escalares*, *ordinales* y *enumerados*.

TIPOS DEFINIDOS POR EL USUARIO

TIPOS ESCALARES Y TIPOS ORDINALES

El tipo de una constante o variable determina los valores que ésta puede tomar. Ya hemos visto los tipos incorporados *integer*, *boolean*, *char* y *real*. Pues bien, se dice que un tipo de datos es **escalar** si los valores que constituyen ese tipo de datos *están ordenados*.

Según esta definición los cuatro tipos de datos incorporados en Pascal son escalares pues

```
1 < 2 < 3 < 4...      ( integer )
false < true          ( boolean)
'a' < 'b' < 'c' ...   ( char )
1.0 < 1.5 < 2.4 ...  ( real )
```

Un tipo de datos escalar se dice que es **ordinal** si cada valor, excepto el primero, tiene *un único predecesor* y cada valor, excepto el último, tiene *un único sucesor*.

Por lo tanto todos los tipos escalares con excepción del tipo real, son tipos ordinales.

Así por ejemplo, el predecesor del número entero 1 es el 0; y su sucesor es el 2. Sin embargo no existe un único predecesor o sucesor de un número real.

El Pascal incorpora dos funciones *Pred* y *Succ* que devuelven el predecesor y el sucesor, respectivamente, del valor que se le pasa como parámetro. El tipo del parámetro debe ser un tipo ordinal, siendo el resultado devuelto del mismo tipo.

Así por ejemplo:

```
Pred ( 5 )      devuelve 4
Succ ( 'A' )    devuelve 'B'
Pred (true)    devuelve false
```

Obsérvese que *Pred (-maxint)* y *Succ (maxint)* no están definidos y se produciría un error.

TIPOS ENUMERADOS

El Pascal, además de los cuatro tipos de datos predefinidos, permite que se puedan crear nuevos tipos de datos *a medida* de las necesidades del usuario. Si, por ejemplo, estamos desarrollando un programa que simula un juego de cartas, nos interesaría disponer de un tipo de datos cuyos valores fueran: *oros*, *copas*, *espadas* y *bastos*.

Esto puede hacerse mediante la siguiente definición de un tipo enumerado en la sección **TYPE** del programa.

```
TYPE
    palos = ( oros, copas, espadas, bastos );
```

el identificador *palos* da nombre al nuevo tipo de datos que se acaba de crear, y los identificadores *oros*, *copas*, *espadas* y *bastos* representan los valores que componen el nuevo tipo de datos.

TIPOS DE DATOS SIMPLES

Según la sintaxis de esta definición de tipo, después del identificador que da nombre al tipo, debe ir el signo '=' y a continuación encerrados entre paréntesis y separados por comas ',' se enumeran (de ahí el nombre de tipos *enumerados*) los valores constantes que componen ese tipo de datos. Tras el paréntesis de cierre debe ponerse un ';'

Otras definiciones de tipos enumerados podrían ser:

```
TYPE
poker      = ( corazones, diamante, picas, trebol );
moneda     = ( cara, cruz );
semaforo   = ( rojo, ambar, verde);
diaSemana  = (lunes, martes, miercoles, jueves, viernes, sabado,
              domingo);
color      = (blanco, rojo, azul, negro, verde);
figura     = (rectangulo, cuadrado, circulo, rombo);
animal     = (raton, gallina, perro, gato);
mes        = (enero, febrero, marzo, abril, mayo, junio, julio,
              agosto,septiembre, octubre, noviembre, diciembre);
anio       = (normal, bisiesto);
```

`poker`, `moneda`, `diaSemana`, `semaforo`, `color`, `figura`, `animal`, `mes`, y `anio` son los indicadores de tipo que nos permitirían declarar variables como:

```
VAR
a, b           :moneda;
semf1, semf2   :semaforo;
carta          :poker;
hoy, maniana, pasadoManiana :diaSemana;
pintura, fondo :color;
formal, forma2 :figura;
eligeTu, elijoYo :animal;
anual          :anio;
```

Otra definición sería:

```
VAR
hoy,maniana,pasadoManiana :(lunes, martes, miercoles, jueves,
                             viernes, sabado, domingo);
pintura,fondo              :(blanco, negro, azul, rojo, verde);
formal, forma2             :(rectangulo, cuadrado, circulo, rombo);
eligeTu, elijoYo           :(raton, gallina, perro, gato);
fechal, fecha2             :(enero, febrero, marzo, abril, mayo, junio,
                             julio, agosto, septiembre, octubre, noviembre,
                             diciembre);
anual                      :(normal, bisiesto);
```

Los tipos enumerados son también ordinales. La relación de orden de sus valores se establece implícitamente en su definición. Por ejemplo en el tipo "palos" definido anteriormente se verifica que:

```
oros < copas < espadas < bastos
```

Por lo tanto podemos aplicar los operadores relacionales para comparar variables y constantes de estos tipos, así como las funciones *Ord*, *Pred* y *Succ*.

TIPOS DEFINIDOS POR EL USUARIO

Ejemplo 5.22

Considerando los tipos enumerados definidos anteriormente, se tienen las expresiones booleanas:

Expresion	Valor
lunes < martes	true
martes < miercoles	true
jueves < viernes	true
sabado < domingo	true
miercoles = sabado	false
lunes <> viernes	true
miercoles > jueves	false
jueves <= viernes	true
jueves = viernes	false

Es decir el orden de los valores está especificado por su orden de enumeración en la definición de *TYPE*.

Continuando con el ejemplo de la declaración de tipos enumerados en la siguiente tabla se recogen varias aplicaciones de la función *Ord* así como el resultado de cada llamada.

Aplicación de la función	Resultado
Ord(oros)	0
Ord(copas)	1
Ord(espadas)	2
Ord(bastos)	3
Ord(lunes)	0
Ord(martes)	1
Ord(miercoles)	2
Ord(jueves)	3
Ord(viernes)	4
Ord(sabado)	5
Ord(domingo)	6

La función *Chr(x)* sólo es aplicable al tipo *char*.

La siguiente tabla recoge varias llamadas de las funciones *Succ* y *Pred* según la defenición de tipos dada anteriormente.

Aplicación de la función	Resultado
Succ (lunes)	martes
Pred (martes)	lunes
Succ (miercoles)	jueves
Pred (domingo)	sabado
Succ (domingo)	* * No definido * *
Pred (lunes)	* * No definido * *
Succ (jueves)	viernes
Pred (jueves)	miercoles

En algunas aplicaciones interesa que se pase automáticamente del último valor del tipo enumerado al primero. Esto se logra por medio de una sentencia condicional. Las sentencias condicionales, se estudiarán en el capítulo siguiente pero adelantamos aquí un sencillo ejemplo de su utilización.

Ejemplo 5.23

Usando la declaración de tipo enumerado del primer ejemplo:

```
IF hoy = domingo
  THEN maniana := lunes
  ELSE maniana := Succ (hoy);
```

Observaciones sobre los tipos enumerados

- Las variables declaradas como de tipo enumerado **no** se pueden introducir en el ordenador mediante sentencias de lectura:

```
Read
Readln
```

- Tampoco se pueden escribir mediante las sentencias:

```
Write
Writeln
```

- Los identificadores que forman el tipo enumerado deben ser únicos, y no pueden ser utilizados para definir otro tipo *enumerado*, u otro tipo de datos.

Ejemplo 5.24

```
VAR    vocal : ('A','E','I','O','U') ;
```

No es correcto pues las constantes que forman el tipo son parte del tipo estándar *char*.

```
VAR    dialaborable : (lunes,martes,miercoles,jueves,viernes,sabado) ;
       diaFestivo : (sabado,domingo) ;
```

No es correcto pues *sabado* se utiliza en dos tipos diferentes.

TIPOS DEFINIDOS POR EL USUARIO

Como acabamos de ver, los valores de tipo enumerado no se pueden leer ni escribir. Tienen existencia únicamente dentro del programa. Para poder leerlos o escribirlos deberemos hacerlo a través de otro tipo de variables, como *integer* o *char*. Por ejemplo, el siguiente fragmento de programa lee una variable *p* de tipo *palos* a través de otra *ch* de tipo *char*:

```
VAR p: palos;
    ch: char;
...
Write ('Escriba la inicial del palo:');
Readln(ch);
IF ch = 'o'
    THEN p:= oros
    ELSE
        IF ch = 'c'
            THEN p:= copas
            ELSE
                IF ch = 'e'
                    THEN p:= espadas
                    ELSE
                        IF ch = 'b'
                            THEN p:= bastos ;
...

```

De forma similar se procedería para escribir variables de estos tipos. En el capítulo sexto se verá un ejemplo utilizando la sentencia *CASE*, mucho más cómoda para estos menesteres, que sustituye a los *IF-THEN-ELSE* anidados.

TIPOS SUBRANGO

En ocasiones sólo es necesario un tipo de datos compuesto de determinado rango, o intervalo, de cualquier tipo ordinal estándar o definido por el usuario.

Tal es el caso, por ejemplo, cuando trabajamos con una variable que va a representar el día del mes. Podríamos utilizar una de tipo *integer*; pero dado que el día del mes nunca va a ser menor que 1 ni mayor que 31 sería más conveniente limitar los valores que puede tomar a ese subrango de *integer*, y no permitir días tales como -5 ó 32 .

Los tipos subrango pueden definirse en la sección *VAR* especificando el primer valor y el último separados con dos puntos '..' (el primer valor debe ser menor que el último).

Para el caso discutido antes se puede poner:

```
VAR
    dia : 1..31 ;
```

A los tipos subrango, igual que ocurría con los enumerados, también podemos asignarle un nombre en la sección *TYPE* y luego usarlo en la sección *VAR*.

La siguiente definición produce los mismos resultados que la presentada arriba:

```
TYPE
    diasMes = 1..31; (* definición del tipo subrango *)
VAR
    dia : diasMes;
```

Otras definiciones de tipo subrango se ilustran a continuación. *No están permitidos subrangos de tipo real.*

```

TYPE
  notaExamen = 0..10 ;
  minusculas = 'a'..'z' ;
  positivos = 0..maxint ;
  diaSemana = (lunes, martes, miercoles, jueves, viernes, sabado,
  domingo);
  diaLaborable = (lunes .. viernes);
  mes = 1..31 ;
  mayusculas = 'A' .. 'Z';
VAR
  hoy, maniana, pasadoManiana : diaSemana;
  diaClaseAlgebra, diaClasePascal : diaLaborable;
  hoyMes, examenMes : mes ;
  inicial1 , inicial2, inicial3 : mayusculas;

```

El ejemplo anterior también puede escribirse con notación abreviada:

```

TYPE
  diasemana = (lunes, martes, miercoles, jueves, viernes, sabado,
  domingo);
VAR
  diaclaseAlgebra, diaClasePascal : lunes .. viernes;
  hoyMes, examen : 1 .. 31;
  inicial1, inicial2, inicial3 : 'A'..'Z' ;

```

Como se puede apreciar, los tipos subrango mejoran la legibilidad del programa a la vez que pueden ahorrar memoria, ya que una variable del subrango *notaExamen* definido antes puede ocupar un solo byte, mientras que si se declarara como integer ocuparía dos o más bytes.

Por otra parte, los tipos subrango, se aprovechan de la comprobación automática de rango. Es decir, cada vez que se le asigna un valor a una variable (en tiempo de ejecución), el sistema comprueba si ese valor está dentro del rango, y si no es así devuelve un mensaje de error.

Dado que los datos de tipo subrango están ordenados, se pueden usar con *operadores relacionales* para formar expresiones booleanas.

Ejemplo 5.25

Expresión booleana	resultado
lunes < martes	true
31 > 1	true
'A' > 'Z'	false

También se pueden usar las funciones intrínsecas *Pred(x)*, *Succ(x)* y *Ord(x)*

AMPLIACION DEL PASCAL ESTANDAR CON TURBO PASCAL

Ejemplos 5.26

Aplicación de la función	resultado
Pred (martes)	lunes
Succ (martes)	miercoles
Ord (martes)	1
Ord (lunes)	0

5.7 AMPLIACION DEL PASCAL ESTANDAR CON TURBO PASCAL

En la tabla 5.8 se muestran los cinco tipos enteros predefinidos denotando cada uno un subconjunto específico de los números enteros.

Tipo de dato	Rango
Byte	0 .. 225
Word	0 .. 65535
ShortInt	-128 .. 127
Integer	-32768 .. 32767
LongInt	-2147483648 .. 2147483647

Tabla 5.8 Tipos enteros predefinidos en Turbo Pascal

Los tipos de operandos para el operador *Xor* pueden ser enteros o booleanos y el resultado es de tipo *boolean*.

En la tabla 5.9 se muestran los cinco tipos reales predefinidos. Estos difieren en el rango y precisión de los valores que almacenan.

Tipo de dato	Rango positivo	Dígitos significativos
Real	$2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$	11-12
Single	$1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$	7-8
Double	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15-16
Extended	$3.4 \times 10^{-4932} \dots 1.1 \times 10^{4932}$	19-20
Comp	$-2^{63} + 1 \dots 2^{63} - 1$	19-20

Tabla 5.9 Tipos reales predefinidos en Turbo Pascal

5.8 CUESTIONES Y EJERCICIOS RESUELTOS

- 5.1 Escribir una expresión booleana que dada la fecha de nacimiento de una persona, indique si es Aries o no.

Solución

Si se considera que las variables `dia` y `mes` indican el día y mes de nacimiento de una persona se obtiene:

```
((dia >= 21) AND (mes = 3) AND (dia <= 31)) OR
((dia <= 22) AND (mes = 4) AND (dia > 0))
```

- 5.2 Indicar las asignaciones incorrectas en el siguiente fragmento de programa teniendo en cuenta las declaraciones.

```
...
TYPE
    Color = (Rojo, Verde, Azul, Rosa, Purpura, Amarillo, Naranja);
    Basico = (Rojo .. Azul);
    dia = 1 .. 7;
VAR
    dado, cubo, lapiz : Basico;
    a : dia;
    b : Basico;
BEGIN
    dado := Succ (Rojo);
    cubo := Succ (Naranja);
    a := Ord(Amarillo);
    Read(b);
    lapiz:=Succ(Rosa);
    ...
END.
```

Solución

<code>cubo := succ (Naranja);</code>	El valor asignado está indefinido
<code>Read(b);</code>	No se pueden leer variables de tipo enumerado
<code>lapiz:=Succ(Rosa);</code>	lapiz es de tipo Basico y no de tipo Color

- 5.3 ¿Qué escribe en Output (pantalla del ordenador) el programa siguiente ?

```
PROGRAM VolumenEsfera (Output);
CONST
    CuatroTerciosPorPi = 4.1887902;
VAR
    Volumen, Radio : Real;
BEGIN
    Radio := 1.2;
    Writeln(' Para radio = 1.2 ');
    Volumen := CuatroTerciosPorPi * Radio * Radio * Radio;
    Writeln(' El volumen es: ', Volumen:8:2);
END.
```

CUESTIONES Y EJERCICIOS PROPUESTOS

Solución

Para radio = 1.2
El volumen es: 7.23

5.4 Escribir las declaraciones necesarias para el siguiente fragmento de programa.

```
...
BEGIN
  DistanciaPuntos := 100.3;
  Fondo := 2;
  Zoom := true;
  Numero := Fondo / 6;
  Writeln (DistanciaPuntos * Numero);
END.
```

Solución

```
VAR
  DistanciaPuntos, Numero : Real;
  Fondo : Integer;
  Zoom : Boolean;
```

5.5 Escribir un programa que determine si un año es bisiesto o no. Será bisiesto si es divisible por cuatro y no es divisible por cien, o es divisible por cuatrocientos.

Solución

```
PROGRAM Bisiesto (Input, Output);
VAR
  anio : Integer;
  bis : boolean;
BEGIN
  Write(' EL AÑO ');
  Readln(anio);
  Writeln(anio);
  bis:=(anio MOD 4 = 0) AND (anio MOD 100 <> 0) OR (anio MOD 400=0);
  Writeln(bis);
END.
```

5.9 CUESTIONES Y EJERCICIOS PROPUESTOS

5.6 ¿ Cómo se almacenan los números enteros en un ordenador ?

5.7 Deducir el valor de maxint en un compilador de PASCAL que utilice dos bytes para almacenar los valores de tipo entero.

5.8 Escribir un programa en Pascal que escriba el valor de Maxint y - Maxint.

TIPOS DE DATOS SIMPLES

5.9 Indicar el orden de evaluación de la siguiente expresión y calcular su resultado.

`Succ((25 + 30 * (2 - 8 * 5) * 25 + 1) DIV 2)`

5.10 Escribir un programa que determine el valor de la expresión anterior.

5.11 Escribir una expresión booleana que compruebe si una fecha es válida.

5.12 Escribir un programa que lea una fecha y compruebe si es válida utilizando expresiones booleanas.

5.13 ¿ Cuántos caracteres contiene una variable de tipo carácter ?

5.14 Definir rango y precisión.

5.15 Explicar la razón por la cual el tipo real no es un tipo ordinal, pero sí escalar.

5.16 Indicar la diferencia entre tipos enumerados y tipo subrango.

5.17 Escribir un programa que dada la fecha de nacimiento de una persona, indique si es Aries o no.

5.18 Enumerar tres números reales que no se puedan representar exactamente en el ordenador.

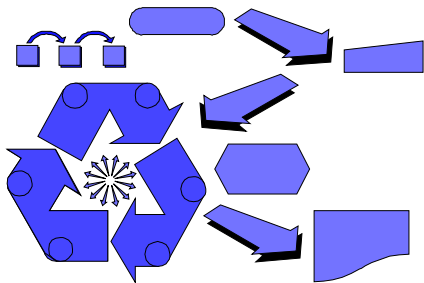
5.19 El rango de los números enteros y reales ¿Depende de la longitud de palabra del ordenador? ¿ Y la precisión?

5.20 ¿Cuál es la diferencia entre los tipos de datos escalares y los tipos de datos ordinales?

5.10 AMPLIACIONES y NOTAS BIBLIOGRAFICAS

En este capítulo se ha hecho una pequeña introducción al *Algebra de Boole*. Para un estudio más profundo y que entrañe un rigor matemático puede consultarse la obra *Lógica y algoritmos* de Robert R. Korfhage (*Limusa-Wiley, 1970*).

Si se desea ampliar en los conceptos: rango, precisión, error representacional y normalización IEEE 745 puede consultarse el libro *Introducción a la Informática* de A. Prieto, Antonio LLoris y Juan Carlos Torres (*McGraw Hill, 1989*)



CAPITULO 6

ESTRUCTURAS DE CONTROL

CONTENIDOS

- 6.1 Introducción
- 6.2 La estructura repetitiva *WHILE*
- 6.3 La estructura alternativa *IF-THEN*
- 6.4 La estructura alternativa *IF-THEN-ELSE*
- 6.5 La estructura repetitiva *FOR*
- 6.6 La estructura repetitiva *REPEAT-UNTIL*
- 6.7 Tratamiento secuencial de la información
- 6.8 Estructura multialternativa *CASE*
- 6.9 Sentencia *GOTO*
- 6.10 Aplicación al Cálculo Numérico. Determinación de raíces de ecuaciones
- 6.11 Extensiones del compilador Turbo Pascal
- 6.12 Cuestiones y ejercicios resueltos
- 6.13 Cuestiones y ejercicios propuestos
- 6.14 Ampliaciones y notas bibliográficas

INTRODUCCION

6.1 INTRODUCCION

Las estructuras básicas de control en *programación estructurada*, tienen todas una propiedad común: *cada estructura tiene un sólo punto de entrada y un sólo punto de salida*. Algunos autores las llaman *Esquemas de Programas*.

Los programas bien estructurados se construyen a base de módulos cuya estructura básica es una de las tres siguientes: *secuencial, alternativa, y repetitiva*.

a) LA ESTRUCTURA SECUENCIAL

Corresponde a una sucesión ordenada de tratamientos o acciones, que se desarrollan una detrás de otra.

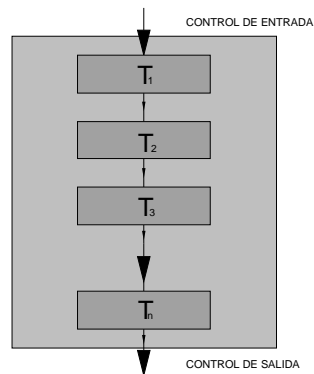


Figura 6.1 Organigrama de la estructura secuencial

Notación algorítmica

```
Acción1;  
Acción2;  
...  
AcciónN;
```

b) LA ESTRUCTURA ALTERNATIVA

Se puede plantear de tres formas diferentes: *la estructura alternativa simple, la estructura alternativa doble, y la estructura alternativa múltiple*.

- **La estructura alternativa simple**

Hace que una acción o tratamiento se ejecute o no, según se cumpla o no una condición . Se puede representar mediante el gráfico siguiente:

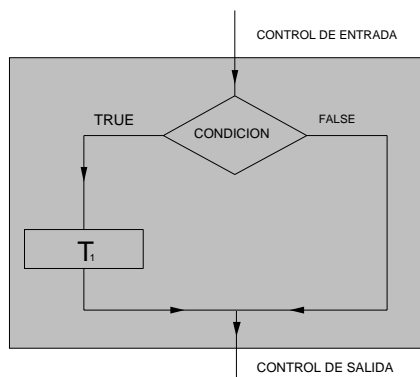


Figura 6.2 Organigrama de la estructura *IF-THEN*

A esta estructura también se la conoce, en Pascal, como *IF-THEN*.

Notación algorítmica

```
SI condición
  ENTONCES Tratamiento;
FIN_SI;
```

- **La estructura alternativa doble**

Contempla dos tratamientos tales que se excluyen mutuamente en función de una condición, es decir si se cumple la condición se ejecuta un tratamiento, y si no se cumple se ejecuta otro, debiéndose ejecutar uno y sólo uno de los tratamientos.

Su organigrama se muestra en la figura 6.3. A esta estructura de control también se la conoce, en Pascal, por *IF-THEN-ELSE*.

Notación algorítmica

```
SI condición
  ENTONCES Tratamiento1
  SI_NO    Tratamiento2;
FIN_SI;
```


INTRODUCCION

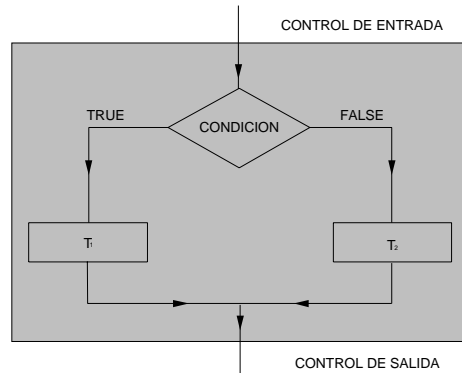


Figura 6.3 Organigrama de la estructura *IF-THEN-ELSE*

• La estructura multialternativa

Se presenta bajo la forma de un selector que puede tomar n valores, ejecutándose un tratamiento distinto según el valor que tome el selector. Se puede representar gráficamente de la siguiente forma:

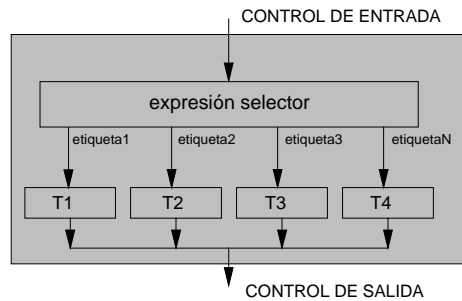


Figura 6.4 Organigrama de la estructura *CASE*

También se le denominará estructura *CASE*

Notación algorítmica

```
SEGUN selector HACER  
    ctel: Tratamiento1;
```

```

cte2: Tratamiento2;
cteN: TratamientoN;
FIN_SEGUN;

```

c) LA ESTRUCTURA REPETITIVA

Consiste en repetir la ejecución de un tratamiento un número de veces. Los tres esquemas de estructura repetitiva son: *mientras que*, *repite hasta que*, y *para hacer desde-hasta*.

• La estructura repetitiva WHILE-DO (mientras que)

Se ejecuta cero o más veces mientras se cumpla la condición. Se representa por el esquema de la figura 6.5.

Notación algorítmica

```

MIENTRAS condición HACER
    Tratamiento;
FIN_MIENTRAS;

```

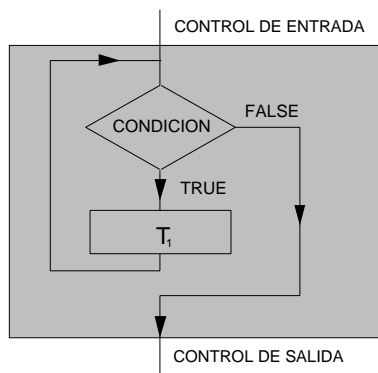


Figura 6.5 Organigrama de la estructura *WHILE*

• La estructura repetitiva REPEAT-UNTIL (repite-hasta que):

Se ejecuta el tratamiento siempre una vez, y se repite mientras la condición sea falsa. Se puede representar por el organigrama de la figura 6.6.

INTRODUCCION

Notación algorítmica

```
REPETIR  
  Tratamiento;  
HASTA condición;
```

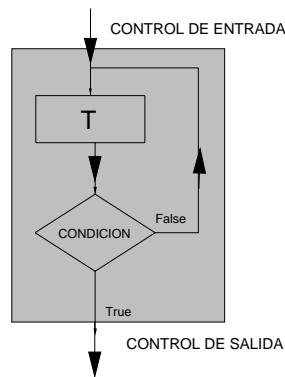


Figura 6.6 Organigrama de la estructura *REPEAT*

Las principales diferencias entre las dos estructuras repetitivas anteriores son:

- Las condiciones de salida del bucle son contrarias, esto es, en la estructura *WHILE* se abandona la ejecución del bucle cuando la condición no se cumple, sin embargo en la estructura *REPEAT* se abandona la ejecución del bucle cuando se cumple la condición.
- El bucle se ejecuta al menos una vez en la estructura *REPEAT*, sin embargo en la estructura *WHILE* se puede ejecutar el bucle cero veces, si al principio no se cumple la condición.

En el caso de que el bucle se ejecute infinitas veces, se dice que es *cerrado*. Esta situación puede darse en estas dos estructuras, pero no en la siguiente.

• La estructura repetitiva FOR (para hacer desde-hasta)

En este tipo de estructura de control repetitiva el bucle se ejecuta un número determinado de veces, indicado por el valor inicial y final de una variable. El organigrama de la estructura FOR se representa en la figura 6.7.

En la figura 6.7 v representa la variable de control, v_i su valor inicial y v_f su valor final.

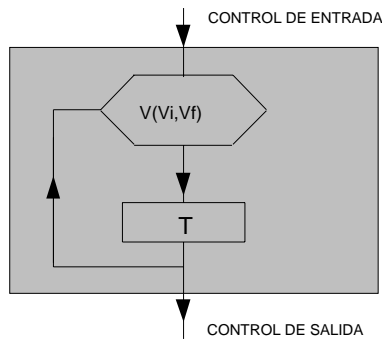
Esta estructura se caracteriza por que el número de veces que se realiza el tratamiento se conoce a priori.

Notación algorítmica

```

DESDE i:= valor inicial HASTA valor final HACER
    Tratamiento;
FIN_DESDE;

```

Figura 6.7 Organigrama de la estructura *FOR***6.2 LA ESTRUCTURA REPETITIVA *WHILE***

Se realiza en Pascal con la sentencia *WHILE*, cuyo diagrama sintáctico se muestra en la figura 6.8.

Figura 6.8 Diagrama sintáctico de *WHILE*

La condición es una expresión booleana, es decir una constante, variable, función o expresión de tipo booleano. Tal que si su resultado es *true* se ejecuta la sentencia o sentencias que forman el tratamiento, y si es *false* se finaliza la ejecución de la sentencia *WHILE*.

El *Tratamiento* es una sentencia simple o compuesta. El diagrama sintáctico de una sentencia compuesta se muestra en la figura 6.9. Las sentencias simples van separadas por *;*, y no necesitan *BEGIN-END*.

LA ESTRUCTURA REPETITIVA *WHILE*

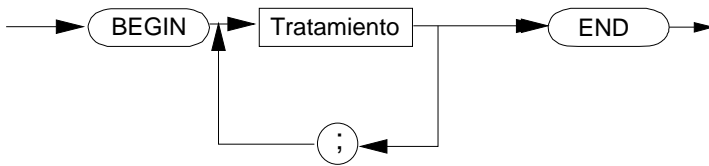


Figura 6.9 Diagrama sintáctico de una sentencia compuesta

La notación EBNF de la sentencia *WHILE* es:

```
<sentencia WHILE> ::= WHILE <expresión> DO <sentencia>
```

La sentencia puede ser simple o compuesta (tratamiento). La notación EBNF de una sentencia compuesta es:

```
<sentencia compuesta> ::= BEGIN <sentencia> { ; <sentencia> } END
```

Recordemos que una sentencia puede estar vacía.

Ejemplo 6.1

Escribir un programa que lea números enteros desde la entrada estándar y calcule su suma. La entrada de datos finalizará al introducir el número 0.

Análisis: El problema se resuelve acumulando en una variable (inicializada a 0) la suma de cada número introducido por teclado. Utilizaremos una estructura repetitiva *Mientras*, cuya condición de salida será: "número leído = 0".

Algoritmo:

```
INICIO
  Inicializar suma:=0;
  Leer n;
  MIENTRAS n≠0 HACER
    suma := suma + n;
    Leer n;
  FIN_MIENTRAS;
  Escribir suma;
FIN
```

Nota: Obsérvese que es necesario leer el primer dato fuera de la estructura *Mientras*, para poder examinar si se cumple la condición antes de ejecutar el bucle por primera vez.

Codificación en Pascal

```
PROGRAM Suma (input,output);
VAR
```

ESTRUCTURAS DE CONTROL

```
    n,sum :integer;
BEGIN
Write ( 'Introduzca un número entero ');
Readln(n);
sum:=0 ;
WHILE n<>0 DO
    BEGIN
        sum:=sum+n;
        Writeln('Introduzca un número entero --ponga cero para finalizar--');
        Readln(n);
    END;
Writeln ('el resultado es ',sum);
Writeln ('Pulse <Return> para volver al Editor');
Readln;
END.
```

Ejemplo 6.2

Escribir un programa que calcule el factorial de un número n utilizando un bucle *WHILE*.

Análisis: El factorial de un número entero positivo viene dado por la fórmula:

- Para $n>0$: $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$
- Por convenio: $0! = 1$

Utilizaremos una estructura *mientras que* para ir acumulando en una variable `fact` (inicializada a 1) el producto de los términos $n, n-1, n-2, \dots$ En cada iteración decrementamos n en una unidad. La condición de salida de la estructura iterativa es $n=1$.

Algoritmo

```
INICIO
  Leer n;
  Inicializar fact=1;
  MIENTRAS n>1 HACER
    fact := fact * n;
    n := n-1;
  FIN_MIENTRAS;
  Escribir fact;
FIN
```

Comentarios sobre el algoritmo

Obsérvese que para acumular una suma en una variable la inicializamos a cero, mientras que para acumular un producto inicializamos a la unidad. En general, para acumular una operación debemos inicializar al elemento neutro de dicha operación.

El algoritmo funciona también para $n=0$. Sería aconsejable realizar la entrada de datos de manera que se evite la posibilidad de que n sea negativo.

LA ESTRUCTURA ALTERNATIVA *IF-THEN*

Si se declara *n* como de tipo entero, en la mayoría de los ordenadores personales no se podrá calcular el factorial de números mayores de 7, ya que el resultado supera el valor de *MAXINT*. Hay diversas maneras de solventar esta dificultad. Por ejemplo, podemos declarar *fact* como *real*, y formatear la salida para que en pantalla aparezca sin decimales, como si fuese un entero.

Codificación en Pascal

```
PROGRAM Factorial (input,output);
(* Como se declara fact como entero no se puede calcular
   el valor de un factorial superior a 32767, n <=7 *)
VAR
  n,fact :integer ;
BEGIN
Write('introduzca un numero entero positivo ');
Readln(n);
(* Se inicializa a 1 el valor de la variable fact *)
fact:=1;
WHILE n>=1 DO
  BEGIN
    fact:=fact*n;
    n:=n-1  (* Se decrementa n en una unidad *)
  END;
Writeln('el resultado es ', fact );
Writeln ('Pulse <Return> para volver al Editor');
Readln
END.
```

6.3 LA ESTRUCTURA ALTERNATIVA *IF-THEN*

La estructura alternativa *IF-THEN* es la representada en la figura 6.1. El tratamiento T1 se ejecutará sólo si la condición es cierta.

Su diagrama sintáctico se muestra en la figura 6.10.



Figura 6.10 Diagrama sintáctico de *IF THEN*

El tratamiento es una sentencia simple o un bloque de sentencias.

Su notación EBNF es:

```
<sentencia IF> ::= IF <expresión> THEN <sentencia>
```

Ejemplo 6.3

Escribir un programa, que dados dos enteros, indique si alguno de los dos es negativo, y la suma de ambos.

Algoritmo

```

INICIO
  Leer a,b;
  SI (a<0) O (b<0)
    ENTONCES Escribir Mensaje;
  FIN_SI;
  Escribir (a+b)
FIN

```

Codificación en Pascal

```

PROGRAM Condiciones (input,output);
VAR
  a,b : integer;
BEGIN
  Write('Introduzca dos números enteros ');
  Readln (a,b);
  IF (a<0) OR (b<0)
    THEN
      Writeln ('uno de los dos números es negativo');
      Writeln ('la suma de los dos números vale ',a+b);
      Writeln ('Pulse <Return> para volver al Editor');
  Readln
END.

```

Ejemplo 6.4

Escribir un programa que dados dos números enteros, en el caso de que ambos sean negativos lo indique y determine el valor absoluto de su suma. El programa también debe de calcular la suma de los dos enteros, en cualquier caso.

Algoritmo

```

INICIO
  Leer a,b;
  SI (a<0) y (b<0)
    ENTONCES
      Escribir Mensaje;
      c := |a+b|
      Escribir c;
  FIN_SI;
  Escribir (a+b)
FIN

```

Codificación en Pascal

```

PROGRAM CondicionesConBloque (input,output);
VAR
  a,b,c : integer;
BEGIN
  Write ('Introduzca dos números enteros ');
  Readln (a,b);

```


LA ESTRUCTURA ALTERNATIVA *IF-THEN-ELSE*

```
IF (a<0) AND (b<0) THEN
  BEGIN
    Writeln ('los dos números son negativos');
    c := a + b;
    c := Abs(c);
    Writeln ('su suma en valor absoluto vale ',c)
  END;
Writeln ('la suma de los dos números vale ',a+b);
Writeln ('Pulse <Return> para volver al Editor');
Readln
END.
```

6.4 LA ESTRUCTURA ALTERNATIVA *IF-THEN-ELSE*

La estructura alternativa *IF-THEN-ELSE* es la representada en la figura 6.3. Según que la condición sea cierta o falsa, se ejecutará el tratamiento T1 o el T2, respectivamente.

Su diagrama sintáctico se muestra en la figura 6.11. Los tratamientos T1 o T2 son sentencias simples o compuestas.

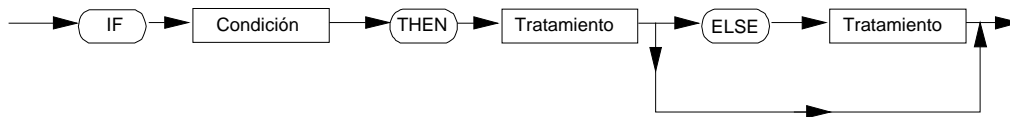


Figura 6.11 Diagrama sintáctico de *IF THEN ELSE*

Su notación EBNF es:

```
<sentencia IF THEN ELSE> ::= IF <expresión> THEN <sentencia>
                             (<vacío> | ELSE <sentencia>)
```

Ejemplo 6.5

Escribir un programa que resuelva ecuaciones de segundo grado.

Análisis: La solución de una ecuación de segundo grado de la forma

$$a \cdot x^2 + b \cdot x + c = 0$$

viene dada por:

$$x = \frac{-b \pm \sqrt{d}}{2a}$$

siendo $d = b^2 - 4 \cdot a \cdot c$ (discriminante)

- Si $d < 0$: La ecuación tiene dos soluciones imaginarias, que no calculamos pues no es el objetivo de este ejemplo.

ESTRUCTURAS DE CONTROL

- Si $d > 0$: Tenemos dos soluciones reales:

$$x_1 = \frac{-b + \sqrt{d}}{2a}$$

$$x_2 = \frac{-b - \sqrt{d}}{2a}$$

- Si $d = 0$: Tenemos una única solución, que se puede obtener aplicando cualquiera de las dos fórmulas anteriores, ya que en este caso $x_1 = x_2$

Algoritmo

```
INICIO
  Leer a,b,c;
  Calcular d:=b2-4ac;
  SI (d<0)
    ENTONCES Escribir Mensaje
    SI_NO
      x1 :=(-b+Sqrt(d))/(2a);
      x2 :=(-b-Sqrt(d))/(2a);
      Escribir x1, x2;
  FIN_SI;
FIN
```

Codificación en Pascal

```
PROGRAM Ecua2grado (input,output);
VAR
  a,b,c : integer;
  d,x1,x2 : real;
BEGIN
  Writeln ('RESOLUCION DE LA ECUACION DE 2º GRADO');
  Writeln ('*****');
  Writeln ('          ax2+bx+c=0          ');
  Writeln ('          *****          ');
  Write ('Introduzca a,b,c ');
  Readln (a,b,c);
  d:=Sqr(b)-4*a*c ;
  IF d<0 THEN
    Writeln ('Soluciones imaginarias')
  ELSE
    BEGIN
      x1:=(-b+Sqrt(d))/(2*a);
      x2:=(-b-Sqrt(d))/(2*a);
      Writeln ('Las soluciones son x1=',x1,' y x2=',x2)
    END;
  Writeln ('Pulse <Return> para volver al Editor');
  Readln
END.
```

LA ESTRUCTURA ALTERNATIVA *IF-THEN-ELSE*

La sentencia **IF** jerarquizada o anidada

Una sentencia *IF* es jerarquizada o anidada cuando alguno de los tratamientos a realizar contiene a su vez otra sentencia *IF*.

Ejemplo de estructura *IF* anidada a tres niveles:

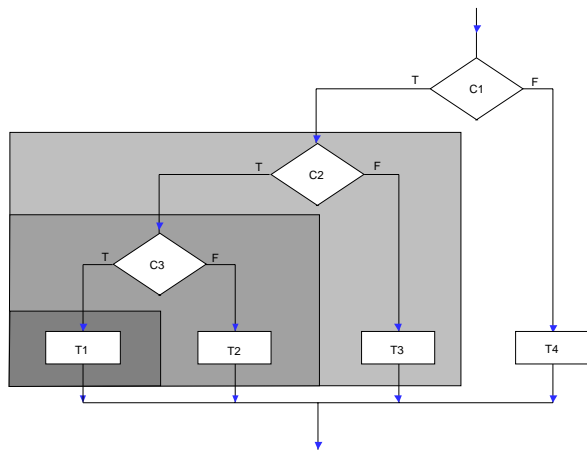


Figura 6.12 Estructura *IF* anidada a tres niveles

Esta estructura corresponde a la sentencia:

```
IF c1
  THEN IF c2
    THEN IF c3
      THEN T1
      ELSE T2
    ELSE T3
  ELSE T4
```

Ambigüedad de la cláusula *ELSE*

Puede darse el caso de que el número de palabras *THEN* no sea el mismo que el de palabras *ELSE*.

Por ejemplo:

```
IF c1 THEN IF c2 THEN T1 ELSE T3
```

En estos casos puede haber una indeterminación, ya que no sabemos si el *ELSE* del ejemplo corresponde al primer *THEN* o al segundo. Pueden ser dos estructuras distintas:

ESTRUCTURAS DE CONTROL

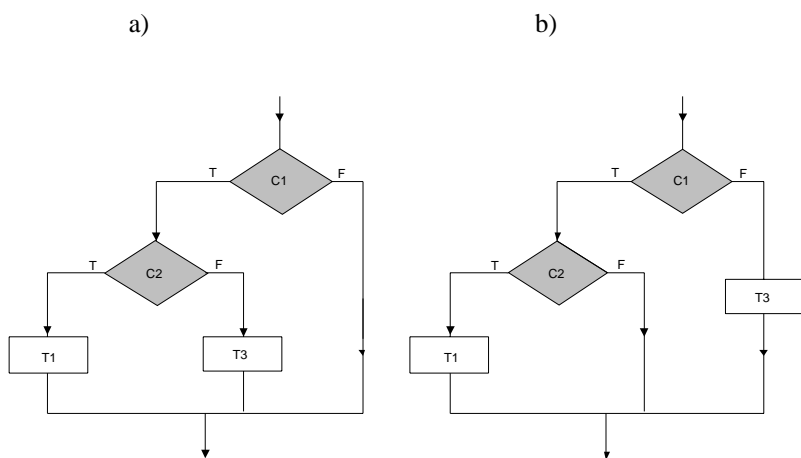


Figura 6.13 Ambigüedad de la cláusula *ELSE*

Por convenio, la correspondencia **THEN-ELSE** es la siguiente:

- Se avanza hasta encontrar el primer *ELSE*, y éste se corresponde con el *THEN* que tiene antes más cerca de él.
- El siguiente *ELSE* que se encuentra corresponde con el *THEN* anterior situado más cerca de él, al que todavía no le ha correspondido ningún *ELSE* y así sucesivamente.

Volviendo al ejemplo anterior, las líneas de asteriscos indican la correspondencia **THEN-ELSE** según el convenio anterior:

```

IF      c1
  THEN  IF      c2
        ****THEN
        *      T1
        *
        ****ELSE
        T3
    
```

Luego la estructura correcta es la mostrada en la figura a).

Si se quisiera utilizar la estructura de la figura b), sería necesario utilizar bloques *BEGIN-END* para alterar el anterior convenio:

```

IF      c1
  THEN  BEGIN
        IF      c2
          THEN  T1
        END
      ELSE  T3
    
```

LA ESTRUCTURA ALTERNATIVA IF-THEN-ELSE

Al utilizar *BEGIN-END* se indica explícitamente que al primer *THEN* le corresponde el único *ELSE* existente.

Ejemplo 6.6

Indicar la estructura que se corresponde con la siguiente IF anidada:

```
IF c1 THEN T1 ELSE IF c2 THEN IF c3 THEN T2 ELSE T3
```

Al no existir ningún *BEGIN-END* la correspondencia se realiza por convenio:

```
IF c1
  THEN T1
  ELSE IF c2
        THEN IF c3
              THEN T2
              ELSE T3
```

La representación gráfica de la estructura está en la figura 6.14.

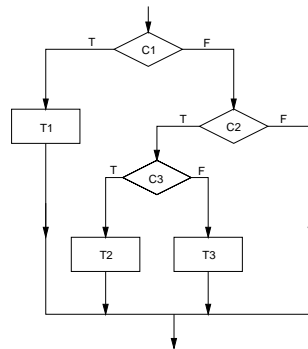


Figura 6.14 Ejemplo de correspondencia *IF-THEN-ELSE*

Ejemplo 6.7

Escribir un programa que lea dos números enteros a y b, y nos diga cuál es mayor o si son iguales.

Algoritmo

```
INICIO
  Leer a, b;
  SI a>b ENTONCES
    Escribir (a>b)
  SI_NO
```

```

SI a<b ENTONCES
    Escribir (a<b)
SI_NO
    Escribir (a=b)
FIN_SI;
FIN_SI;

```

Codificación en Pascal

```

PROGRAM Comparar (input, output);
VAR
  a,b :integer;
BEGIN
  Write ('meter dos números enteros ');
  Readln (a,b);
  IF a>b
  THEN
    Writeln(a, ' es mayor que ',b)
  ELSE
    IF a<b
    THEN
      Writeln( a, ' es menor que ',b)
    ELSE
      Writeln ( a , ' es igual que ' , b);
  Writeln ('Pulse <Return> para volver al editor');
  Readln;
END.

```

6.5 LA ESTRUCTURA REPETITIVA FOR

La sentencia *FOR*, también denominada estructura repetitiva con contador, es la representada en la figura 6.7.

Permite realizar un tratamiento un número de veces conocido a priori.

Su diagrama sintáctico es el representado en la figura 6.15.

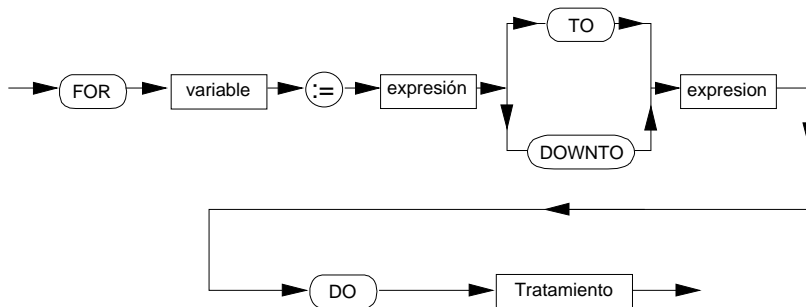


Figura 6.15 Diagrama sintáctico de *FOR*

La notación EBNF de la sentencia *FOR* se muestra a continuación:

LA ESTRUCTURA REPETITIVA FOR

```
<sentencia for>      ::= FOR <variable de control> := <lista for> DO
                        <sentencia>
<lista for>          ::= <valor inicial> TO <valor final> |
                        <valor inicial> DOWNTO <valor final>
<variable de control> ::= <variable>
<valor inicial>      ::= <expresión>
<valor final>        ::= <expresión>
```

Características de la sentencia FOR

- El tratamiento consiste en una sentencia simple o compuesta que se forma de manera idéntica que en el caso de la sentencia *WHILE*.
- La variable de control, valor inicial y valor final han de ser del mismo tipo. Los tipos posibles son:
 - integer
 - char
 - enumerado u ordinal
 - tipo subrango de los tres anteriores
- Los valores inicial y final pueden ser constantes, variables o expresiones de los tipos anteriormente citados.
- En el tratamiento o bucle de la sentencia *FOR* no se pueden modificar los valores de la variable de control, valor inicial y valor final.
- Cuando finaliza la ejecución de la sentencia *FOR* la variable de control tiene un valor indeterminado.
- El funcionamiento de la sentencia *FOR* es el siguiente:

- Con cláusula TO

Si utilizamos la cláusula *TO* la variable de control toma inicialmente el valor inicial y cada vez que se ejecuta el bucle se incrementa en uno ese valor de acuerdo con la función de Pascal *Succ*, función ya vista anteriormente, que en el caso de argumento entero da como resultado la suma de una unidad a dicho argumento:

```
Succ (6) = 7
Succ (-4) = -3
```

El bucle dejará de ejecutarse cuando el valor de la variable de control supere al valor final.

Sea por ejemplo:

```
FOR i := 2 TO 10 DO T
```

ESTRUCTURAS DE CONTROL

En este ejemplo trabajamos con el tipo entero, i será una variable entera, 2 y 10 son constantes enteras y el tratamiento τ se ejecutará 9 veces; para cada una de ellas i tomará respectivamente los valores 2,3,4,5,6,7,8,9,10.

En el caso de que el valor inicial sea mayor que el valor final el bucle no se ejecutará ninguna vez. Si son iguales, se ejecuta una vez.

- Con cláusula *DOWNTO*

Si utilizamos la cláusula *DOWNTO* la variable de control toma inicialmente el valor inicial y cada vez que se ejecuta el bucle su valor se decrementa de acuerdo con la función de Pascal *Pred*, función que con argumento entero da como resultado el restar una unidad al argumento.

Sea por ejemplo:

```
FOR i := 2 DOWNTO -2 DO T
```

En este caso i será una variable entera y el tratamiento τ se ejecutará 5 veces, para cada una de ellas i tomará respectivamente los valores -2, 1, 0, -1, -2.

En el caso de que el valor inicial sea menor que el valor final el bucle no se ejecutará ninguna vez.

Ejemplo 6.8

Realizar un programa que calcule la media de n números, usando la sentencia *FOR*.

Análisis: Leeremos el número de datos, n , y acumularemos la suma de cada dato introducido mediante una estructura *FOR*, de manera similar a la utilizada en el ejemplo 6.1 (estructura *WHILE*). Por último, dividimos por n para calcular la media. Todas las variables pueden ser de tipo entero excepto la que represente la media, que debe ser real porque almacena el resultado de una división.

Algoritmo

```
INICIO
  Leer n;
  Inicializar s:=0;
  DESDE i:=1 HASTA n HACER
    Leer x;
    s := s+x;
  FIN_DESDE;
  xm := s/n;
  Escribir xm;
FIN
```


LA ESTRUCTURA REPETITIVA FOR

Codificación en Pascal

```
PROGRAM media (input,output);
VAR
    n,x,i,s :integer;
    xm :real;
BEGIN
    Write ('Introduzca el número de casos ');
    Readln ( n );
    s := 0;
    FOR i:=1 TO n DO
        BEGIN
            Write ('Introduzca el valor del caso ',i,' =');
            Readln (x) ;
            s := s + x;
        END;
    xm := s/n;
    Writeln ('La media de ',n,' casos vale ',xm);
    Writeln ('Pulse <Return> para volver al Editor');
    Readln;
END.
```

Ejemplo 6.9

Escribir un programa que calcule el factorial de un número, con un bucle *FOR* con cláusula *DOWNTO*.

Análisis: Se resuelve análogamente al ejemplo 6.2 (estructura *WHILE*), pero en este caso no es necesario decrementar *n*, pues lo hace automáticamente la estructura *FOR* si usamos cláusula *DOWNTO*.

Algoritmo

```
INICIO
    Leer n;
    Inicializar fact:=1;
    DESDE i:=n DESCENDIENDO HASTA 1 HACER
        fact := fact*i
    FIN_DESDE;
    Escribir fact;
FIN
```

Codificación en Pascal

```
PROGRAM Factorial3 (input,output);
(* Este programa calcula el factorial de un número entero *)
(* Si el valor leído es negativo genera un mensaje *)
(* Sólo puede calcular HASTA el factorial de 7 con MAXINT=32767 *)
VAR
    n,i,fact : integer ;
BEGIN
    Write ('Introduzca un número entero ');
    Readln ( n );
    (* Se inicializa el valor de fact *)
```

ESTRUCTURAS DE CONTROL

```
fact := 1;
FOR i:=n DOWNTO 1 DO
    fact := fact * i ;
Writeln ('El factorial de ', n , ' vale ',fact);
Writeln ('Pulse <Return> para volver al Editor');
Readln
END.
```

Ejemplo 6.10

Modificar el programa anterior para realizar el bucle *FOR* con la cláusula *TO*.

Análisis: Puesto que el producto es conmutativo, podemos utilizar también la cláusula *TO* en el cálculo del factorial. Incluiremos en el algoritmo la posibilidad de trabajar con *n* negativo o mayor que siete.

Algoritmo

```
INICIO
  Leer n;
  SI n >= 0
    ENTONCES
      Inicializar fact:=1;
      DESDE i:=1 HASTA n HACER
        fact := fact*i
      FIN_DESDE;
      Escribir fact
    SI_NO
      Escribir Mensaje de error
  FIN_SI
FIN
```

Codificación en Pascal

```
PROGRAM Facto2 (input,output);
(* Este programa calcula el factorial de un número entero *)
(* Si el valor leído es negativo genera un mensaje *)
(* Sólo puede calcular hasta el factorial de 7 con MAXINT=32767 *)
VAR
  i,n,fact : integer;
BEGIN
  Write ('Introduzca un número entero ');
  Readln ( n );
  IF n>=0
    THEN
      BEGIN
        fact := 1;
        FOR i:= 2 TO n DO
          fact := fact * i;
        Writeln ('El factorial de ',n,' vale ',fact )
      END
    ELSE
      Writeln ('No existe factorial de números negativos');
  Writeln ('Pulse <Return> para volver al Editor');
  Readln
END.
```

LA ESTRUCTURA REPETITIVA FOR

Sentencias anidadas

Anteriormente se estudió la sentencia *IF* anidada, pero cualquier sentencia puede estar anidada en otras. Es decir en los tratamientos puede haber cualquier tipo de sentencia.

Los puntos suspensivos indican sentencias o grupos de sentencias.

Ejemplo 6.11

Diseñar un programa que escriba las tablas de multiplicar del 0 al 9.

Análisis: Usaremos dos variables de tipo entero: *i*, que representa cada número del que se escribe la tabla (del 0 al 9) y *j*, para ir calculando $i*j$, con *i* constante y *j* variando de 0 a 9. Separamos dos tablas consecutivas mediante otro bucle *FOR* con otro contador, *k*, que representa el número de líneas de separación (podríamos utilizar *j*, pero oscurecería el algoritmo).

Algoritmo

```
INICIO
  DESDE i:=0 HASTA 9 HACER
    DESDE j:=0 HASTA 9 HACER
      Escribir i*j;
    FIN_DESDE j;
    Retener salida por pantalla; (con la tabla de i)
    DESDE k:=1 HASTA p (nº de líneas de separación) HACER
      Saltar línea;
    FIN_DESDE k;
  FIN_DESDE i;
FIN
```

Nota: Obsérvese que no necesitamos entrada de datos, por lo que no hay que especificar input entre los parámetros del programa.

Codificación en Pascal

```
PROGRAM TablaDeMultiplicar (output);
(* Escribe la tabla de multiplicar, separando las tablas de los
distintos números, y parando la salida por pantalla para ver
cada tabla *)
CONST
  n=0;
  m=9;
  p=4;
VAR
  i,j,k :integer;
BEGIN
  FOR i:=n TO m DO
    BEGIN
      FOR j:=n TO m DO
        Writeln (i,' * ',j,' = ',i*j);
      Write ('Pulse <Return> para continuar');
      Readln;      (* Para detener la salida por pantalla *)
    FOR k:=1 TO p DO
```

```

        Writeln
    END;
    Write ('Pulse <Return> para volver al Editor');
    Readln
END.

```

6.6 LA ESTRUCTURA REPETITIVA REPEAT-UNTIL

La estructura repetitiva *REPEAT* es una estructura de control iterativa, parecida a la sentencia *WHILE*, que nos ofrece una nueva posibilidad de construir bucles.

En *REPEAT* la condición de fin de bucle se comprueba al final del mismo en vez de hacerlo al principio como ocurría en *WHILE*. Esta es la diferencia fundamental entre ambos tipos de bucle.

Su diagrama sintáctico está en la figura 6.16.

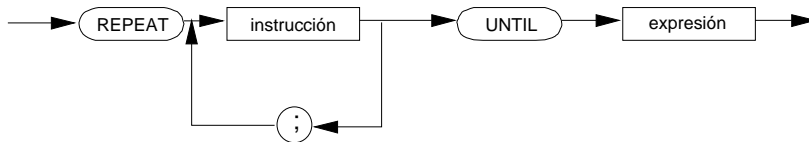


Figura 6.16 Diagrama sintáctico de *REPEAT-UNTIL*

Su notación EBNF es:

```

<sentencia REPEAT> ::= REPEAT <instrucción> { ; <instrucción> } UNTIL <expresión>

```

Su organigrama es el representado en la figura 6.6.

Como se aprecia en la figura, la sentencia o sentencias que componen el cuerpo del bucle se repiten *hasta (UNTIL)* que la condición se haga *TRUE*.

Comparación de las estructuras WHILE y REPEAT

Observando los organigramas de ambas estructuras (figura 6.17), se aprecian claramente las similitudes y diferencias que existen entre ellas. Ambas estructuras sirven para repetir un tratamiento según el valor de una condición. En resumen se diferencian en dos puntos:

- En *WHILE* la condición es de entrada al bucle, y en *REPEAT* es de salida. Para realizar la misma tarea, usaríamos condiciones contrarias según la estructura elegida.
- Como consecuencia, el bucle *WHILE* puede ejecutarse cero veces, mientras que *REPEAT* se ejecuta al menos una vez.

LA ESTRUCTURA REPETITIVA *REPEAT-UNTIL*

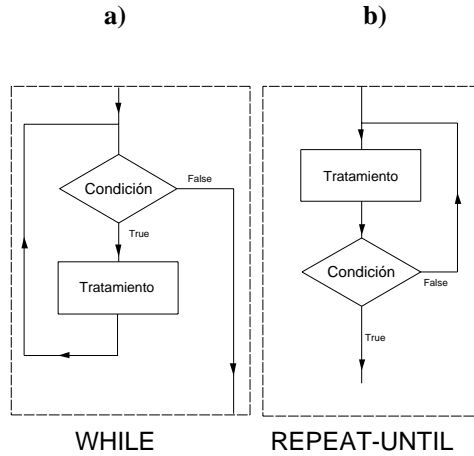


Figura 6.17 Comparación de las estructuras *WHILE* y *REPEAT*

Ejemplo 6.12

Como ejemplo comparativo veamos un bucle contador utilizando *WHILE* y su equivalente con *REPEAT*.

<pre> cont:=1; WHILE cont < 5 DO BEGIN sentencias; cont := cont+1; END; </pre>	<pre> cont:=1; REPEAT sentencias; cont := cont+1; UNTIL cont = 5; </pre>
---	--

En función del tipo de problema a resolver, nos convendrá utilizar un bucle *WHILE* o un *REPEAT*, según cual de ellos resulte más simple o más apropiado.

Ejemplo 6.13

Supongamos que nuestro problema es obtener el primer carácter distinto de blanco de una línea del fichero *input*. Ese carácter puede que sea el primero de la línea o bien tener uno o más blancos que le precedan. Nuestro bucle debe saltar esos blancos, si existen, y entregar en la variable *ch* de tipo *char* el carácter no blanco.

Observe que el bucle *REPEAT* para resolver el problema es más corto que el *WHILE* equivalente:

<pre> REPEAT Read (ch); UNTIL ch <> ' '; </pre>	<pre> Read (ch); WHILE ch = ' ' DO Read (ch); </pre>
---	--

Ejemplo 6.14

Escribir un programa que calcule la suma $1 + 1/2 + 1/3 + \dots + 1/n$, para un valor de n leído por teclado.

Análisis: El problema es similar a los ya resueltos con *WHILE* y *FOR*, para calcular la suma y la media de un conjunto de números. En este caso lo que hay que acumular es la suma de $1/n$. Los ejemplos anteriores podrían resolverse utilizando una estructura *REPEAT*.

Algoritmo

```

INICIO
  Leer n;
  Inicializar suma:=0;
  Repetir
    suma := suma + 1/n;
    n := n-1;
  HASTA n=0;
  Escribir suma;
FIN

```

Codificación en Pascal

```

PROGRAM Sumatorio (input,output);
VAR
  n: integer;
  suma: real;
BEGIN
  Write('Introduzca n ');
  Readln(n);
  suma := 0;
  REPEAT
    suma := suma + 1/n;
    n := n-1;
  UNTIL n=0;
  Writeln('La suma vale', suma:10:3);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.

```

6.7 TRATAMIENTO SECUENCIAL DE LA INFORMACION

El **tratamiento de la información mediante secuencias** es un método sistemático para construir algoritmos iterativos. Se basa en la técnica de Diseño Descendente para construir algoritmos, estudiada en la sección 2.2, *Las Fases del Proceso de Programación*, del capítulo 2.

Dicho método sistemático se basa en:

- Reconocer o inducir una *estructura de secuencia*
- Aplicar un *esquema de recorrido* (o enumeración)

Noción de Secuencia

Una secuencia es una cuádrupla formada por:

- Un *conjunto* de elementos (generalmente una estructura de datos)
- Un *elemento inicial*, reconocible, que se distingue del resto
- Un *elemento final*, también reconocible entre los demás
- Una *relación de sucesión* entre dichos objetos, de tal manera que cada uno tiene un único predecesor (excepto el primero) y un único sucesor (excepto el último)

La característica fundamental de una secuencia es el **acceso secuencial** a sus elementos: para acceder al elemento *i-ésimo* hay que recorrer los *i-1* elementos anteriores. Se accede al primer objeto, y se pasa de cada elemento al siguiente, hasta llegar al último.

Esquemas de recorrido o enumeración de secuencias

El método propone dos esquemas, basados en los dos casos posibles que se pueden presentar en una secuencia:

- Esquema nº 1: Usado cuando el tratamiento de un elemento genérico (elemento en curso) es distinto del tratamiento del elemento final.
- Esquema nº 2: Usado cuando el tratamiento del elemento en curso es igual que el tratamiento del elemento final.
- **Esquema nº 1:** En este caso hay que detener la iteración **antes** de tratar el elemento final. Es necesario observar la condición que controla el bucle antes del tratamiento. Por tanto, se basa en la utilización de una **Estructura Mientras** (*WHILE* en Pascal).
- **Esquema nº 2:** En este caso hay que detener la iteración **después** de tratar el elemento final. Es necesario observar la condición que controla el bucle **después** del tratamiento. Por tanto, se basa en la utilización de una **Estructura Repetir** (*REPEAT* en Pascal).

Antes de contruir los algoritmos de ambos esquemas, especificaremos las *acciones elementales* que aparecen en ellos:

- **Inicializar tratamiento:** Conjunto de operaciones necesarias antes de tratar los elementos.
- **Obtener primer elemento:** Esta acción se puede descomponer en otras dos más elementales:
 - Inicializar adquisición: Conjunto de operaciones necesarias antes de empezar a obtener elementos.
 - Obtener elemento siguiente

ESTRUCTURAS DE CONTROL

- Obtener elemento siguiente
- Tratar elemento
- Tratar elemento final

Los algoritmos de estos dos esquemas son:

```
ACCION esquema de recorrido n° 1 ES
  {T. final ≠ T. en curso}
  inicializar tratamiento;
  obtener primer elemento;
  MIENTRAS NO(elemento final) HACER
    tratar elemento;
    obtener elemento siguiente;
  FIN_MIENTRAS;
  tratar elemento final;
FIN_ACCION;
```

```
ACCION esquema de recorrido n° 2 ES
  {T. final = T. en curso}
  inicializar tratamiento;
  inicializar adquisición;
  REPETIR
    obtener elemento siguiente;
    tratar elemento;
  HASTA elemento final;
FIN_ACCION;
```

Esquema de Búsqueda Asociativa

Es un caso particular de recorrido de secuencias. Se trata de buscar en una secuencia el elemento que cumple una condición determinada. No es imprescindible recorrer la secuencia completa. La búsqueda finaliza cuando se cumple una de las dos condiciones siguientes:

- Se ha encontrado el elemento buscado (*éxito en la búsqueda*).
- Se ha encontrado antes el elemento final (*fracaso en la búsqueda*).

En la construcción del algoritmo correspondiente usaremos dos nuevas acciones elementales:

- Tratar elemento buscado: Conjunto de operaciones a ejecutar una vez hallado el elemento.
- Tratar ausencia: Conjunto de operaciones a ejecutar si fracasa la búsqueda.

Se utiliza además la condición *elemento hallado*, que es *cierta* si el elemento en curso tiene la propiedad buscada y *falsa* en caso contrario.

TRATAMIENTO SECUENCIAL DE LA INFORMACION

Algoritmo

```
ACCION Búsqueda Asociativa ES
  inicializar tratamiento;
  obtener primer elemento;
  MIENTRAS NO(elemento hallado) Y NO(elemento final) HACER
    obtener elemento siguiente;
  FIN_MIENTRAS;
  SI (elemento hallado)
    ENTONCES Tratar elemento buscado
    SI_NO Tratar ausencia;
  FIN_SI;
FIN_ACCION;
```

Etapas para la aplicación sistemática del método

Para aplicar sistemáticamente este método de construcción de algoritmos iterativos, hay que seguir las siguientes etapas:

- Detectar o inducir una **estructura de secuencia** en el conjunto de datos a tratar.
- Concretar las acciones Tratar elemento en curso y Tratar elemento final.
- *Seleccionar* el esquema de recorrido adecuado.
- *Adaptarlo* a la secuencia a tratar

En la construcción de la secuencia se pueden presentar dos casos:

- ☒ Sus elementos son directamente asimilables por la máquina
- ☒ Sus elementos **no** son directamente asimilables: Será necesario descomponerlos en elementos más pequeños, hasta el nivel de la máquina, aplicando el método de **Análisis Descendente**, estudiado en la sección 2.2, *Las fases del proceso de programación*, del capítulo 2.

La máquina de caracteres

La máquina de caracteres se utiliza para el aprendizaje del tratamiento de secuencias. Es como una caja, por la que pasa una cinta con caracteres impresos, con un agujero por el que sólo se ve un carácter de la cinta (elemento en curso). La máquina funciona pulsando dos botones: uno de rebobinado de la cinta, y otro de avance al siguiente carácter. Cuando se coloca una cinta nueva es necesario rebobinarla, y una vez rebobinada hay que pulsar el botón de avance para acceder al primer carácter (para verlo por el agujero). El final de la cinta está marcado por el carácter punto ('.'). Esquemáticamente se puede representar la máquina así:

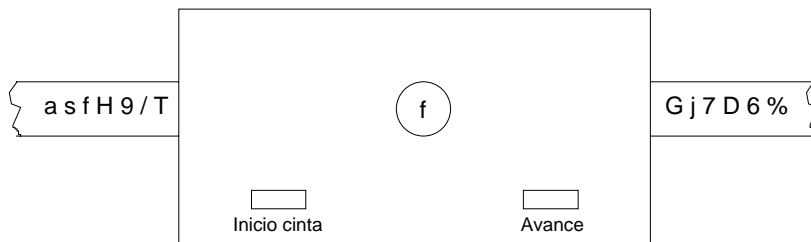
Máquina de caracteres:

Figura 6.18 La máquina de caracteres

Ejemplo 6.15

Escribir un algoritmo para contar las apariciones del caracter 'A' en la cinta de la máquina de caracteres.

Análisis: Aplicaremos el método de tratamiento secuencial de la información. El tratamiento de cada elemento consiste en observar si el caracter en curso es igual al buscado, e incrementar un contador. El último elemento ('.') no tiene el mismo tratamiento que los demás, luego aplicaremos el esquema nº 1. Necesitaremos una variable contador de tipo entero, y una variable de tipo caracter para ir leyendo la cinta.

Algoritmo

```

ACCION ContarA ES
  {inicializar tratamiento}
  numA := 0;
  {inicializar adquisición}
  inicio_cinta; {pulsar botón de rebobinado}
  c:= caracter siguiente; {obtener caracter siguiente}
  MIENTRAS c<> '.' HACER
    {tratar caracter en curso}
    SI c='A' ENTONCES
      numA := numA + 1;
    FIN_SI;
    c := caracter siguiente;
  FIN_MIENTRAS;
  Imprimir numA;
FIN_ACCION;

```

6.8 LA ESTRUCTURA MULTIALTERNATIVA CASE

La estructura de control multialternativa *CASE* permite ejecutar una entre varias acciones, en función del valor que tome una determinada variable o expresión denominada **selector** de la *CASE*. Su organigrama es el representado en la figura 6.4, y su diagrama sintáctico se muestra en la figura 6.19.

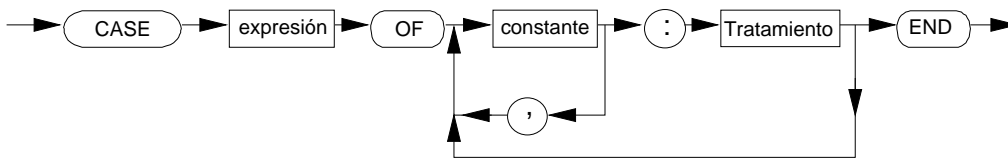


Figura 6.19 Diagrama sintáctico de CASE

Su notación EBNF es:

```

<sentencia case> ::= CASE <expresión> OF <elemento de la lista case>
                  { ; <elemento de la lista case> } END
<elemento de la lista case> ::= <lista de rótulos case> : <sentencia> | <vacía>
<lista de rótulos case> ::= <rótulo de case> { , <rótulo de case> }
    
```

Esta sentencia es una generalización de la sentencia *IF*, equivalente a una serie de sentencias *IF-THEN-ELSE* anidadas, como demuestra el ejemplo siguiente:

Ejemplo 6.16

```
VAR i: integer;
```

Sentencia CASE

```

CASE i OF
  1 : sent1;
  2 : sent2;
  5 : sent3;
END; (* de CASE *)
    
```

IF-THEN-ELSE equivalentes

```

IF i = 1
  THEN sent1
  ELSE IF i = 2
    THEN sent2
    ELSE IF i = 5
      THEN sent3;
    
```

En este ejemplo, *i*, de tipo *integer*, es el selector de *CASE*, el cual debe ser de un tipo ordinal, por lo que no están permitidas expresiones o variables de tipo real como selectores.

Los valores 1, 2 y 5 del ejemplo se denominan *etiquetas-CASE*, y deben ser constantes del mismo tipo que el *selector*. Estas constantes son las que seleccionan la sentencia a ejecutar en cada caso.

Ejemplo 6.17

Si queremos que para más de un valor del selector se ejecute la misma acción, formaremos una lista de *etiquetas-CASE* separadas por comas delante de la sentencia que queremos ejecutar, como se muestra a continuación:

```
CASE i OF
    1 : sent1;
    2,4,8 : sent2;
    5 : sent3;
END;
```

la sentencia `sent2` se ejecuta si `i` vale `2`, `4` u `8`. Observe que no se exige ningún orden en las *etiquetas-CASE*.

Las sentencias a ejecutar pueden ser sentencias simples, como en los ejemplos anteriores, o compuestas, es decir, un grupo de sentencias encerradas entre *BEGIN* y *END*.

Antes de exponer más ejemplos veamos en detalle la sintaxis de la sentencia *CASE*:

- El selector se coloca entre las palabras reservadas *CASE* y *OF*
- Las etiquetas van separadas de la acción por `'.'`
- Cuando hay varias etiquetas, se separan entre sí por `'.'`
- Detrás de cada acción debe ir un `'.'` excepto la última que puede llevarlo opcionalmente (precede a un *END*).
- Finalmente la palabra reservada *END* cierra la sentencia *CASE*.

Ejemplo 6.18

Se puede utilizar una expresión entera como selector:

```
CASE (i+5) MOD 3 OF
    0: ; (* no se hace nada *)
    1: i := i+2;
    2: i := i-2;
END;
```

obsérvese que para la primera etiqueta, `0`, no se ejecuta nada (sentencia vacía).

Ejemplo 6.19

Otro ejemplo de la utilización de *CASE* sería:

```
PROGRAM EscribeTrimestres (input, output);
VAR
    i:integer;
```

LA ESTRUCTURA MULTIALTERNATIVA CASE

```
BEGIN
  Writeln ('Introduzca código del mes');
  Readln (i);
  IF (i<1) OR (i>12)
  THEN
    Writeln ('Código fuera de rango')
  ELSE
    CASE i OF
      1,2,3 : Writeln ('primer trimestre');
      4,5,6 : Writeln ('segundo trimestre');
      7,8,9 : Writeln ('tercer trimestre');
      10,11,12: Writeln ('cuarto trimestre')
    END
  END.
END.
```

Ejemplo 6.20

Veamos un ejemplo con selector de tipo *char*:

```
CASE ch OF
  '?' : Write ('Interrogación');
  'a','e','i','o','u': BEGIN
                        Write('Vocal');
                        contVocales := contVocales+1;
                        END;
  '*' : Write ('Asterisco');
END;
```

Ejemplo 6.21

Es muy útil para la entrada/salida de tipos enumerados sustituyendo a sentencias *IF* anidadas, que se utilizaron con este fin en el capítulo anterior.

Para el tipo palos, definido en la página 146 (*Tipos enumerados*), se puede cambiar la *IF* anidada de la página 150, por el tratamiento de la forma:

```
CASE carta OF
  oros : Write ('oros');
  copas : Write ('copas');
  espadas: Write ('espadas');
  bastos : Write ('bastos');
END;
```

Ejemplo 6.22

Quizás la aplicación más frecuente de esta sentencia sea la implementación de menús de opciones. Una sentencia *CASE* suele usarse en el bloque principal de un programa gobernado por un menú. Como selector suele utilizarse la variable en que se lee la opción elegida. Ejemplo:

```
PROGRAM FicheroDatos(input, output);
CONST ...
TYPE ...
VAR ...
  opcion: char;
  fin: boolean;
BEGIN
  fin := false;
```

ESTRUCTURAS DE CONTROL

```
WHILE NOT fin DO
  BEGIN
    Writeln('MENU:');
    Writeln;
    Writeln('1...Crear fichero de datos.');
```

```
Writeln('2...Ordenar.');
```

```
Writeln('3...Listar.');
```

```
Writeln('4...Fin.');
```

```
Write('.....Escoja una opción.....');
```

```
Readln(opcion);
```

```
CASE opcion OF
```

```
  1: BEGIN
```

```
    ... (* Bloque para crear el fichero de datos *)
```

```
  END;
```

```
  2: BEGIN
```

```
    ... (* Ordenación *)
```

```
  END;
```

```
  3: BEGIN
```

```
    ... (* Listado *)
```

```
  END;
```

```
  4: fin := true;
```

```
END; (* CASE *)
```

```
END; (* WHILE *)
```

```
END.
```

En este ejemplo, hemos utilizado una estructura *CASE* para seleccionar entre varias opciones la tarea a ejecutar, anidada dentro de una estructura *WHILE*, para repetir la aparición del menú y la selección y ejecución de la opción correspondiente, hasta que se elige la opción *Fin*.

En resumen, las estructuras *CASE* se utilizan sustituyendo a estructuras *IF* anidadas por ser más sencillas, pero sólo son útiles si el selector adopta valores que se pueden especificar como constantes ordinales. Las etiquetas no pueden ser variables ni expresiones.

Falta un aspecto a considerar ¿Qué ocurre si el selector toma un valor que no corresponde con ninguna de las etiquetas *CASE*? En el lenguaje Pascal estándar el resultado es indefinido y puede producirse un error en tiempo de ejecución, pero muchas implementaciones del Pascal admiten esta posibilidad, simplemente, no ejecutando ninguna acción en este supuesto. Otras completan la sentencia *CASE* con una cláusula *OTHERWISE* ó *ELSE*, tras la cual se indica la acción a ejecutar en esos casos. Al final del capítulo se explican las peculiaridades de esta sentencia en Turbo Pascal.

Ejemplo 6.23

Escribir un programa que escriba el mes, a partir de su codificación como número.

Análisis: Leemos el código numérico del mes, y mediante una estructura *CASE*, escribimos el nombre completo correspondiente al código leído.

Algoritmo

```
INICIO
  Leer i; (código del mes)
  SEGUN i HACER
```

LA ESTRUCTURA MULTIALTERNATIVA CASE

```
        1: Escribir (Enero);
        2: Escribir (Febrero);
        ...
        ...
        12: Escribir (Diciembre);
FIN_SEGUN;
FIN
```

Codificación en Pascal

```
PROGRAM EscribeMeses (input, output);
VAR i : integer;
BEGIN
  Write ('Introduzca el número de mes');
  Readln(i);
  CASE i OF
    1 : Writeln ('Enero');
    2 : Writeln ('Febrero');
    3 : Writeln ('Marzo');
    4 : Writeln ('Abril');
    5 : Writeln ('Mayo');
    6 : Writeln ('Junio');
    7 : Writeln ('Julio');
    8 : Writeln ('Agosto');
    9 : Writeln ('Septiembre');
    10 : Writeln ('Octubre');
    11 : Writeln ('Noviembre');
    12 : Writeln ('Diciembre');
  END; (* de CASE *)
END.
```

Ejemplo 6.24

El problema que se plantea cuando se introduce un valor en la expresión selector distinto de los de las etiquetas, se puede resolver mediante una sentencia *IF-THEN-ELSE*.

Así el algoritmo del ejemplo anterior podría modificarse:

```
INICIO
  Leer i; (código del mes)
  SI (i<1) o (i>12)
    ENTONCES Escribir Mensaje de error
  SI_NO
    SEGUN i HACER
      1: Escribir (Enero);
      2: Escribir (Febrero);
      ...
      ...
      12: Escribir (Diciembre);
    FIN_SEGUN;
FIN
```

Codificación en Pascal

```

PROGRAM EscribeMeses2 (input, output);
VAR i: integer;
BEGIN
  Writeln ('Introduzca el código del mes');
  Readln (i);
  IF (i<1) OR (i>12)
  THEN
    Writeln ('Código fuera de rango')
  ELSE
    CASE i OF
      1: Writeln ('Enero');
      2: Writeln ('Febrero');
      3: Writeln ('Marzo');
      4: Writeln ('Abril');
      5: Writeln ('Mayo');
      6: Writeln ('Junio');
      7: Writeln ('Julio');
      8: Writeln ('Agosto');
      9: Writeln ('Septiembre');
     10: Writeln ('Octubre');
     11: Writeln ('Noviembre');
     12: Writeln ('Diciembre');
    END;
END.

```

6.9 SENTENCIA GOTO

Es la sentencia de *salto* o *bifurcación incondicional*. Su sintaxis es de la forma:

```
GOTO <etiqueta>
```

donde *etiqueta* es un entero en el rango 1 a 9999 que sirve para marcar la sentencia a la cual se transfiere el control tras ejecutarse el *GOTO*.

El diagrama sintáctico de la sentencia *GOTO* se muestra en la figura 6.10.



Figura 6.20 Diagrama sintáctico de *GOTO*

Su notación en gramática EBNF es:

```
<sentencia goto> ::= GOTO <rótulo>
```

El efecto de la sentencia *GOTO* es un salto incondicional hacia adelante o hacia atrás en la secuencia de ejecución de las instrucciones del programa.

SENTENCIA GOTO

• bifurcación hacia atrás

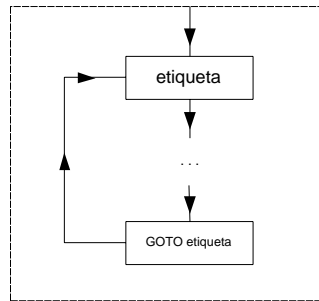


Figura 6.21 Bifurcación (GOTO) hacia atrás

• bifurcación hacia adelante

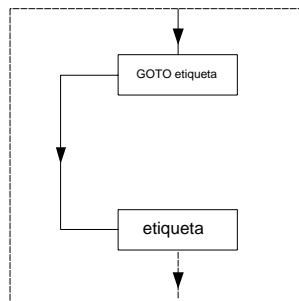


Figura 6.22 Bifurcación (GOTO) hacia adelante

Ejemplo 6.25

Si tenemos:

```
sent1;  
sent2;  
GOTO 55;  
sent4;  
sent5;  
55 : sent6;  
sent7;
```

al ejecutarse la sentencia GOTO 55; se salta directamente a la sentencia etiquetada con 55 , que en este caso es `sent6;`.

El lenguaje Pascal tiene suficientes estructuras de control como para evitar el uso de *GOTO*. Las sentencias *IF*, *WHILE*, *REPEAT*, etc., son mucho más convenientes, hacen los programas más legibles y que el flujo de control sea mucho más claro.

La *Programación Estructurada* se caracteriza por el uso de estructuras de control con entrada única y salida única. El uso del *GOTO* conduce a una programación sin estructurar, haciendo que los programas sean más difíciles de leer, de modificar y de depurar. Por todo ello el uso del *GOTO* debe restringirse a casos excepcionales en los cuales su supresión resulte excesivamente incómoda, como por ejemplo ante una determinada situación de error que nos obligue a abandonar todo el proceso y lo más conveniente sea hacer un salto al final del programa.

El uso del *GOTO* en Pascal debe cumplir las siguientes reglas:

- Deben declararse todas las etiquetas, separándolas por comas, en la sección *LABEL*.
- La sección *LABEL* debe ser la primera del programa o bloque.
- Cada etiqueta solo puede aparecer una vez, y etiquetará a una sola sentencia.
- No se puede saltar al interior de una estructura de control, procedimiento o función; aunque sí está permitido hacerlo desde ellos hacia el exterior.

Ejemplo 6.26

Veamos un programa sencillo usando *GOTO* y su equivalente estructurado al lado. El programa suma una serie de números hasta que se introduce el cero. El mismo problema fue resuelto como ejemplo de uso de la estructura *WHILE*.

<pre>PROGRAM NoEstructurado(input,output); LABEL 10, 20; (* declaración de etiquetas *) CONST ultimo = 0 ; VAR i, suma : integer ; BEGIN suma := 0; Readln (i); 10: IF i = ultimo THEN GOTO 20; suma := suma+i; Readln (i); GOTO 10; 20: Writeln (suma); END.</pre>	<pre>PROGRAM Estructurado(input,output); CONST ultimo = 0 ; VAR i, suma : integer ; BEGIN suma := 0; Readln (i); WHILE i <> ultimo DO BEGIN suma := suma+i; Readln (i); END; Writeln (suma); END.</pre>
---	--

Notas sobre la sentencia *GOTO*

- ✎ Nótese la diferencia entre las etiquetas de la sentencia *GOTO* y las etiquetas de la sentencia *CASE*. Las etiquetas de la sentencia *GOTO* están restringidas a enteros positivos entre 1 y 9999, y deben declararse previamente con la sentencia *LABEL*. Sin embargo las etiquetas de la sentencia *CASE* pueden ser de otros tipos diferentes al tipo entero, y no se declaran antes de usarse.

SENTENCIA GOTO

- ✘ En un programa no puede haber dos sentencias con la misma etiqueta.

Ejemplo 6.27

El siguiente fragmento de código es incorrecto:

```
...  
33: Writeln ('Voy por aquí');  
...  
33: Writeln ('Esto no vale');  
...
```

- ✘ En un programa puede haber varias sentencias *GOTO* que transfieren el control a la misma etiqueta.

Ejemplo 6.28

```
...  
GOTO 30 ;  
...  
GOTO 30 ;  
...  
GOTO 30 ;  
...  
GOTO 30 ;  
...  
30: Writeln ('Estoy aquí');
```

- ✘ Utilización de la sentencia *GOTO* con sentencias compuestas *BEGIN-END*.
 - a) Se puede transferir el control fuera de la sentencia compuesta.

Ejemplo 6.29

```
...  
BEGIN  
...  
IF flag THEN GOTO 33;  
...  
END;  
...  
33: Writeln ('Estoy aquí');  
...
```

- b) Se puede transferir el control con una sentencia *GOTO* dentro de una sentencia compuesta.

Ejemplo 6.30

```
...  
BEGIN  
...
```

ESTRUCTURAS DE CONTROL

```
    IF  flag THEN GOTO 33;
    ...
33: Writeln ('Estoy aquí');
    ...
    END;
    ...
```

c) Se puede transferir el control con una sentencia *GOTO* al *END* de una sentencia compuesta desde la misma sentencia compuesta. Es decir la palabra reservada *END* de la sentencia compuesta puede estar rotulada. La etiqueta debe de ir precedida de la sentencia vacía, esto se consigue colocando un punto y coma en la sentencia anterior a *END*.

d) Es ilegal transferir el control al interior de una sentencia compuesta desde el exterior.

Ejemplo 6.31

El siguiente fragmento de programa es incorrecto:

```
    ...
    IF flag THEN GOTO 33
    ...
    BEGIN
    ...
33: Writeln ('Errorillo');
    ...
    END;
    ...
```

Un programa con esta estructura producirá un error de sintaxis durante la compilación.

- ✘ Las restricciones del uso de la sentencia *GOTO* relacionadas con el ámbito de un programa quedan fuera del propósito de este texto.
- ✘ En lenguaje Pascal (y otros lenguajes estructurados) se desaconseja el uso de la sentencia *GOTO* por las siguientes razones:
 - Altera el flujo claro y secuencial de la lógica del programa
 - Todas las sentencias *GOTO* se pueden suprimir utilizando sentencias *WHILE-DO* y *REPEAT-UNTIL*. Aunque existen excepciones, en las cuales por comodidad se utiliza la sentencia *GOTO*.
 - La programación estructurada se caracteriza por el uso de estructuras de control con una única entrada y una única salida. El uso del *GOTO* conduce a una programación sin estructurar o lo que es lo mismo estructuras de control con múltiples entradas y salidas. Esto hace que los programas sean difíciles de leer, de comprender, de modificar y de depurar.

APLICACION AL CALCULO NUMERICO. DETERMINACION DE RAICES DE ECUACIONES

Por todo lo anterior se debe de evitar el uso de la sentencia *GOTO*; utilícese sólo en casos excepcionales y en bifurcaciones hacia adelante, nunca hacia atrás.

6.10 APLICACION AL CALCULO NUMERICO. DETERMINACION DE RAICES DE ECUACIONES

En una gran variedad de aplicaciones prácticas en física, ingeniería, etc., es necesario calcular las raíces (o ceros) de una ecuación. En el caso más general, dada una función de x , $F(x)$, deseamos encontrar el valor de x tal que

$$F(x) = 0 \quad (1)$$

La función $F(x)$ puede ser, por ejemplo, polinómica:

$$F(x) = 4 \cdot x^5 - x^3 + 3 \cdot x^2 = 0$$

o puede ser una función trascendente (trigonométrica, exponencial o logarítmica):

$$F(x) = e^{-2x} + 2 \cdot \sin x \cdot \cos x = 0$$

Aunque la distinción entre ecuaciones polinómicas y trascendentes suele ser importante en desarrollos matemáticos, a la hora de desarrollar un programa de ordenador para encontrar las raíces de una ecuación no suele ser necesaria esta distinción.

El problema de encontrar los puntos de corte de dos curvas puede reducirse al caso en estudio, encontrar las raíces de una ecuación. Suponiendo las curvas representadas por las ecuaciones:

$$y = f(x)$$
$$y = g(x)$$

Los puntos de corte serán los valores de x tales que:

$$f(x) = g(x)$$

Es decir, serán las raíces de la ecuación:

$$F(x) = f(x) - g(x) = 0$$

En muchos casos, la ecuación (1) tiene una o varias soluciones exactas (ejemplo: ecuación polinómica de primer o segundo grado). En otros, hay que recurrir a métodos de aproximación de las raíces, que pueden descomponerse en dos pasos:

- Encontrar una raíz aproximada
- Mejorar la aproximación, hasta la precisión deseada

En algún caso particular puede interesarnos sólo la primera raíz, o las raíces positivas, o las raíces que cumplan una determinada condición, en aquellos casos en que la ecuación tiene infinitas raíces.

Vamos a comenzar con el caso particular de encontrar un valor más preciso, partiendo de un valor aproximado de una raíz. Utilizaremos algoritmos iterativos basados en aproximaciones sucesivas. Cada paso o aproximación se denomina *iteración*. Si las raíces aproximadas obtenidas en sucesivas iteraciones están cada vez más cerca de la solución, se dice que el algoritmo iterativo *converge*. En caso contrario, el algoritmo *diverge*, y no nos servirá para encontrar la solución.

Quizá el método más sencillo sea el método de *bisección* o *búsqueda dicotómica*. Este es el método utilizado en el ejercicio resuelto 7.7, del capítulo 7. Nosotros abordaremos un método de aproximaciones sucesivas que, modificado, conduce al algoritmo de Newton-Raphson.

Métodos de aproximaciones sucesivas

Para desarrollar este método, reescribiremos la ecuación (1), presentándola de otra forma:

$$x = f(x) \quad (2)$$

Supongamos que conocemos ya una raíz aproximada de la ecuación (2), que llamaremos x_0 . Puede tomarse como siguiente aproximación:

$$x_1 = f(x_0)$$

Y la siguiente aproximación sería:

$$x_2 = f(x_1)$$

Continuando de esta manera, para la n -ésima iteración tendremos:

$$x_n = f(x_{n-1})$$

Si el método converge, para n lo suficientemente grande podremos considerar x_n como una buena aproximación de la solución correcta. Veámoslo gráficamente en la figura 6.23.

La solución de la ecuación (2) viene dada por la intersección de la curva $y = f(x)$ (segundo miembro de la ecuación) y la línea recta $y = x$ (primer miembro). Sea a la abscisa de la solución, desconocida de antemano. Partimos de una solución aproximada, cuya abscisa llamamos x_0 . La siguiente aproximación será $x_1 = f(x_0) = \overline{OA}$. Podemos determinar x_1 trazando una línea horizontal hasta cortar la línea $y = x$ en el punto B .

El valor de $x_2 = f(x_1)$ se obtiene de la misma manera. Prolongando la vertical por x_1 hasta cortar la curva $y = f(x)$ obtenemos $f(x_1) = \overline{OC}$, y trazando una horizontal hasta intersectar la línea $y = x$ determinamos la nueva abscisa, x_2 . Repitiendo este proceso, vamos obteniendo nuevos valores, cada vez más próximos a la solución, a .

Para la curva trazada en la figura 6.17 se ve claramente que el método converge, ya que cada nuevo valor aproximado está más próximo a a que el valor obtenido en la iteración anterior. Obsérvese que hemos elegido una curva tal que $0 < f'(x) < 1$.

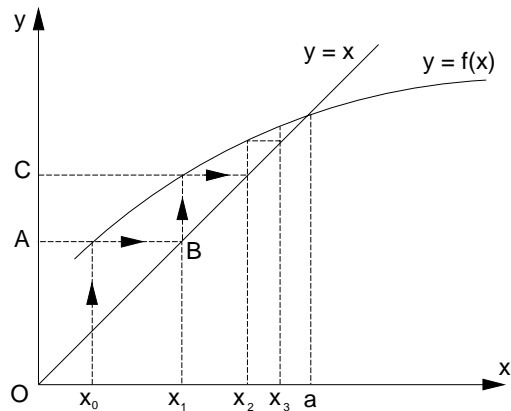


Figura 6.23 Método de aproximaciones sucesivas

Se puede demostrar que:

- Si $|f'(x)| < 1 \rightarrow$ el proceso converge
- Si $|f'(x)| > 1 \rightarrow$ el proceso diverge
- Si en unos puntos se cumple la primera condición y en otros la segunda, el proceso a veces converge y a veces no. Es una cuestión que no ha sido resuelta.

Método modificado de aproximaciones sucesivas

En la figura anterior, figura 6.17, puede observarse que, aunque en cada iteración estamos más cerca de la solución, el método podría mejorarse para que la aproximación sea más rápida. En lugar de hacer

$$x_{n+1} = x_n + \Delta x \quad (3)$$

siendo

$$\Delta x = f(x_n) - x_n \quad (4)$$

podemos obtener los sucesivos valores aproximados mediante

$$x_{n+1} = x_n + \alpha \cdot \Delta x \quad (5)$$

siendo $\alpha > 1$. La mejor selección de α es la mostrada en la figura 6.24, que hace que $x_{n+1} = a$. Determinemos empíricamente el mejor valor de α .

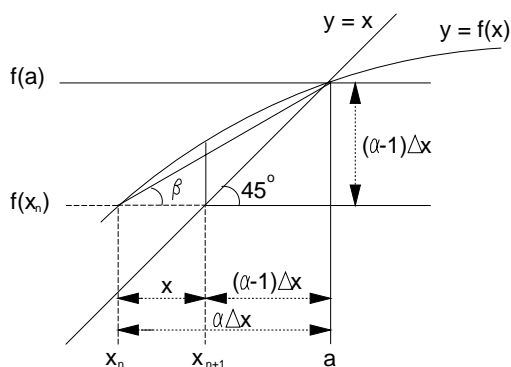


Figura 6.24 Método modificado de aproximaciones sucesivas

En la figura 6.24 se observa que

$$\tan(\beta) = \frac{(\alpha - 1) \cdot \Delta x}{\alpha \cdot \Delta x} = \frac{\alpha - 1}{\alpha} \quad (6)$$

Por otra parte:

$$\tan(\beta) = \frac{f(a) - f(x_n)}{a - x_n}$$

y, usando el teorema del valor medio:

$$\tan(\beta) = f'(\xi) \quad (7)$$

Recordemos que el teorema del valor medio establece que, dados dos puntos, a y b , en una curva $y = f(x)$, en que $f(x)$ tiene derivada continua, la pendiente de la cuerda entre a y b :

$$\frac{f(b) - f(a)}{b - a}$$

es igual a la pendiente de la tangente en algún punto intermedio, $f'(\xi)$, siendo $x_n \leq \xi \leq a$.

De las ecuaciones (6) y (7) se deduce:

$$\alpha = \frac{1}{1 - f'(\xi)} \quad (8)$$

El valor de ξ es desconocido. Una aproximación de $f'(\xi)$ puede ser:

$$f'(\xi) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} = \frac{f(x_n) - x_n}{x_n - x_{n-1}} \quad (9)$$

Geoméricamente equivale a trazar la cuerda entre los puntos $(x_n, f(x_n))$ y $(x_{n-1}, f(x_{n-1}))$ y determinar su intersección con la línea $y = x$.

Hemos obtenido una fórmula para las sucesivas aproximaciones, que se deduce de las ecuaciones (4) y (5):

$$x_{n+1} = x_n + \alpha \cdot (f(x_n) - x_n) \quad (10)$$

El valor de α se deduce de las ecuaciones (8) y (9).

Respecto a la convergencia del método, puede demostrarse que, si $\alpha > 1$, se alargan los pasos, disminuyendo su número, con lo que la convergencia es más rápida.

Método de Newton-Raphson

Este es uno de los algoritmos de cálculo numérico más conocidos para la determinación de raíces de ecuaciones. Se basa en el método de aproximaciones sucesivas que acabamos de estudiar.

A la hora de encontrar una aproximación para ξ , fijamos la condición $x_n \leq \xi \leq a$. Para simplificar los cálculos vamos a elegir $\xi = x_n$. La ecuación (8) se reduce entonces a:

$$\alpha = \frac{1}{1 - f'(x_n)} \quad (11)$$

sustituyendo este valor de α en la ecuación (10), nos queda:

$$x_{n+1} = \frac{f(x_n) - x_n \cdot f'(x_n)}{1 - f'(x_n)} \quad (12)$$

Esta ecuación se corresponde con el método de aproximaciones sucesivas expresado según la ecuación (2), que podemos representar así:

$$x = g(x)$$

siendo

$$g(x) = \frac{f(x) - x \cdot f'(x)}{1 - f'(x)} \quad (13)$$

Este es el método de aproximaciones sucesivas de Newton-Raphson. Generalmente se utiliza en la forma de la ecuación (3):

$$x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)} \quad (14)$$

siendo

$$\Delta x = -\frac{F(x)}{F'(x)}$$

$$F(x) = f(x) - x = 0$$

Se puede demostrar que las condiciones de convergencia son:

- x_0 debe estar suficientemente cerca de una raíz de $F(x) = 0$
- $F''(x)$ no debe ser excesivamente grande
- $F'(x)$ no debe estar muy próxima a cero.

Gráficamente, observando la figura 6.24, la condición $\xi = x_n$ equivale a seleccionar el ángulo β igual a la pendiente de la curva $y = f(x)$ en x_n . El proceso se representa en la figura 6.25. Consiste, para cada iteración, en trazar la tangente a la curva $y = f(x)$ por el punto $(x_n, f(x_n))$ y encontrar su intersección con la línea $y = x$. La abscisa del punto de intersección será el nuevo valor aproximado de la raíz, x_{n+1} . En la siguiente iteración trazaremos la tangente a la curva por el punto $(x_{n+1}, f(x_{n+1}))$, y su intersección con $y = x$ nos determinará x_{n+2} . Repetiremos el proceso hasta que la diferencia entre dos raíces aproximadas consecutivas sea muy pequeña, es decir, menor que la precisión requerida.

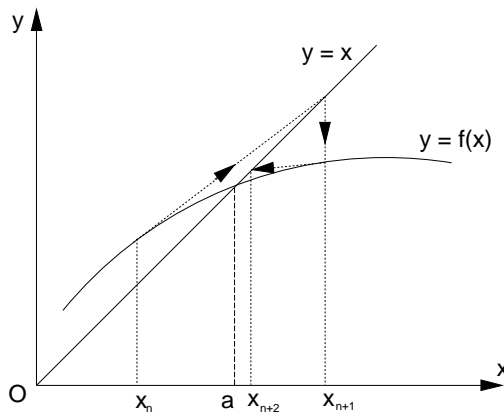


Figura 6.25 Método de Newton-Raphson

Posibles problemas de aplicación del método de Newton-Raphson

El método de Newton-Raphson goza de gran popularidad debido a su rapidez de convergencia y precisión, pero presenta varios inconvenientes que descartan su utilización en algunos casos. Uno de sus inconvenientes es que en cada iteración es necesario evaluar el valor de la función y de su derivada. Esta evaluación puede ser muy sencilla o muy complicada, según la función a tratar en cada caso. Por ejemplo, puede darse el caso de que no tengamos una ecuación para la función, sino una tabla de valores. En este caso se puede utilizar el primer método estudiado de aproximaciones sucesivas, más o menos modificado.

En general, los problemas de aplicación pueden incluirse en uno de los casos siguientes:

- Valor de partida muy lejos de la raíz buscada. Si x_0 está próximo a un mínimo local, es probable que el método falle.
- Si $f'(x_0)$ es próxima a cero (pendiente de la curva casi horizontal), será fácil que en la primera iteración nos alejemos de la proximidad de la raíz buscada.
- El método no sirve para encontrar una raíz en aquellos puntos en que la derivada sea infinita (o muy grande).
- Si la raíz buscada es múltiple, surgen problemas de indeterminación en los cálculos, ya que

$$f(x) = f'(x) = 0$$

$$\Delta x = -\frac{f(x)}{f'(x)} = \frac{0}{0}$$

En el diseño del algoritmo tenemos que incluir la detección de este tipo de problemas. Si aparecen, deberá imprimirse un mensaje de error que además identifique la causa del problema.

Ejemplo 6.32

Escribir un programa para el cálculo de las raíces de una función por el método de Newton.

Análisis: Vamos a resumir el desarrollo matemático anterior en las siguientes fórmulas:

$$x_{n+1} := x_n + Dx$$

$$Dx := -F(x_n) / F'(x_n)$$

El valor de las funciones $F(x)$ y $F'(x)$ suele determinarse mediante subprogramas de tipo *FUNCTION*, que se estudiarán en el capítulo 7. En el ejercicio resuelto 7.7 se determinan las raíces de una ecuación por el método de *bisección* o *búsqueda dicotómica*, mediante un subprograma que recibe como parámetros la función y su derivada. Este sistema permite utilizar el mismo subprograma con distintas funciones $F(x)$. De momento, hasta que sepamos utilizar subprogramas, aplicaremos el método a una función $F(x)$ predeterminada.

Partiendo de un valor inicial, aplicamos las fórmulas anteriores para obtener un nuevo valor más aproximado de la raíz. Este valor será utilizado como valor inicial en la siguiente iteración. Repetimos este proceso hasta que la diferencia entre dos raíces sucesivas sea menor que la precisión requerida.

Incluiremos los siguientes diagnósticos de error:

- Número máximo de iteraciones: Si no se alcanza la solución en un cierto número de iteraciones prefijado, consideramos el método divergente.
- Derivada $F'(x) = 0$. En este caso el proceso diverge.

ESTRUCTURAS DE CONTROL

- Si el incremento o paso, Dx , crece de una iteración a otra, suponemos que el proceso diverge (es la causa más posible, aunque también puede ser debido a un error de redondeo)

Para simplificar el algoritmo, prescindiremos de considerar la posibilidad de raíces múltiples.

Se pueden hacer las siguientes observaciones y recomendaciones:

- ✘ Puede ser muy importante el partir de un valor inicial bueno. Puede ayudarnos una gráfica o tabla de valores de la función.
- ✘ Es necesario controlar la evolución del programa, observar si el método converge. Se consigue con una sentencia que escriba los resultados parciales en cada iteración.

Definimos las siguientes constantes:

ϵ = precisión requerida
 $IterMax$ = N° máximo de iteraciones
 $DxMax$ = Máximo valor del incremento o paso

Utilizamos además dos variables booleanas para controlar el desarrollo del proceso:

$seguir$ = Tomará el valor *false* si detectamos un error que cause divergencia del algoritmo.

$exito$ = Tomará el valor *true* si no se producen problemas y encontramos una solución de la raíz buscada (cuando la diferencia entre dos raíces aproximadas consecutivas sea menor que ϵ).

Algoritmo

```
INICIO
  Leer valor inicial,  $x_0$ .
  Inicializaciones:
     $i := 0$ ; (Número de iteración)
     $seguir := true$ ;
     $exito := false$ ;
     $x := x_0$ ;
     $Dx_0 := DxMax$ ;
  MIENTRAS  $seguir$  HACER
     $i := i + 1$ ;
    Primer diagnóstico de error:
      SI se alcanza el n° máximo de iteraciones
        ENTONCES
          Escribir mensaje de error;
           $seguir := false$ ;
        FIN_SI;
    SI  $seguir$ 
      ENTONCES
        Calcular  $F(x)$ ;
        Calcular  $F'(x)$ ;
```

APLICACION AL CALCULO NUMERICO. DETERMINACION DE RAICES DE ECUACIONES

```
        Segundo diagnóstico de error:
          SI F'(X) = 0
            ENTONCES
              Escribir mensaje de error;
              seguir := false;
          FIN_SI;
    FIN_SI;
  SI seguir
    ENTONCES
      Calcular Dx = -F(x)/F'(x)
      Tercer diagnóstico de error:
        SI |Dx| > |Dx0|
          ENTONCES
            Escribir mensaje de error;
            seguir := false;
          FIN_SI;
    FIN_SI;
  Cálculo del nuevo valor de la raíz, x:
  SI seguir
    ENTONCES
      x := x + Dx;
      Dx0 := Dx;
      Escribir resultados parciales (i, x, Dx, F(x), F'(x))
        (para seguir la evolución del algoritmo)
      ¿Tenemos ya una aproximación suficiente?
        exito := Abs(Dx) < epsilon;
        SI exito
          ENTONCES seguir := false;
        FIN_SI;
    FIN_SI;
  FIN_MIENTRAS;
  SI exito
    ENTONCES Escribir el valor de la solución, xn;
    SI_NO Escribir Mensaje (no se ha llegado a una solución)
  FIN_SI;
FIN
```

Codificación en Pascal

```
PROGRAM Newton (input, output);
(* Programa para calcular una raíz más exacta de una ecuación, partiendo *)
(* de un valor aproximado. Caso particular: F(x) = 4x3-3x2+6x-7=0 *)
(* F'(x) = 12x2-6x+6 = 0 *)
CONST
  epsilon = 1E-8;
  iterMax = 100;
  DxMax = 10;
VAR
  x0, x, Dx, Dx0, F, DF: real;
  i: integer; (* número de iteraciones *)
  seguir, exito: boolean;
BEGIN
  Write('Introduzca un valor inicial aproximado de la raíz:');
  Readln(x0);
  i := 0;
  seguir := true;
```

ESTRUCTURAS DE CONTROL

```

exito := false;
x := x0;
Dx0 := DxMax; (* almacenará el paso anterior para comprobar convergencia *)
Writeln ('Nº iteración      x          Dx          F          DF');
WHILE seguir DO
  BEGIN
    i := i+1;
    (* primer diagnóstico de error *)
    IF (i > iterMax)
      THEN
        BEGIN
          Write('Error: Se ha alcanzado el número máximo ');
          Write('de iteraciones. Proceso divergente. ');
          seguir := false;
        END;
    IF seguir (* Cálculo de F y DF *)
      THEN (* segundo diagnóstico de error *)
        BEGIN
          F := 4*x*x*x-3*x*x+6*x-7;
          DF := 12*x*x-6*x+6;
          IF (Abs(DF)<epsilon)
            THEN
              BEGIN
                Writeln('Error: F'(x)=0');
                seguir := false;
              END;
            END;
          IF seguir (* Cálculo de Dx y tercer diagnóstico de error *)
            THEN
              BEGIN
                Dx := -F/DF;
                IF Abs(Dx) > Abs(Dx0)
                  THEN
                    BEGIN
                      Writeln('Error: Dx creciendo. Proceso divergente');
                      seguir := false;
                    END;
                  END;
          IF seguir (* Cálculo del nuevo valor de la raíz *)
            THEN
              BEGIN
                x := x + Dx;
                Dx0 := Dx;
                Writeln(i:7, x:17:8, Dx:15:8, F:15:8, DF:15:8);
                (* ¿Tenemos ya una buena aproximación? *)
                exito := Abs(Dx) < epsilon;
                IF exito THEN seguir := false;
              END;
          END; (* WHILE seguir *)
        IF exito
          THEN
            BEGIN
              Writeln('Solución hallada: x = ', x:8:2);
              Writeln('Nº de iteraciones = ', i:3);
              Writeln('Precisión = ', epsilon:12:8);
            END;
          Write('Pulse <Intro> para volver al editor...');
          Readln;
        END.

```

6.11 EXTENSIONES DEL COMPILADOR TURBO PASCAL

Sentencia FOR

En Turbo Pascal está permitido que una sentencia altere el valor de la variable de control (contador) dentro de una estructura *FOR*. Sin embargo, los resultados probablemente no serán los esperados. Al salir de la estructura el valor de la variable de control está indefinido, a no ser que se haya salido del bucle mediante una sentencia *GOTO*.

Sentencia GOTO

En Turbo Pascal pueden utilizarse identificadores como etiquetas de sentencias *GOTO*.

Sentencia CASE

La sentencia *CASE* en Turbo Pascal presenta algunas novedades respecto al Pascal estándar: Admite una cláusula *ELSE*, opcional, de manera que si el selector toma un valor no incluido entre las etiquetas-*CASE*, se ejecuta la sentencia (simple o compuesta) correspondiente a la cláusula *ELSE*, si está presente.

Si no aparece cláusula *ELSE* y el selector toma un valor no incluido entre las etiquetas-*CASE*, la ejecución continúa con la siguiente sentencia, a diferencia de Pascal standard, donde el resultado sería indefinido y podría producirse un error en tiempo de ejecución.

Veamos unos ejemplos de utilización de *CASE* en Turbo Pascal:

Ejemplo 6.33

En este ejemplo, si *operador* tomase un valor distinto de las *etiquetas-CASE*, no se ejecutaría ninguna de las sentencias incluidas en la estructura *CASE*, y la ejecución continuaría en la siguiente sentencia.

```
CASE operador OF
  mas   : x := x + y;
  menos: x := x - y;
  por   : x := x * y;
END;
```

Ejemplo 6.34

En este otro ejemplo, sin embargo, si el selector *i* toma un valor no incluido en las *etiquetas-CASE*, se ejecuta la sentencia a continuación de la cláusula *ELSE*, imprimiéndose el correspondiente mensaje.

ESTRUCTURAS DE CONTROL

```
CASE i OF
  0,2,4,6,8: Writeln('Dígito par');
  1,3,5,7,9: Writeln('Dígito impar');
  10..100  : Writeln('Número entre 10 y 100');
ELSE
  Writeln('Número negativo o mayor que 100');
END;
```

Procedimiento Break

Este procedimiento es una versión de la sentencia *break* del lenguaje C. El procedimiento *Break* interrumpe la ejecución de una sentencia *for*, *while* o *repeat*. La más interior que contenga el *Break*, en el caso de sentencias anidadas. No lleva parámetros. Si aparece fuera de una estructura repetitiva se produce un error de compilación. El objetivo de este procedimiento es anular uno de los pocos casos en los que se puede justificar el uso de la instrucción GOTO: la salida de bucles anidados, antes de finalizar su ejecución.

Produce el mismo efecto que un *GOTO* dirigido a la sentencia justo a continuación del *END* (o el *UNTIL*) de una sentencia repetitiva.

Procedimiento Continue

Este procedimiento es una versión de la sentencia *continue* del lenguaje C. El procedimiento *Continue* tampoco lleva parámetros. Su efecto es obligar a la estructura repetitiva en la que está incluida (la más interior en el caso de sentencias anidadas), a interrumpir la iteración actual y continuar inmediatamente con la siguiente iteración. También produce un error de compilación si no está dentro de una estructura repetitiva. También tiene como objetivo evitar el uso de la instrucción GOTO.

Procedimiento Halt

Interrumpe la ejecución del programa y produce la salida al sistema operativo. Su declaración es:

```
Procedure Halt [ (CodigoSalida: Word) ];
```

El parámetro *CodigoSalida* es opcional, especifica el código de salida del programa. *Halt* usado sin parámetros se corresponde con *Halt(0)*.

6.12 CUESTIONES Y EJERCICIOS RESUELTOS

6.1 Supuesta la declaración de variables:

```
VAR
  i: integer;
  a: ARRAY [1..5] OF integer;
```


CUESTIONES Y EJERCICIOS RESUELTOS

considere el siguiente fragmento de programa:

```
FOR i:=1 TO 5 DO
  BEGIN
    Read(i);
    Writeln(i);
    a[i] := 2 * i - 6;
    i := i + 2;
  END;
```

¿Qué sentencia o sentencias de las incluidas en el bucle *FOR* no están permitidas?

¿Por qué?

Nota: No trate de buscarle ningún sentido al programa. Analice únicamente la validez de cada sentencia dentro del bucle *FOR*. El tipo *ARRAY* se estudia en el capítulo 8.

Solución. No son válidas las siguientes sentencias, por alterar el valor de la variable de control del bucle, *i*:

```
  Read(i);
  i := i + 2;
```

6.2 ¿Qué valores escribe por *output* el siguiente programa?

```
PROGRAM Programa (output);
TYPE
  parchis = (azul, amarillo, verde, rojo);
VAR
  i: integer;
  color: parchis;
BEGIN
  FOR i := 1 TO 2 DO
    FOR color := verde DOWNTO azul DO
      Writeln(i, Ord(color));
    END.
  END.
```

Solución. Se escribirán las siguientes parejas de valores:

1	2
1	1
1	0
2	2
2	1
2	0

6.3 Dada la sentencia

```
WHILE condicion DO s;
```

Escribir una sentencia *REPEAT* equivalente. Es necesario utilizar también la sentencia *IF*.

Solución

```

IF condicion
  THEN
    REPEAT
      S
    UNTIL NOT(condicion);

```

6.4 Dado el siguiente fragmento de programa:

```

VAR
  n, cont: integer;
...
...
Readln(n);
cont := 0;
WHILE n >= 0 DO
  BEGIN
    cont := cont + 10;
    n := n - 1;
  END;
Write(cont);
...

```

Escribir un trozo de programa equivalente, para todos los valores de n , usando las sentencias *IF* y *REPEAT*.

Solución

```

VAR
  n, cont: integer;
...
...
Readln(n);
cont := 0;
IF n >= 0
  THEN
    REPEAT
      cont := cont + 10;
      n := n - 1;
    UNTIL n < 0
  Write(cont);
...

```

6.5 Qué valor de v se escribirá por *output* tras ejecutarse las sentencias:

```

v := 0;
FOR i:=1 TO 5 DO
  FOR j:=1 DOWNT0 5 DO
    FOR k:=1 TO 5 DO
      v := v + 1;
    Writeln('v = ',v);

```

CUESTIONES Y EJERCICIOS RESUELTOS

Solución

El segundo bucle *FOR* no se ejecuta, pues el valor inicial es menor que el final. El tercero tampoco, por estar anidado dentro del primero. Por tanto, el valor de *v* no se altera, y se escribirá:

```
v = 0
```

6.6 Sea la declaración:

```
TYPE
  tipo = (c1, c2, c3, c4, c5, c6);
VAR
  i: tipo;
```

Y la sentencia:

```
FOR i := c1 TO c4 DO s;
```

Sustituir esta sentencia por una sentencia *WHILE* equivalente. Recuérdese la existencia de la función *Succ()*;

Solución

```
i := c1;
WHILE c1 <= c4 DO
  BEGIN
    s;
    i := Succ(i);
  END;
```

6.7 ¿Cuáles de los siguientes bucles son sintácticamente correctos? En caso afirmativo, ¿cuántas veces se ejecuta la sentencia *s*?

- a) WHILE false DO s;
- b) WHILE true DO s;
- c) REPEAT s UNTIL false;
- d) REPEAT s UNTIL true;
- e) VAR a: integer;
...
a := 2;
WHILE a:=2 DO sentencia;

Solución

Suponemos que la sentencia *s* no contiene ningún *GOTO*.

ESTRUCTURAS DE CONTROL

- a) Correcto. No se ejecuta ninguna vez.
- b) Correcto. Se ejecuta infinitas veces.
- c) Correcto. Se ejecuta infinitas veces.
- d) Correcto. Se ejecuta una vez.
- e) Incorrecto: `a:=2` no es una comparación, sino una asignación. Suponiendo la comparación correctamente escrita, si en la sentencia `s` no cambia el valor de `a`, el bucle se ejecutará infinitas veces.

6.8 Indicar lo que escribe por *output* el siguiente programa:

```
PROGRAM cuestion(output);
VAR
  i, j, k: integer;
BEGIN
  WHILE false DO
    FOR i:=99 DOWNT0 0 DO
      FOR j:=1 TO 99 DO
        FOR k:=0 TO 9 DO
          IF (i MOD j = k) AND Odd(k)
            THEN Writeln(i, '-', j, '-', k);
        END.
      END.
    END.
  END.
```

Solución

No escribe nada, ya que todas las sentencias están dentro de un bucle `WHILE false DO`, que no se ejecuta ninguna vez.

6.9 Simplifique al máximo el siguiente fragmento de programa:

```
VAR
  ch: char;
...
Read(ch);
ch := Pred(ch);
IF NOT ( Ord(ch) = Ord(Pred(ch))+1 )
  THEN Write(ch)
  ELSE Write(Succ(ch));
```

Solución

```
Read(ch);
Write(ch);
```

6.10 Dadas las siguientes declaraciones:

```
TYPE
  palos = (corazones, diamantes, picas, treboles);
VAR
  naipe: palos;
```

CUESTIONES Y EJERCICIOS RESUELTOS

considere la siguiente pareja de bucles anidados:

```
FOR naipe := corazones TO treboles DO
  WHILE naipe < treboles DO
    BEGIN
      Write(naipe);
      naipe := Succ(naipe);
    END;
  Writeln(Ord(naipe));
```

- a) Si se considera que el bucle es correcto, escriba otro equivalente utilizando exclusivamente bucles *FOR*. Si lo considera incorrecto, indique las razones.
- b) Respecto a la sentencia `writeln(Ord(naipe))` si bien es sintácticamente correcta, es inadmisibles en Pascal estándar, justo a continuación del bucle *FOR*. ¿Por qué?

Solución

- a) **Incorrecciones:**
`write(naipe)` es incorrecta porque no se pueden leer ni escribir variables de tipo enumerado (`naipe`) con *Read*, *Readln*, *Write* o *Writeln*.
`naipe:=Succ(naipe)` es incorrecta, pues `naipe` es la variable de control del bucle *FOR*, y no se puede alterar su valor en el interior del bucle.
- b) Al salir del bucle *FOR*, la variable de control, en Pascal estándar queda indeterminada, por lo que el resultado de la sentencia no será el esperado.

6.11 En algunos lenguajes de programación existe una estructura de bucle de la forma:

```
REPETIR
  Sent1_1;
  Sent1_2;
  ...
  SALIR SI condicion;
  Sent2_1;
  Sent2_2;
  ...
SIEMPRE;
```

¿Cómo implementaría esta estructura de control en Pascal?

Solución

```
REPEAT
  Sent1_1;
  Sent1_2;
  ...
  IF NOT (condicion)
  THEN
```

ESTRUCTURAS DE CONTROL

```
BEGIN
  Sent2_1;
  Sent2_2;
  ...
END;
UNTIL condicion;
```

Nota: El valor de `condicion` no debe ser alterado por las sentencias `Sent2_1;` `Sent2_2;` ..., sino que debe cambiar en el primer grupo (`Sent1_1;` `Sent1_2;` ...).

6.12 En algunos lenguajes de programación existe una estructura de control de la forma:

```
DESDE k:= valor inicial HASTA valor final PASO paso
HACER
  sentencia;
FIN_DESDE;
```

donde `k`, `valor inicial` y `valor final` son de tipo *real*. La variable `k` toma inicialmente el `valor inicial` y, tras cada ejecución del bucle, se incrementa en el valor dado por `paso` hasta que se supera el `valor final`, en cuyo caso se sale del bucle. Nótese que si en el último incremento coincide exactamente con el `valor final`, se ejecuta para ese valor. Dada la pareja de bucles anidados:

```
DESDE i:= vi1 HASTA vf1 PASO p1 HACER
  DESDE j:= vi2 HASTA vf2 PASO p2 HACER
    sentencia;
  FIN_DESDEj;
FIN_DESDEi;
```

Se pide construir una estructura equivalente en Pascal

Solución

```
CONST
  epsilon = 1e-5;
VAR
  c1, c2, p1, p2, vi1, vi2, vf1, vf2: real;
  i, j, n1, n2: integer;
...
BEGIN
  ...
  (* n° de veces que se ejecutan los bucles *)
  n1 := Trunc((vf1-vi1+1)/p1);
  n2 := Trunc((vf2-vi2+1)/p2);
  c1 := vi1;
  FOR i:=1 TO n1 DO
    BEGIN
      IF ((c1<vf1) OR (Abs(c1-vf1) < epsilon))
      THEN
        BEGIN
          c2 := vi2;
          FOR j:=1 TO n2 DO
            BEGIN
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
                IF ( (c2<vf2) OR (Abs(c2-vf2)<epsilon) )
                  THEN
                    sentencia;
                    c2 := c2+p2;
                END;
            END;
        c1 := c1+p1;
    END;
    ...
```

Nota: Las variables p_1 , p_2 , c_1 , c_2 , vi_1 , vf_1 , vi_2 , vf_2 , no deben ser alteradas en sentencia. No está permitido modificar el valor de i , j , n_1 , n_2 en sentencia.

6.13 Dado el siguiente programa:

```
PROGRAM Cuestion(input, output);
TYPE dl =(c1, c2, c3, c4);
VAR dato: dl;
BEGIN
    dato := c2;
    WHILE true DO
        WHILE (Ord(dato) <= 1) DO
            Writeln(Ord(dato));
        END;
    END.
```

Se pide:

- ¿Qué salida se obtiene de su ejecución?
- Escribir un programa equivalente utilizando exclusivamente sentencias *REPEAT*.
- Escribir un programa equivalente utilizando exclusivamente sentencias *FOR*.

Solución

- La salida serán infinitas líneas conteniendo el valor 1:

```
1
1
1
...
1
```

Como $\text{Ord}(\text{dato})$ no cambia dentro del segundo bucle, siempre vale $\text{Ord}(c_2)$, es decir 1. La condición $\text{Ord}(\text{dato}) \leq 1$ será siempre cierta. Ambos bucles son infinitos.

ESTRUCTURAS DE CONTROL

b)

```
PROGRAM Cuestion(input, output);
TYPE d1 =(c1, c2, c3, c4);
VAR dato: d1;
BEGIN
  dato := c2;
  REPEAT
    Writeln(Ord(dato));
  UNTIL false;
END.
```

c) No es posible, ya que un bucle *FOR* se repite un número finito de veces.

6.14 Sea la declaración:

```
VAR car: char;
    i, j: integer;
```

y el siguiente fragmento de programa:

```
...
car := 'as';
FOR j:=5 DOWNTO i DO
  BEGIN
    IF (car<>' ') AND (car<>'..')
      THEN Write(car);
    Readln(car);
  END;
...
```

- a)** ¿Es sintácticamente correcto? Conteste razonadamente.
- b)** Hacer las modificaciones oportunas para que pueda leer una línea de 80 caracteres.

Solución

a) No es sintácticamente correcto. La variable *car* solo puede contener un carácter. La sentencia *car := 'as'* es incorrecta.

b)

```
...
i := 0;
FOR j:=79 DOWNTO i DO
  BEGIN
    Read(car);
    IF (car<>' ') AND (car<>'..')
      THEN Write(car);
  END;
...
```


CUESTIONES Y EJERCICIOS RESUELTOS

6.15 Considere los bucles siguientes:

```
VAR
  a, b: boolean;
```

```
1) WHILE a AND b DO
    sentencia;
```

```
2) WHILE a DO
    WHILE b DO
    sentencia;
```

Conteste:

- ¿Son totalmente equivalentes? Razone su respuesta
- ¿Existe alguna combinación de valores de *a* y *b* para la cual alguno de los bucles (o ambos) se ejecute indefinidamente?

Solución

No son totalmente equivalentes:

- Para *a=false* y *b=false* son equivalentes, pues no se ejecuta ninguno de los dos.
- Para *a=false* y *b=true* también son equivalentes, no se ejecuta *sentencia* en ningún caso.
- Para *a=true* y *b=false* el bucle (1) no se ejecuta, pues la condición compuesta es falsa, mientras que en el caso (2) se ejecuta el primer *WHILE* indefinidamente, pero sin entrar en el segundo. No se producirá ningún resultado en ninguno de los dos casos, pues no se ejecuta *sentencia*, pero no son equivalentes.
- Para *a=true* y *b=true* ambos bucles se ejecutan indefinidamente, si el valor de las variables *a* y *b* no cambia es *sentencia*.

6.16 Dado el siguiente fragmento de programa

```
PROGRAM QueSera(output);
VAR c1, c2, c3: boolean;
BEGIN
  ...
  REPEAT WHILE c2 DO BEGIN
    c3 := true;
    c2 := NOT c2;
  END;
  REPEAT c2 := false;
  UNTIL c3;
  UNTIL c1;
  Write(c1, ' ', c2, ' ', c3);
  ...
END;
```

ESTRUCTURAS DE CONTROL

Se pide determinar todas las posibles combinaciones de valores iniciales para c_1 , c_2 , c_3 que hagan que la salida del programa sea:

TRUE FALSE TRUE

Solución. Se muestra en la tabla siguiente:

Iniciales			Finales		
c1	c2	c3	c1	c2	c3
T	T	T	T	F	T
T	T	F	T	F	T
T	F	T	T	F	T
T	F	F		*	
F	T	T		*	
F	T	F		*	
F	F	T		*	
F	F	F		*	

* Bucle infinito.

6.17 Simplifique al máximo el siguiente bucle de lectura, transformándolo en otro equivalente.

```

VAR ch: char;
WHILE NOT Eof DO
  BEGIN
    Read(ch);
    Write(ch);
    IF Eoln OR (ch='a')
      THEN Readln;
    IF NOT(Eof)
      THEN
        REPEAT
          Read(ch);
          Write(ch);
          IF Eoln OR (ch='a')
            THEN Readln;
        UNTIL Eof;
  END;

```

Solución

```

WHILE NOT(Eof) DO
  BEGIN
    Read(ch);
    Write(ch);
    IF Eoln OR (ch='a')
      THEN Readln;
  END;

```

CUESTIONES Y EJERCICIOS RESUELTOS

- 6.18** Escribir un programa que lea una línea y cuente el número de comas y el número de letras mayúsculas que contiene. Supondremos que la línea de entrada finaliza al teclear el carácter punto ('.').

Solución

```
PROGRAM Contar (input,output);
CONST
  finLinea = '.'; (* carácter de fin de línea *)
VAR
  contMayusculas, contComas: integer; (* contadores *)
  ch: char ;
BEGIN (* programa *)
  Writeln ('Introduzca una línea terminada por ', finLinea);
  contComas := 0;
  contMayusculas := 0;
  Read (ch); (* leemos un carácter *)
  Write(ch); (* devolvemos el eco *)
  WHILE ch <> finLinea DO
  BEGIN
    IF ch = ','
    THEN contComas:= contComas + 1
    ELSE IF (ch >='A') AND (ch<='Z') (* es mayúscula *)
    THEN contMayusculas:= contMayusculas + 1;
    Read (ch);
    Write(ch);
  END; (* WHILE *)
  Writeln;
  Writeln ('Total comas =', contComas);
  Writeln ('Total mayúsculas =', contMayusculas);
END. (* programa *)
```

- 6.19** El siguiente programa escribe una lista de todos los caracteres imprimibles junto con su ordinal. En código ASCII, los caracteres con ordinal inferior al 32 son caracteres de control, es decir, no imprimibles.

Solución

```
PROGRAM ConjuntoCaracteres (output);
CONST
  primerCar = 32 ; (* primer carácter imprimible ASCII *)
  ultimoCar =127 ; (* el 127 tampoco es imprimible *)
VAR
  i : integer ;
BEGIN (* programa *)
  i:= primerCar ;
  WHILE i < ultimoCar DO
  BEGIN
    Writeln ('Ordinal= ',i, ': Carácter= ', Chr(i));
    i:= i + 1;
  END;
END. (* programa *)
```

- 6.20** Escribir un programa en Pascal que cuente el número de veces que aparece un carácter, leído por teclado, en una frase acabada en punto, introducida también por teclado.

Solución

```
PROGRAM Caracter (input,output);
CONST
  punto='.';
VAR
  letra, car:char;
  s:integer;
BEGIN
  (* Inicialización del contador a cero *)
  s:=0;
  Write('Deme el carácter a contar: ');
  Readln(car);
  Writeln('Introduzca una frase acabada en un punto. ');
  Read(letra);
  WHILE letra<>punto DO
    BEGIN
      IF letra=car THEN s:=s+1;
      Read(letra)
    END;
  Readln;
  Writeln;
  Writeln('El número de apariciones de ',car,' es ',s);
  Writeln ('Pulse <Return> para volver al Editor');
  Readln
END.
```

- 6.21** Dada una frase introducida por teclado y acabada en un punto, escribir un programa que, mediante sentencias *IF-THEN-ELSE* anidadas, cuente:

- El número de caracteres leídos
- El número de apariciones de cada vocal mayúscula y minúscula
- El número de espacios en blanco
- El número de comas
- El número total de vocales

Solución¹³

```
PROGRAM CuentaVocales (input,output);
CONST
  final = '.';
VAR
  letra:char;
  n,a,e,i,o,u,blancos,comas,am,em,im,om,um,v :integer;
BEGIN
  n:=0;a:=0;e:=0;i:=0;o:=0;u:=0;
  blancos:=0;comas:=0;am:=0;em:=0;im:=0;om:=0;um:=0;
  Writeln ('Introduzca una frase acabada en un punto. ');
  Read (letra);
```

13 Otra solución es utilizando la estructura multialternativa CASE

CUESTIONES Y EJERCICIOS RESUELTOS

```
WHILE letra <> final DO
BEGIN
  n:=n+1;
  IF letra='a'
  THEN
    a:=a+1
  ELSE
    IF letra='e'
    THEN
      e:=e+1
    ELSE
      IF letra='i'
      THEN
        i:=i+1
      ELSE
        IF letra='o'
        THEN
          o:=o+1
        ELSE
          IF letra='u'
          THEN
            u:=u+1
          ELSE
            IF letra=' '
            THEN
              blancos:=blancos+1
            ELSE
              IF letra=','
              THEN
                comas:=comas+1
              ELSE
                IF letra='A'
                THEN
                  am:=am+1
                ELSE
                  IF letra='E'
                  THEN
                    em:=em+1
                  ELSE
                    IF letra='I'
                    THEN
                      im:=im+1
                    ELSE
                      IF letra='O'
                      THEN
                        om:=om+1
                      ELSE
                        IF letra='U'
                        THEN
                          um:=um+1;

  Read(letra)
END;
Readln;
Writeln;
Writeln('El número de caracteres leídos es ',n);
Writeln('El número de a minúsculas es ',a);
Writeln('El número de e minúsculas es ',e);
Writeln('El número de i minúsculas es ',i);
Writeln('El número de o minúsculas es ',o);
Writeln('El número de u minúsculas es ',u);
Writeln('El número de blancos es ',blancos);
Writeln('El número de comas es ',comas);
Writeln('El número de A mayúsculas es ',am);
Writeln('El número de E mayúsculas es ',em);
Writeln('El número de I mayúsculas es ',im);
Writeln('El número de O mayúsculas es ',om);
Writeln('El número de U mayúsculas es ',um);
```

ESTRUCTURAS DE CONTROL

```
v:=a+e+i+o+u+am+em+im+om+um;
Writeln('*****');
Writeln('      LAS VOCALES SON ',v );
Writeln('*****');
Writeln ('Pulse <Return> para volver al Editor');
Readln
END.
```

- 6.22** Dada una frase introducida por teclado y acabada en un punto, escribir un programa que cuente el número de apariciones de la cadena *la*.

Solución

```
PROGRAM Character (input,output);
CONST
    punto='.';
    car1='l';
    car2='a';
    car1m='L';
    car2m='A';
VAR
    letra,letral,letra2 :char;
    s:integer;
BEGIN
    (* Inicialización del contador a cero *)
    s:=0;
    Writeln('Introduzca una frase letra a letra y pulsando <INTRO> ');
    Readln(letra);
    letral:=letra;
    WHILE (letra <> punto) DO
        BEGIN
            Readln(letra);
            letra2:=letra;
            IF ( (letral = car1) OR (letral = car1m) ) AND
                ( (letra2 = car2) OR (letra2 = car2m) ) )
                THEN s:=s+1;
            letral:=letra2
        END;
    Writeln;
    Writeln('El número de apariciones de la cadena <la> es ',s)
    Write ('Pulse <Return> para volver al Editor');
    Readln;
END.
```

- 6.23** Dada una frase introducida por teclado y acabada en un punto, escribir un programa que cuente el número de apariciones de una subcadena de dos letras introducida también por teclado.

Solución

```
PROGRAM Character (input, output);
CONST
    punto='.';
VAR
    letra,letral,letra2, car1,car2:char;
    s:integer;
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
BEGIN
(* Inicialización del contador a cero *)
s:=0;
Write('Deme el primer carácter de la cadena a buscar: ');
Readln(car1);
Write('Deme el segundo carácter de la cadena a buscar: ');
Readln(car2);
Write('Introduzca una frase acabada en punto: ');
Read(letra);
letra1:=letra;
WHILE (letra<>punto) DO
  BEGIN
    Read(letra);
    letra2:=letra;
    IF (letra1=car1)AND(letra2=car2) THEN s:=s+1;
    letra1:=letra2
  END;
Readln;
Writeln;
Writeln('El número de apariciones de ',car1,car2,' es ',s);
Writeln('Pulse <Return> para volver al Editor');
Readln
END.
```

- 6.24** Escribir un programa Pascal que genere todas las matrículas de automóviles de la serie de dos letras correspondientes a la provincia de Asturias.

Para ello tenga en cuenta las siguientes observaciones:

- a) Una matrícula se compone de la letra *O* seguida de un guión, una serie de cuatro dígitos (desde el *0000* al *9999*), otro guión y una serie de dos letras mayúsculas.

Ejemplo: *O-6552-AC*

- b) Tenga en cuenta que si decide utilizar una variable integer para representar al número, los números menores que 1000 deben aparecer con los ceros a la izquierda necesarios.

Así, por ejemplo, la matrícula *O-15-AC* no sería correcta. Debería ser *O-0015-AC*. (Con esto no se está diciendo que sea obligatorio utilizar una variable integer).

- c) Las siguientes series de matrículas no están permitidas:

- Las que contengan doble vocal: Ejemplo: *O-7865-AE*
- Las que contengan las letras *Q*, *R* ó *Ñ*.

En definitiva, el rango de matrículas pedido es:

O-0000-AB
O-0001-AB
.....
O-9998-ZZ
O-9999-ZZ

Solución

```

PROGRAM Matriculas(output);
VAR
  i, j, k, l, m, n: char;
BEGIN
  FOR m:= 'A' TO 'Z' DO
    IF (m<>'Q') AND (m<>'R')
      THEN
        FOR n:= 'A' TO 'Z' DO
          IF (n<>'Q') AND (n<>'R')
            THEN
              IF NOT( ((m='A')OR(m='E')OR(m='I')OR(m='O')OR(m='U')) AND
                ((n='A')OR(n='E')OR(n='I')OR(n='O')OR(n='U')) )
                THEN
                  FOR i:= '0' TO '9' DO
                    FOR j:= '0' TO '9' DO
                      FOR k:= '0' TO '9' DO
                        FOR l:= '0' TO '9' DO
                          Writeln('O-', i:1, j:1, k:1, l:1, '-', m:1, n:1);
                        Writeln ('Pulse <Return> para volver al Editor');
                      END;
                    END;
                  END;
                END;
              END;
            END;
          END;
        END;
      END;
    END;
  Readln
END.

```

6.25 Realizar un programa que calcule el área de un triángulo, a partir

- De la base y la altura
- De sus tres lados
- De dos de sus lados y el ángulo que forman entre ellos

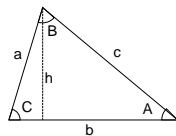


Figura 6.26 Ejemplo de cálculo del área de un triángulo

Fórmulas necesarias:

$$a) \quad S = \frac{b \cdot h}{2}$$

$$b) \quad S = \sqrt{(p \cdot (p - a) \cdot (p - b) \cdot (p - c))}$$

siendo p el semiperímetro:

$$p = \frac{a + b + c}{2}$$

c) Aplicando el teorema del Coseno:

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos C$$

CUESTIONES Y EJERCICIOS RESUELTOS

y aplicando la fórmula del apartado *b*).

Solución

```
PROGRAM Triangulo (input,output);
VAR
  a,b,c,p,h,alfa,s :real;
  opcion :integer;
  flag : boolean;
BEGIN
  flag := true;
  REPEAT
    Writeln;
    Writeln('***** CALCULO DEL AREA DE UN TRIANGULO *****');
    Writeln;
    Writeln('METODOS :');
    Writeln;
    Writeln(' 1 - A partir de la base y la altura');
    Writeln;
    Writeln(' 2 - A partir de sus tres lados ');
    Writeln;
    Writeln(' 3 - A partir de dos lados y el ángulo que forman ');
    Writeln;
    Writeln(' 4 - FIN      ');
    Writeln;
    Write('Elija método : ');
    Readln( opcion);
    IF (opcion < 1) OR (opcion > 4)
      THEN
        Writeln ('Se ha equivocado')
      ELSE
        CASE opcion OF
          1 : BEGIN
              Write('Introduzca base y altura ');
              Readln(b,h);
              s:= b*h/2;
              Writeln('Superficie=',s:10:5)
            END;
          2 : BEGIN
              Write('Introduzca los tres lados ');
              Readln(a,b,c);
              p:=(a+b+c)/2;
              s:=Sqrt(p*(p-a)*(p-b)*(p-c));
              Writeln('Superficie',s:10:5)
            END;
          3 : BEGIN
              Write('Introduzca los dos lados ');
              Readln(a,b);
              Write('Introduzca el ángulo que forman');
              Write(' (en grados sexagesimales) ');
              Readln(alfa);
              alfa:= alfa * 3.141592 / 180;
              c:=Sqrt(Sqr(a)+Sqr(b)-2*a*b*Cos(alfa));
              p:=(a+b+c)/2;
              s:=Sqrt(p*(p-a)*(p-b)*(p-c));
              Writeln('Superficie = ',s:10:5)
            END;
          4 : BEGIN
              Writeln('***** FIN *****');
              flag :=false
            END;
        END (* CASE *)
    UNTIL flag = false
  END.
```

- 6.26** Escribir un programa que calcule el valor de la suma $1 + 1/2 + 1/3 + \dots + 1/n$ para un valor de n determinado que se lee por teclado, utilizando una estructura *FOR*.

Solución

```
PROGRAM Sumatorio(input, output);
VAR
  i,n : integer;
  suma: real;
BEGIN
  Write(' Introduzca n: ');
  Readln(n);
  suma := 0;
  FOR i := 1 TO n DO
    suma := suma + 1/i;
  Writeln(' para ', n:7, ' la suma es: ', suma:10:4)
END.
```

- 6.27** Diseñar un programa que imprima la tabla de logaritmos decimales, para números comprendidos entre el 1 y el 9.

Solución

```
PROGRAM TablaDeLogaritmos (output);
(* Este programa escribe una tabla de logaritmos decimales del 1 al 9
*)
CONST
  minimo = 1;
  maximo = 9;
  blancos4='    ';
  blancos5='      ';
VAR
  i, j :integer; (* índices de los bucles *)
  x, y :real; (* x es una variable auxiliar *)
          (* y es el valor del logaritmo *)
BEGIN
  Writeln('***** TABLA DE LOGARITMOS DECIMALES *****');
  Writeln;
  (* escribe la cabecera de la tabla *)
  Write(blancos5);
  x:=0.0;
  FOR i:=1 TO 10 DO
    BEGIN
      Write(blancos4,x:0:1);
      x:=x+0.1;
    END;
  Writeln;
  (* Escribe la tabla línea a línea *)
  FOR i:=minimo TO maximo DO
    BEGIN
      Writeln;
      Writeln;
      x:=i;
      y:=Ln(i)/Ln(10);
      Write(x:3:1, y:10:4);
      FOR j:=1 TO 9 DO
        BEGIN
          x:=x+0.1;
          y:=Ln(x)/Ln(10);
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
Write(y:7:4)
END;
END;
Writeln;
Writeln ('Pulse <Return> para volver al Editor');
Readln
END.
```

- 6.28 La fórmula para calcular la superficie encerrada por un contorno poligonal de n vértices es la siguiente:

$$S = \frac{|\sum_{i=1}^{n+1} (x_i \cdot y_{i-1} - x_{i-1} \cdot y_i)|}{2}$$

en la que (x_i, y_i) son los vértices del polígono. El vértice $n+1$ coincide con el primero. Escribir un programa para calcular la superficie de un polígono a partir de las coordenadas de sus vértices

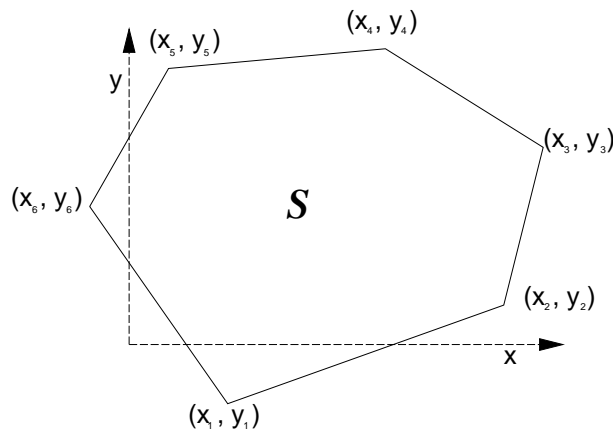


Figura 6.27 Ejemplo de cálculo de la superficie de un polígono

Solución

```
PROGRAM SuperficieDePoligonos (input,output);
VAR
  x,y,x1,y1,xi,yi,acu,s :real;
  i,n:integer;
BEGIN
  acu:=0;
  Write('Introduzca el número de vértices del polígono ');
  Readln(n);
  Write('Introduzca las coordenadas del primer vértice ');
  Readln(x1,y1);
  x:=x1;
```

ESTRUCTURAS DE CONTROL

```
y:=y1;
FOR i:=2 TO n DO
  BEGIN
    xi:=x;
    yi:=y;
    Write('Introduzca las coordenadas del vértice nº ',i:3,' : ');
    Readln(x,y);
    acu:=acu+xi*y-x*yi
  END;
acu:=acu+x*y1-x1*y;
s:=Abs(acu)/2;
Writeln('La superficie del polígono es ',s:10:4);
Writeln ('Pulse <Return> para volver al Editor');
Readln
END.
```

6.29 Escribir un programa para, dada una fecha, escribir la fecha del día siguiente.

Solución

```
PROGRAM FechaSiguiente (input,output);
(* Dada una fecha, determina la del día siguiente. *)
TYPE
  dias = 1..31;
  meses = 1..12;
VAR
  dia, diaSiguiente,diasPorMes: dias;
  mes,mesSiguiente:meses;
  any,anysiguiente:integer;
BEGIN
  (* Lectura de la fecha de entrada *)
  Write ('Dame una fecha en el formato dd mm aa : ');
  Readln (dia,mes,any);
  (* Determinación del número de días del mes leído *)
  CASE mes OF
    1,3,5,7,8,10,12: diasPorMes := 31;
    4,6,9,11 : diasPorMes := 30;
    2: IF (any MOD 4 = 0) AND NOT (any MOD 100 = 0)
        THEN diasPorMes := 29 (* bisiesto*)
        ELSE diasPorMes := 28;
  END;
  (* Proceso de incrementar fecha *)
  mesSiguiente := mes;
  anysiguiente := any;
  IF dia<>diasPorMes
  THEN
    diaSiguiente := dia + 1 (* No es fin de mes *)
  ELSE
    BEGIN
      diaSiguiente := 1; (* Es fin de mes *)
      IF mes <>12
      THEN
        mesSiguiente := mes +1 (* No es fin de año *)
      ELSE
        BEGIN
          mesSiguiente := 1;
          anysiguiente := any +1
        END
      END
  END;
END;
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
(* Salida de la fecha del día siguiente *)
Writeln ('La siguiente es: ',diaSiguiente,'-',mesSiguiente,'-
',anySiguiente);
END.
```

- 6.30** Escribir un programa que simule una calculadora sencilla, con dos operandos y los cuatro operadores aritméticos básicos.

Solución

```
PROGRAM Operaciones (input,output);
VAR
  x1,x2 : real;
  operador : char;
BEGIN
  Write('Introduzca el primer operando ');
  Readln(x1);
  Write('Introduzca el operador ');
  Readln(operador);
  Write('Introduzca el segundo operando ');
  Readln(x2);
  CASE operador OF
    '+' : Writeln(x1:15:7,' + ',x2:15:7,' = ',(x1+x2):15:7);
    '-' : Writeln(x1:15:7,' - ',x2:15:7,' = ',(x1-x2):15:7);
    '*' : Writeln(x1:15:7,' * ',x2:15:7,' = ',(x1*x2):15:7);
    '/' : Writeln(x1:15:7,' / ',x2:15:7,' = ',(x1/x2):15:7);
  END;
END.
```

- 6.31** Este programa calcula el valor de π sabiendo que la suma de la serie:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

tiende a $\pi/4$, con un error menor de una milésima. Ilustra el uso de la sentencia *REPEAT* y el tipo real. Escribe π en pantalla utilizando la función *chr()*.

Solución

```
PROGRAM CalculoDePiConRepeat (output);
CONST
  epsilon = 1E-4;
VAR
  denominador, signo : integer;
  pi, termino : real;
BEGIN
  pi := 1;
  denominador := 3;
  signo := -1;
  termino := signo * (1/denominador);
```

ESTRUCTURAS DE CONTROL

```
REPEAT
    pi := pi + termino;
    signo := - signo;
    denominador := denominador +2;
    termino := signo * (1/denominador);
UNTIL Abs(termino) < epsilon;
Writeln (chr(227),' = ', (4 * pi):6:4);
Writeln ('Pulse <Return> para volver al editor');
Readln;
END.
```

6.32 A continuación escribimos el programa anterior utilizando la sentencia *WHILE*.

Solución

```
PROGRAM CalculoDePi (output);
CONST
    epsilon = 1E-4;
VAR
    denominador, signo : integer;
    pi, termino : real;
BEGIN
    pi := 1;
    denominador := 3;
    signo := -1;
    termino := signo * (1/denominador);
    WHILE Abs(termino) > epsilon DO
        BEGIN
            pi := pi + termino;
            signo := - signo;
            denominador := denominador +2;
            termino := signo * (1/denominador)
        END;
    Writeln (chr(227),' = ', (4 * pi):6:4);
    Writeln ('Pulse <Return> para volver al editor');
    Readln;
END.
```

6.33 Escribir un programa que dado un entero comprendido entre 0 y 3999, lo traduzca a su equivalente en números romanos.

Solución

```
PROGRAM Romanos (input,output);
VAR
    a,b,c,d: integer;
BEGIN
    Writeln ('Escribe una cifra entre 0 y 3999 ');
    Writeln ('Para escribir 2, teclee 0 0 2');
    Write('Separa cada dígito por un espacio --> ');
    Readln(a,b,c,d);
    Writeln;
    Write(a,b,c,d,' en numeros romanos es: ');
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
IF a>0
  THEN
    CASE a of
      1:Write('M');
      2:Write('MM');
      3:Write('MMM');
    END;
IF b>0
  THEN
    CASE b OF
      1:Write('C');
      2:Write('CC');
      3:Write('CCC');
      4:Write('CD');
      5:Write('D');
      6:Write('DC');
      7:Write('DCC');
      8:Write('DCCC');
      9:Write('CM');
    END;
IF c>0
  THEN
    CASE c OF
      1:Write('X');
      2:Write('XX');
      3:Write('XXX');
      4:Write('XL');
      5:Write('L');
      6:Write('LX');
      7:Write('LXX');
      8:Write('LXXX');
      9:Write('XC');
    END;
IF d>0
  THEN
    CASE d OF
      1:Write('I');
      2:Write('II');
      3:Write('III');
      4:Write('IV');
      5:Write('V');
      6:Write('VI');
      7:Write('VII');
      8:Write('VIII');
      9:Write('IX');
    END;
END.
```

Nota: Obsérvese que el programa se puede simplificar, ya que para $b>0$, $c>0$ y $d>0$ tenemos bloques muy similares. También se puede mejorar la entrada de datos, cuando se estudien las cadenas de caracteres, en el capítulo 8.

6.34 Este programa lee un número como una secuencia de caracteres, y lo convierte en un número de tipo real.

Solución

```

PROGRAM Conversion ( input, output);
CONST
    cero = '0';
    nueve = '9';
    punto = '.';
    base = 10;
VAR
    car: char;
    resultado: real;
    decimales: integer;
BEGIN
    resultado := 0;
    Write ('Introduzca un número pulsando <Intro> después de cada')
    Write (' cifra y del punto decimal. ');
    Write (' Acaba con cualquier caracter que no sea una cifra: ');
    Readln (car);
    (* Lectura y calculo de la parte entera *)
    WHILE (cero <= car) AND (car <= nueve) DO
        BEGIN
            resultado := base * resultado + ord(car) - ord(cero);
            Readln (car);
        END;
    (* Lectura y calculo de la parte decimal *)
    IF car = punto
    THEN
        BEGIN
            decimales := 0;
            Readln (car);
            WHILE (cero <= car) AND (car <= nueve) DO
                BEGIN
                    resultado := resultado * base + ord(car) - ord(cero);
                    Readln (car);
                    decimales := decimales + 1
                END;
            WHILE decimales > 0 DO
                BEGIN
                    resultado := resultado / base;
                    decimales := decimales - 1;
                END
            END;
        END;
    Writeln;
    Writeln ('Resultado = ', resultado:10:5)
END.

```

6.35 Este programa calcula el número e según la expresión:

$$\begin{aligned}
 & 2 + \frac{1}{1 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \dots}}}}} \\
 & \dots
 \end{aligned}$$

con una precisión de una décima.

CUESTIONES Y EJERCICIOS PROPUESTOS

Solución

```
PROGRAM CalculoDelNumero_e (output);
VAR
    i: integer;
    denominador, e: real;
BEGIN
    denominador := 1;
    i := 8;
    WHILE I>1 DO
        BEGIN
            denominador := 1 + 1 / (i + 1 / denominador);
            i := i - 2;
        END;
    e := 2 + 1 / denominador;
    Writeln (e:6:4);
    Writeln ('Pulse <Return> para volver al editor');
    Readln
END.
```

6.13 CUESTIONES Y EJERCICIOS PROPUESTOS

6.36 Sea la declaración:

```
VAR
    i, j: integer;
    a, b: real;
```

y el fragmento de programa:

```
...
FOR i:=1 TO 2 DO
    BEGIN
        a := a+i;
        FOR j:=3 DOWNT0 i DO
            BEGIN
                b := b+j;
                Read(i);
                a:=a+b;
                i:=j;
            END;
        b := 0;
    END;
Writeln(a, b);
...
```

Se pide:

- Ver si el fragmento de programa es sintácticamente correcto. Si no lo es, modificar las sentencias necesarias.
- Escribir los valores de salida para a y b.

6.37 Escribir un bucle *REPEAT* para la lectura de los coeficientes (a, b, c) de una ecuación de segundo grado $(ax^2+bx+c=0)$, de manera que en el caso de que dichos coeficientes produzcan soluciones imaginarias se indique y se lean otros coeficientes. No escribir sentencias fuera del bucle, ya que no es necesario.

6.38 Sea el siguiente fragmento de programa:

```

...
TYPE
  materias = (Algebra, Calculo, Fisica, Programacion);
VAR
  asignatura: materias;
  i: integer;
...
BEGIN
  ...
  asignatura := Algebra;
  FOR i := Ord(Algebra) TO Ord(Programacion) DO
    BEGIN
      Writeln(i, asignatura);
      asignatura := Succ(asignatura);
    END;
  ...

```

Si se considera que el bucle *FOR* es correcto sintácticamente, simplificarlo al máximo.
Si se considera incorrecto, escribir uno correcto que realice lo que se pretende.

6.39 Modificar el ejercicio resuelto 6.21, sustituyendo las sentencias condicionales anidadas por una estructura multialternativa *CASE*.

6.40 Escribir un programa en Pascal para contar el número de palabras de un texto, introducido por teclado y acabado en punto.

6.41 Escribir un programa que lea puntos (x, y) del plano y nos diga si están dentro, sobre o fuera de la circunferencia $x^2 + y^2 = 25$.

6.42 Escribir un programa que lea números enteros positivos, cuente el número de ellos, nos diga cuántos hay mayores que 10 y cuántos hay comprendidos entre 5 y 100. Para finalizar la entrada de datos se introducirá un número negativo.

6.43 Realizar un programa que lea una secuencia de notas (en el rango de 0 a 100) y calcule la siguiente información:

CUESTIONES Y EJERCICIOS PROPUESTOS

- a) El número de notas introducidas
- b) La nota media
- c) La nota mínima
- d) La nota máxima
- e) El número de suspensos (por debajo de 50)
- f) El número de aprobados

Nota: Utilizar un número negativo para finalizar la entrada de notas.

6.44 Escribir un programa, que dado un número entero entre 1 y 80, imprima un cuadrado formado por:

- puntos, . , en la diagonal principal
- signos + por encima de la diagonal principal
- signos * por debajo de la diagonal principal

6.45 Escribir un programa que calcule el día de la semana para una fecha dada en la forma numérica día, mes y año. Sólo será válido para el calendario Gregoriano.

6.46 Realizar un programa que por medio de tres preguntas y dos posibles respuestas discrimine 8 elementos.

6.47 Dos números primos son gemelos si difieren en dos unidades; (por ejemplo el 17 y el 19 son gemelos). Diseñar un programa que escriba todas las parejas de primos gemelos entre 1 y un tope leído por teclado.

6.48 Realizar una gráfica mediante caracteres (sin utilizar librerías de gráficos) de la función:

$$y = e^{-x} \text{ Sen } 2x$$

en un intervalo leído por teclado.

6.49 Realizar un programa que con tres preguntas y tres posibles respuestas, discrimine 27 objetos diferentes. ($3^3 = 27$).

6.50 Escribir un programa, que para un n dado, calcule el sumatorio: $\sum_{i=1}^n n^n$

6.51 Sea x un valor leído por teclado, escribir un programa que determine e^x , con una precisión de una diezmilésima, usando la fórmula:

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!} + \dots$$

6.52 Análogamente al anterior, para:

$$\text{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

6.53 Dados ocho tipos de personalidad, realizar un programa a modo de juego de salón, que usando tipos enumerados, con tres preguntas determine la personalidad del individuo.

6.54 Dadas dos fechas, la del nacimiento y la actual, realizar un programa que calcule el número de años, meses y días de una persona.

6.55 Simplificar el código del ejercicio resuelto 6.33, teniendo en cuenta la observación hecha después de la solución del ejercicio.

6.14 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

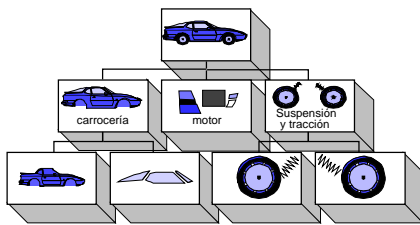
Las estructuras de control se estudian con mayor o menor detalle en todos los libros de introducción a la programación en los distintos lenguajes. En el caso del lenguaje Pascal podemos citar varios clásicos (a nivel básico, de introducción): el libro titulado *PASCAL*, de Dale/Orshalick, editado por McGraw-Hill en 1985; existe una nueva edición de esta obra, ampliada, revisada y modificada, escrita por Dale/Weems y editada también por Mc-Graw-Hill en 1990. Luis Joyanes Aguilar es autor de *Programación en Turbo Pascal, Versiones 5.5, 6.0 y 7.0* (1993); y *Fundamentos de programación: Algoritmos y estructura de datos* (1988), ambos en la editorial McGraw-Hill. El lenguaje Pascal como vehículo de la programación estructurada también es utilizado en la obra *Programación con Pascal* de M. Katrib Mora y E. Quesada Orozco, publicada por la editorial Pueblo y Educación de Cuba (1991).

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Sobre tratamiento secuencial de la información, consultar la obra de *Lucas, Peyrin y Scholl*, titulada *Algorítmica y Representación de Datos, volumen I: Secuencias, autómatas de estados finitos*, editada por *Masson* en 1985. En este libro se pueden encontrar muchos ejemplos de algoritmos para la máquina de caracteres descrita en el apartado 6.7, desde contar el número de caracteres de la cinta, hasta contar el número de apariciones de una palabra determinada. También se puede consultar la obra *Curso de programación* de *J. Castro, F. Cucker, X. Messeguer, A. Rubio, Ll. Solano, y B. Valles* (Ed. MacGraw-Hill, 1993).

La primera referencia contra el uso indiscriminado de las instrucciones de salto incondicional (GOTO), fue el artículo de *E.W. Dijkstra* titulado "*Goto statement considered harmful*" publicado en la revista *Communications of the ACM* (Vol 11, núm. 3, marzo 1968). El concepto de programación estructurada fue desarrollado en el libro titulado *Structured Programming*, de los autores *O.J. Dahl, E.W. Dijkstra, y C.A.R. Hoare*, publicado por la editorial *Academic Press* en 1972.

Existen muchos libros sobre Cálculo Numérico, pero muy pocos que desarrollen aplicaciones matemáticas en lenguaje Pascal. Recomendamos consultar *Numerical Recipes*, de *Press, Flannery, Teukolsky y Vetterling*, publicado por *Cambridge University Press* en 1988. Está disponible en lenguajes C, Fortran y Pascal. Los mismos autores con *W.T. Vetterling* de primer autor, también tienen el libro *Numerical Recipes, example book* que incluye un disquete con ejercicios resueltos en el lenguaje correspondiente. En castellano se pueden citar las obras: *Algoritmos numéricos en Pascal*, de *J. Puy*, editado por la E.T.S. de Ingenieros de Caminos de Madrid (1985); *Programación en PASCAL, con aplicaciones a Ciencias e Ingeniería* (tomo I), *Soluciones avanzadas* (tomo 2), *Guía del profesor* (tomo 3), por *Susan Finger, y Ellen Finger*, en la editorial Anaya (1989). La mayoría de los libros clásicos de Cálculo Numérico utilizan el lenguaje Fortran para implementar los algoritmos desarrollados. Citaremos entre ellos *Programación en Fortran 77 con aplicaciones de Cálculo Numérico en Ciencias e Ingeniería*, de *G. J. Borse*, editado en español por Anaya en 1989; y *Applied Numerical Analysis*, de *Curtis F. Gerald*, publicado por *Addison-Wesley* en 1978.



CAPITULO 7

SUBPROGRAMAS

CONTENIDOS

- 7.1 Introducción
- 7.2 Funciones
- 7.3 Procedimientos
- 7.4 Transformación de funciones en procedimientos
- 7.5 Anidamiento de subprogramas
- 7.6 Efectos laterales
- 7.7 Subprogramas externos e internos. Bibliotecas.
- 7.8 Declaración *forward*
- 7.9 Programación gráfica
- 7.10 Control de pantalla alfanumérica
- 7.11 Tipos procedurales de Turbo Pascal
- 7.12 *Units* de Turbo Pascal
- 7.13 Cuestiones y ejercicios resueltos
- 7.14 Ejercicios propuestos
- 7.15 Ampliaciones y notas bibliográficas

7.1 INTRODUCCION

En el capítulo segundo, se estudió el diseño descendente, que consistía en dividir un problema en distintos subproblemas (figura 7.1). En el lenguaje de programación Pascal también existe la facilidad de descomponer los distintos *subproblemas* de un programa en *subprogramas* (figura 7.2). Estos subprogramas pueden representar tareas simples las cuales se ejecutan una sola vez o bien tareas repetitivas que se ejecutan varias veces. Los subprogramas pueden contener o dividirse en otros subprogramas, y así sucesivamente hasta llegar a los *subprogramas terminales* (marcados con trama y doble rectángulo en la figura 7.2).

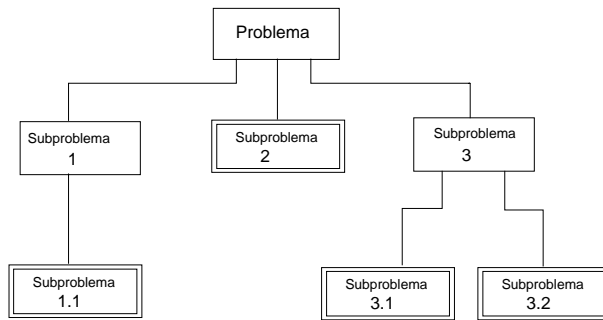


Figura 7.1 División de un problema en subproblemas

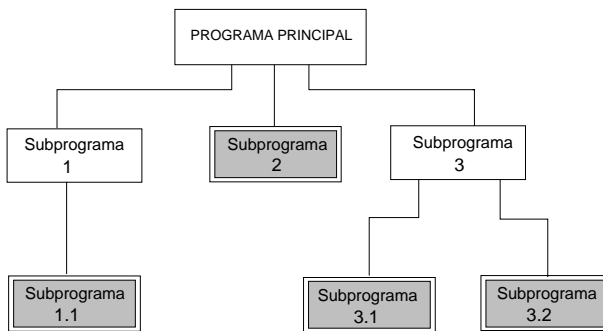


Figura 7.2 División de un programa en subprogramas

Los *subprogramas* se ejecutan según el orden impuesto por lo que se denomina *programa principal* y pueden ser llamados también desde otro programa. El programa principal es equivalente al módulo problema del diseño descendente, y los subprogramas a los subproblemas.

La programación se simplifica mucho cuando se hace una división modular. Cada módulo se puede considerar como una tarea independiente, y la atención se centra en *qué* debe de realizar el módulo y *cómo* lo hace.

VENTAJAS DE LA UTILIZACION DE SUBPROGRAMAS

- Facilitan la modularidad y estructuración de la programación.
- Facilitan la lectura, comprensión e inteligibilidad de los programas.
- En el caso de que haya que escribir un mismo grupo de instrucciones varias veces en un programa, se pueden considerar esas instrucciones como un subprograma, y llamarlas desde el programa principal cada vez que sea necesario. La ventaja para el programador consiste en que sólo debe de escribir una vez esas instrucciones, además el programa fuente y el objeto serán más cortos, lo que supone un ahorro de memoria.
- Un subprograma que realiza una operación determinada puede ser llamado por distintos subprogramas.
- Facilitan la puesta a punto y corrección de errores.
- Facilitan el mantenimiento.

TIPOS DE SUBPROGRAMAS EN PASCAL

En el lenguaje Pascal existen dos tipos de subprogramas:

- Funciones
- Procedimientos

Primeramente se estudiarán las funciones y a continuación los procedimientos.

7.2 FUNCIONES

Pueden ser de dos tipos:

- **Funciones estándar**
- **Funciones definidas por el usuario**

FUNCIONES ESTANDAR

Ya se han estudiado anteriormente, y están incorporadas al lenguaje. Se pueden utilizar en cualquier parte del programa en que se necesiten.

Ejemplo 7.1

```
Sin (x)
Cos (x)
Ln (x)
```

Estas funciones devuelven un valor *real* para un argumento x *real*.

FUNCIONES

FUNCIONES DEFINIDAS POR EL USUARIO

Son las programadas por el usuario y se pueden llamar desde el programa principal o desde otro subprograma.

Las funciones definidas por el usuario tienen que definirse y construirse antes de su llamada, por lo tanto deben de ir antes del programa principal, según el siguiente esquema:

```

    FUNCION f1
    FUNCION f2
    ...
    FUNCION fn

(* Programa Principal *)
...
Llamada a FUNCION f8
Llamada a FUNCION f1
Llamada a FUNCION f5
```

Las llamadas a las funciones **no** tienen por qué realizarse en el orden de declaración y construcción.

ESTRUCTURA DE UN SUBPROGRAMA FUNCTION

La estructura de una función es muy similar a la de un programa. Solamente se diferencia en el encabezamiento y el cálculo del valor que devuelve la función.

Cuerpo de un programa con declaración de funciones

```

PROGRAM      ...
...
FUNCTION     ...
...
BEGIN
(*Cuerpo de la función*)

END;
BEGIN      (* Programa Principal *)
...
llamada/s a la función.
...
END.      (* Programa Principal*)
```

Estructura interna de una función

```

FUNCTION Nombre (declaración de parámetros formales):tipo del resultado;
LABEL
    (* Declaración de etiquetas *)
    ...
CONST
    (* Declaración de constantes *)
    ...
TYPE
    (* Declaración de tipos *)
    ...
```

```

VAR
  (* Declaración de variables *)
  ...
  (* Declaración de otras funciones o procedimientos locales *)
  ...
BEGIN
  (*Cuerpo de la función*)
  ...
  Nombre := ...
END; (* FUNCTION *)
    
```

Explicación

Nombre: Identificador de la función, y es donde se devuelve el resultado de la función.

Declaración de parámetros formales: son las variables y sus tipos que actúan como parámetros de la función. La sintaxis de la declaración viene dada por el siguiente diagrama sintáctico:

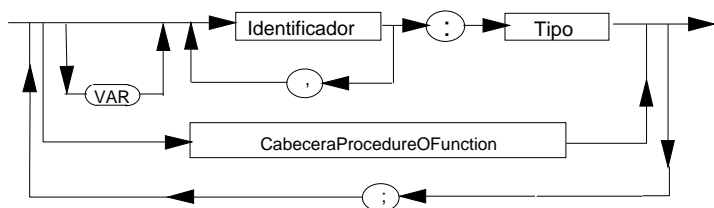


Figura 7.3 Declaración de parámetros

El diagrama sintáctico de **CabeceraProcedureOFunction** es el siguiente:

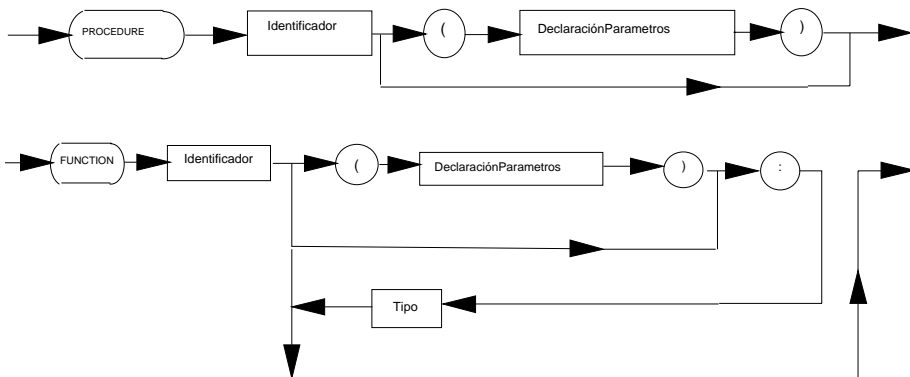


Figura 7.4 Cabecera de procedimientos y funciones

Tipo del resultado: es el tipo del resultado de la función, y es el tipo asignado al identificador *Nombre*. Sólo puede ser un tipo simple (entero, real, carácter, boolean o enumerado).

FUNCIONES

Declaración de etiquetas, constantes, tipos y variables: se realiza de la misma forma que en el programa principal, pero solo tienen validez o ámbito dentro de la función y subprogramas anidados en ella.

Declaración de subprogramas locales: serán subprogramas anidados a la función.

Cuerpo de la función: es una sentencia compuesta.

Cuando **finaliza la función** se debe **asignar el resultado al nombre de la función**.

Nunca debe aparecer el nombre de la función en la parte derecha de una sentencia de asignación dentro de la misma función, ya que el compilador lo tomaría como una llamada recursiva (como se verá más adelante al hablar de *recursividad*). Se puede lograr lo anterior con variables auxiliares.

La declaración de una función en notación EBNF es la siguiente:

```
<declaración de función> ::= <cabeceraFunction> ; <bloque> |
<identificadorFunction> ; <bloque>
<cabeceraFunction> ::= FUNCTION <identificador>
(<sección de parámetros formales>
{ ; <sección de parámetros formales> })
: <tipo de resultado>
<identificadorFunction> ::= FUNCTION <identificador>
<sección de parámetros formales> ::= <grupo de parámetros> |
VAR <grupo de parámetros> |
<cabeceraFunction> |
<cabeceraProcedure>
<grupo de parámetros> ::= <identificador> { , <identificador> } :
<identificador de tipo>
```

Ejemplo 7.2

El lenguaje Pascal no tiene la función trigonométrica tangente como estándar, pero se puede construir una función tangente de la siguiente forma:

```
FUNCTION Tg (z : real) : real;
BEGIN
  Tg := Sin (z) / Cos (z);
END;
```

Explicación

Tg es el nombre de la función, y el valor devuelto es de tipo real.

z es el parámetro, y es de tipo real (z:real).

El *cuerpo de la función* está constituido en este caso por una única sentencia, que coincide con la asignación del resultado de la función al nombre.

LLAMADA A FUNCIONES

Se realiza de forma similar a como se utilizan las funciones estándar:

```
Nombre (lista de argumentos)
```

Su diagrama sintáctico es el siguiente:

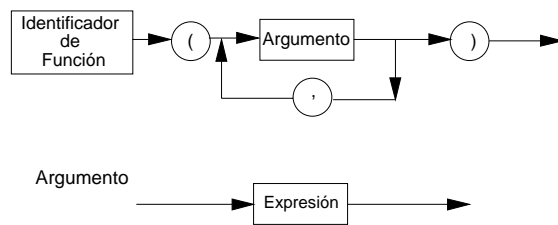


Figura 7.5 Llamada a una función

donde el argumento es una expresión.

A los argumentos también se les llama *parámetros actuales*, y a los parámetros de la función *parámetros ficticios* o *parámetros formales*.

Donde pueden aparecer las llamadas a funciones

- Formando parte de una expresión. En este caso se ejecutaría la función que devolvería un resultado, y después se evaluaría la expresión.
- Como parte de una sentencia de escritura.
- Como argumento de llamada a otra función o procedimiento.

Donde no pueden aparecer las llamadas a funciones

- Como una sentencia de la forma:

```

...
Nombre (argumentos);
...

```

- A la izquierda de una *sentencia de asignación*:

```
Nombre(argumentos) := expresión;
```

Ejemplo 7.3

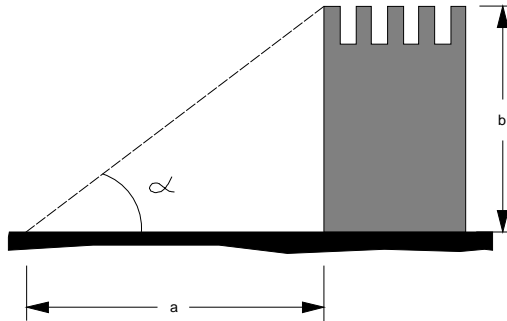
Realizar un programa que calcule la altura de una torre a partir de la distancia a y el ángulo α , según el esquema y fórmulas siguientes:

Análisis

En Pascal y en la mayoría de los lenguajes, los argumentos de las funciones trigonométricas deben ir expresados en radianes.

$$\operatorname{tg} \alpha = \frac{b}{a} \quad \rightarrow \quad b = a \operatorname{tg} \alpha$$

FUNCIONES



Algoritmo:

```
INICIO
  NIVEL 0
  Leer distancia a y ángulo alfa
  Pasar de grados a radianes
  Calcular tangente
  NIVEL 1
  Tangente = Seno(alfa) / Coseno(alfa)
FIN
```

Codificación en Pascal:

```
PROGRAM Torre (input, output);
CONST pi = 3.14159;

VAR
  b, a, alfa : real;

FUNCTION Tg (z : real): real;
BEGIN (* Tg *)
  Tg := Sin (z) / Cos (z);
END; (* Tg *)

BEGIN (*Programa principal*)
  Write ('Dame la distancia y el ángulo en grados sexagesimales');
  Readln (a, alfa);
  alfa := alfa * pi / 180;
  b := a * tg(alfa);
  Writeln('La altura es ', b:7:2);
END.
```

Primero se evalúa $Tg(alfa)$ y luego la expresión $a * Tg(alfa)$.

Ejemplo 7.4

Escribir una función booleana tal que dada una fecha (día, mes, año) de tipo entero, nos indique si la fecha es correcta comprobando:

SUBPROGRAMAS

- que los tres números sean positivos
- que el día esté entre 1 y 31
- que el mes esté entre 1 y 12

Realizar la llamada de la función desde un programa.

```
PROGRAM Fechas (input, output);
VAR
    dia, mes, any: integer;
    a : boolean;

{-----}

FUNCTION Correcta(d, m, a:integer):boolean;
BEGIN (* Correcta *)
Correcta := (a > 0) AND (d >= 1) AND (d <= 31) AND
            (m >= 1) AND (m <= 12)
END; (* Correcta *)

{***** Programa principal *****}

BEGIN
REPEAT
Writeln ('Introducir dia,mes, y año');
Readln (dia, mes, any);
a:= Correcta (dia, mes, any);
IF NOT a
    THEN Writeln ('Fecha incorrecta');
UNTIL a;
END.
```

Ejemplo 7.5

Escribir una función que calcule el espacio recorrido por un objeto en movimiento.

Realizar la llamada de la función desde un programa.

```
PROGRAM Distancia (input, output);
VAR
    t, e:integer;

{-----}

FUNCTION CalculoDistancia (t:integer);
VAR
    v:integer;
BEGIN
Writeln ('¿ Velocidad en metros/segundos ? ');
Readln(v);
CalculoDistancia := v * t;
END;

{***** Programa principal *****}

BEGIN
Writeln ('¿ Tiempo en segundos ? ');
Readln(t);
e := CalculoDistancia(t);
Writeln;
Writeln (' Espacio recorrido en ', t:5, ' segundos es ',e:8);
Writeln;
```

FUNCIONES

```
Writeln (' Pulse una tecla para continuar ');
Readln;
END.
```

DECLARACIONES LOCALES

Las variables, constantes y tipos que se declaran *dentro de una función o procedimiento*, sólo pueden utilizarse dentro de dicha función o procedimiento, y se les denomina **locales**. Su valor se pierde después de cada llamada al subprograma.

Ejemplo: En el programa `Distancia` es local `v` ya que se declara dentro de la función `CalculoDistancia`.

DECLARACIONES GLOBALES

Las variables, constantes y tipos declarados en un programa, pueden utilizarse en cualquier parte del programa, tanto dentro de los subprogramas internos al programa como fuera de ellos. A estas declaraciones se les denomina **globales**.

Ejemplo: En el programa `Fechas` son globales `día`, `mes`, `any`, `a`.

AMPLITUD DE UNA VARIABLE O AMBITO

Es importante conocer en cualquier parte del programa, si una variable se puede utilizar o no. Se denomina ámbito de una variable a la parte de programa en que se puede utilizar, es decir, el compilador la reconoce.

Dentro de las funciones y de los subprogramas anidados a ellas, si existen identificadores con igual nombre y distintos ámbitos, tendrá validez el identificador local.

Ejemplo: En el programa `Fechas`, dentro de la función `Correcta`, tiene validez el identificador `a` de tipo *integer* local a la función y no `a` de tipo *boolean*.

RECURSIVIDAD

El lenguaje Pascal permite utilizar *la recursividad* que es la propiedad que posee un lenguaje de poder llamarse desde el interior de un subprograma al propio subprograma.

Una función o procedimiento se denomina **recursivo** si se invoca o llama a sí mismo.

Las funciones recursivas son muy útiles fundamentalmente cuando un problema se puede definir en función: a) de sus propios términos, es decir, recursivamente, y b) de uno o varios casos triviales.

Así si se desea conocer el factorial de un número `n` que se expresa matemáticamente como

$$n! = n * (n - 1) * (n - 2) * \dots * 2 * 1$$

se puede obtener a partir de una función recursiva de la siguiente manera:

```

factorial (1) = 1      (caso trivial)
factorial (2) = 2 * 1 = 2 * factorial (1)
factorial (3) = 3 * 2 * 1 = 3 * factorial (2)
.
.
.
factorial (n) = n! = n * factorial (n - 1)

```

Ejemplo 7.6

Escribir un programa que calcule las combinaciones de n elementos tomados de m en m.

Análisis. Se utiliza la fórmula:

$$\binom{n}{m} = \frac{n!}{m! \times (n-m)!}$$

donde el factorial se calcula mediante una función recursiva.

Algoritmo

```

INICIO
  NIVEL 0
    Leer los valores de n y m
    Calcular Factorial(n) / (Factorial(m) * Factorial(n-m))
  NIVEL 1
    Factorial
      INICIO
        SI x = 0
          ENTONCES
            Factorial = 1
        SI_NO
          SI x > 0
            ENTONCES
              Factorial = x * Factorial(x-1)
            SI_NO
              Error
          FIN_SI
        FIN_SI
      FIN
    FIN
  FIN

```

Codificación en Pascal

```

PROGRAM Combinaciones (input,output);
VAR
  n, m: integer;
  c : real;
{-----}

```


PROCEDIMIENTOS

```
FUNCTION Factorial (x : integer) : real;
(* El resultado se coloca de tipo real para que pueda tomar
valores que superen a maxint. *)
BEGIN
  IF x=0
  THEN Factorial:=1
  ELSE IF x > 0
  THEN Factorial:= x * Factorial(x-1)
  ELSE Writeln('ERROR:factorial de un número negativo')
  END;
  (*****
BEGIN (* Programa Principal *)
  Write('Deme n y m: ');
  Readln(n,m);
  c:= Factorial(n) / (Factorial(m) * Factorial(n-m));
  Writeln('c =',c:10:0);
  END.
```

7.3 PROCEDIMIENTOS

Anteriormente se han estudiado los subprogramas llamados funciones (*FUNCTION*), que tomando unos valores como parámetros **devuelven un valor** como resultado, asignado al nombre de la función. La descripción de los subprogramas función corresponde a la implementación del concepto matemático de función.

A menudo en programación nos encontraremos con el problema de escribir subprogramas que no tengan las restricciones que tienen los subprogramas función, es decir, deseamos obtener **varios resultados**, o **ningún resultado**. Esto se puede hacer mediante los subprogramas llamados procedimientos (*PROCEDURE*).

Las funciones también pueden obtener varios resultados, utilizando sus parámetros para devolver otros resultados. Es lo que se llama comunicación argumento-parámetro *por dirección*.

Aunque las funciones pueden realizar esta comunicación por dirección, sólo se suele utilizar con el fin de ahorrar memoria. Debe mantenerse la filosofía de que la función devuelve un único resultado.

Los procedimientos pueden ser *procedimientos estándar*, y *procedimientos definidos por el programador*. Los procedimientos estándar son los que están definidos por el propio lenguaje Pascal, por ejemplo los procedimientos de entrada y salida *Read* y *Write*. Los procedimientos definidos por el programador se explican en los apartados siguientes.

DEFINICION DE PROCEDIMIENTOS

La estructura de los procedimientos es similar a la de las funciones variando las sintaxis de la declaración, y de la llamada.

Cuerpo de un programa con procedimientos

```

PROGRAM      ...
...
PROCEDURE   ...
...
BEGIN
(*Cuerpo del procedimiento*)
END;
BEGIN      (* Programa Principal *)
...
  llamada/s al procedimiento
...
END.       (* Programa Principal*)

```

Estructura interna de un procedimiento

```

PROCEDURE Nombre (Declaración de parámetros formales);
LABEL
  (* Declaración de etiquetas *)
  ...
CONST
  (* Declaración de constantes *)
  ...
TYPE
  (* Declaración de tipos *)
  ...
VAR
  (* Declaración de variables *)
  ...
  (* Declaración de otras funciones o procedimientos locales *)
  ...
BEGIN
  (*Cuerpo del procedimiento*)
  ...
END;  (* PROCEDURE *)

```

Explicación

Nombre: es un identificador que representa al procedimiento, se utilizará para llamar al procedimiento.

Declaración de parámetros formales: su sintaxis se muestra en el diagrama sintáctico de la figura 7.3.

Si se utiliza VAR la comunicación es por **dirección** (usada para salida o entrada/salida) y si no se indica VAR es una comunicación por **valor** (utilizada para entrada), como se explicará posteriormente.

Declaración de etiquetas, constantes, tipos y variables: se realiza de la misma forma que en el programa principal, pero sólo tienen validez o ámbito dentro del procedimiento en que se declaran y en los subprogramas anidados a él.

Declaración de subprogramas: funciones y/o procedimientos locales.

Cuerpo del procedimiento: es una sentencia compuesta.

PROCEDIMIENTOS

RESUMEN

Los procedimientos se definen con una cabecera y un bloque.

La **cabecera** contiene la lista de parámetros (también llamados parámetros formales del procedimiento).

El **bloque** del procedimiento se compone de una parte de **declaraciones** y de otra zona que es el **cuerpo** del procedimiento.

La comunicación en un procedimiento se hace tanto por *dirección* como por *valor*, (valor = entrada, dirección = entrada y/o salida).

Los procedimientos deben de aparecer físicamente en el programa antes de que se les llame desde el cuerpo principal o desde otro procedimiento.

La declaración de un procedimiento en notación EBNF es la siguiente:

```
<declaración de procedimiento> ::= <cabeceraProcedure> ; <bloque> |  
                                <identificadorProcedure> ; <bloque>  
<cabeceraProcedure>           ::= PROCEDURE <identificador>  
                                {<sección de parámetros formales>  
                                {;<sección de parámetros formales> } }  
<identificadorProcedure>      ::= PROCEDURE <identificador>
```

LLAMADAS A PROCEDIMIENTOS

Se realizan de la siguiente forma:

```
Nombre (lista de argumentos)
```

Las llamadas a procedimientos pueden aparecer:

- Como una sentencia (su utilización es similar a una sentencia de lectura o escritura).

```
...  
Nombre (lista de argumentos);  
...
```

- Como argumento de llamada a otra función o procedimiento.

```
...  
OtroProcedim (... ,Nombre_Proc_o_Func, ... )  
...
```

Estas llamadas **no** pueden aparecer:

- Ni a la izquierda, ni a la derecha de una sentencia de asignación.
- Como parte de una sentencia de escritura o de lectura:

```
Write(...,Nombre(lista de argumentos),...)
```

Ejemplo 7.7

Realizar un programa que escriba 2 triángulos de asteriscos.

SUBPROGRAMAS

```
PROGRAM EscribeTriangulos (output);
VAR
  i:integer;

{-----}

PROCEDURE ImprimeTriangulo;
BEGIN
  Writeln (' * ');
  Writeln (' *** ');
  Writeln ('*****');
END;

{***** Programa principal *****}

BEGIN
  FOR i:= 1 TO 2 DO
    BEGIN
      ImprimeTriangulo; (* llamada al procedimiento *)
      Writeln;
    END;
  END.
```

El resultado de la ejecución del programa anterior es:

```
*
***
*****

*
***
*****
```

COMUNICACION ARGUMENTO-PARAMETRO

Cada vez que se hace una llamada a una función o procedimiento, se establece una comunicación entre cada argumento y su parámetro. Existen dos métodos para realizar esta comunicación:

- Comunicación **por valor** (o paso por valor)
- Comunicación **por dirección**, (o paso por VAR o referencia)

Comunicación por valor

La llamada por valor asigna el valor del argumento a su parámetro correspondiente en la cabecera del subprograma.

El parámetro es una variable independiente de nueva creación, con su propia situación en memoria, que recibe el valor del argumento al comienzo de la ejecución del subprograma. Puesto que parámetro y argumento son variables independientes, cualquier cambio en el parámetro que ocurra durante la ejecución del subprograma no tienen ningún efecto en el valor del argumento original.

Esto significa que cuando se emplea la llamada por valor, es imposible la devolución de valores al punto de llamada por medio del parámetro. La llamada por valor, es, por tanto, más útil

PROCEDIMIENTOS

cuando es necesaria una *transmisión del valor inicial del argumento al parámetro sin que pueda ocurrir después ninguna otra comunicación entre ambos*. Al terminar la ejecución del subprograma, la variable parámetro se destruye.

En este caso *los argumentos pueden ser constantes, variables o expresiones de cualquier tipo*.

A los parámetros sobre los que se realiza una comunicación por valor se les puede considerar como *parámetros de entrada*, ya que se les da valor cuando se hace la llamada, pero no dan valor a los argumentos cuando finaliza la ejecución de la función.

El argumento y el parámetro deben de ser del mismo tipo, sólo se permite la excepción de un argumento entero y un parámetro real.

En el programa `FECHAS` del ejemplo 7.4 la comunicación entre `dia`, `mes`, `any` y `d`, `m`, `a` es por valor.

Comunicación por dirección

A diferencia de la comunicación por valor, *no se crea una posición aparte de memoria para el parámetro*, sino que se realiza el paso de la dirección de la posición real de memoria donde se almacena el valor del argumento. El parámetro es simplemente otro nombre para la misma dirección ya creada para el valor del argumento.

Este método de asociación implica que cada vez que se encuentra el nombre del parámetro en el subprograma, su dirección real de referencia es precisamente la del argumento, lo que tiene como efecto *el paso de los valores en ambas direcciones entre argumento y parámetro*. Es decir, el parámetro recibe el valor del argumento actual en el momento de llamada, de modo que cualquier asignación subsiguiente de nuevos valores al parámetro durante la ejecución del subprograma hace que la asignación tenga lugar también en el argumento.

La comunicación por dirección es más útil para parámetros donde resulta necesario la comunicación del valor en ambas direcciones.

En este caso los argumentos deben de ser *variables de cualquier tipo* (no pueden ser ni constantes ni expresiones).

A los parámetros sobre los que se efectúa una comunicación por dirección se les considera como de salida o entrada-salida.

Para indicar que se desea realizar una comunicación por dirección se debe de poner `VAR` antes de declarar el parámetro y su correspondiente tipo.

Siempre que un parámetro lleve delante `VAR` la comunicación se hace por dirección.

Ejemplo 7.8

El siguiente procedimiento sirve para intercambiar el contenido de dos variables que se le pasan como parámetros.

```
PROCEDURE Intercambia (VAR x,y: integer);
VAR
  temp: integer;
BEGIN
  temp:=x;
  x:=y;
  y:=temp;
END;
```

si se supone que *a* y *b* son dos variables globales de tipo *integer*, tras la ejecución del siguiente segmento de programa

```
a:=1;
b:=2;
Intercambia (a,b);
Writeln(a,b);
```

se observa que los contenidos de *a* y *b* se han intercambiado realmente, y la sentencia *Writeln* escribirá

2 1

como era de esperar. Al utilizar la comunicación por dirección (por *VAR*) no se crea ninguna nueva posición en memoria para los parámetros *x* e *y*, sino que éstos actúan directamente sobre las variables *a* y *b* respectivamente.

Sin embargo si se suprime la palabra *VAR* de la cabecera del procedimiento y volvemos a ejecutar el programa, veremos que no se produce el intercambio de las variables *a* y *b*, y la sentencia *Writeln* escribirá

1 2

Ello es así porque ahora estamos utilizando la comunicación por valor, con lo cual al llamar al procedimiento *Intercambia* se crean dos nuevas variables locales *x* e *y* sobre las cuales se *copian* respectivamente los contenidos de *a* y *b*. El procedimiento intercambiará los contenidos de *x* e *y*, que solo son una copia de *a* y *b*, pero éstas quedarán inalteradas.

Cuando se hace la comunicación por dirección no se crea memoria, sino que se ocupa la memoria ya asignada. Luego este método también puede servir para ahorrar memoria, sobre todo cuando se trata de estructuras de datos con gran ocupación de memoria (por ejemplo arrays como se verá en el capítulo octavo).

Aunque en las funciones se puede realizar la comunicación por dirección, sólo se suele utilizar con el fin de ahorrar memoria, o en casos especiales como la función del ejemplo 7.9 para generar números aleatorios.

PROCEDIMIENTOS

La filosofía del uso de funciones es que devuelvan un sólo resultado, asignado al nombre de la función.

Ejemplo 7.9

Se construye un programa que genera números aleatorios¹⁴. En muchos algoritmos es necesario el uso de números aleatorios. Sin embargo los valores generados por los ordenadores son números pseudoaleatorios, pues se aplican fórmulas o algoritmos para calcularlos. Un método de cálculo es el denominado de los **restos potenciales**, que genera números pseudoaleatorios con distribución uniforme y según la fórmula:

$$R_{i+1} = k(R_i \text{ MOD } p)$$

Dado que los ordenadores digitales efectúan con mayor rapidez las divisiones por números múltiplos de 2, ya que se reducen los bits (dígitos binarios), se utiliza como p una potencia de 2. La potencia acostumbra a ser el número de bits de la palabra menos 1. En cuanto a k es recomendable que sea igual o aproximadamente igual a $2^{b/2}$ donde b es igual al número de dígitos binarios de la palabra del ordenador. Como número inicial de la serie debe tomarse un número impar (R_0 impar). De todo lo anterior se deduce que se genera la misma secuencia de números para una misma semilla u origen (R_0).

Función que genera números pseudoaleatorios uniformemente distribuidos entre 0 y maxint

```
FUNCTION Aleatorio (VAR i:integer):integer;
CONST k =259;
BEGIN
  i:= k*i;
  IF i<0 THEN i:= i + maxint + 1;
  Aleatorio :=i;      (* p = 1 *)
END;
```

Función que genera números pseudoaleatorios uniformemente distribuidos entre 0 y 1

```
FUNCTION Rand (VAR i: integer) : real;
CONST k = 259;
BEGIN
  i:= k*i;
  IF i<0 THEN i:= i+maxint+1;
  Rand :=i/maxint;  (* p = maxint *)
END;
```

Ejemplo 7.10

Escribir un procedimiento que devuelva la suma de dos números enteros, y además en una variable el valor *true* si los dos son positivos y *false* en caso contrario.

14 El compilador Turbo Pascal incorpora: la función **Random** (que devuelve un número pseudoaleatorio entre 0 y 1, también se puede asignar otro rango); el procedimiento **Radomize** (que inicializa el generador de números pseudoaleatorios con un valor aleatorio tomado del reloj del sistema); y la variable **RandSeed** (que contiene la semilla del generador de números pseudoaleatorios).

SUBPROGRAMAS

```
PROCEDURE SumaPosBool(x,y,:integer; VAR z:integer;VAR a:boolean);
BEGIN
  z:= x+y;
  a:=(x >= 0) AND (y >= 0);
END;
```

Una llamada a este procedimiento podría ser:

```
SumaPosBool (x1, x2, x3, x4);
```

donde:

- x1 y x2 son constantes, variables o expresiones enteras.
- x3 es una variable entera.
- x4 es una variable booleana.

Cuando se efectúa la llamada:

- Se crea memoria para las variables x e y , tomándola de la pila interna (*stack*).
- Los valores de x_1 y x_2 pasan a la zona de memoria reservada a x e y . Es decir se asignan a las variables x e y .
- Los valores de x_3 y x_4 pasan a z y a respectivamente, sin embargo no se crea memoria para z y a , sino que utilizarán las mismas posiciones de memoria usadas en el programa que hace la llamada para x_3 y x_4 .

Cuando finaliza el procedimiento:

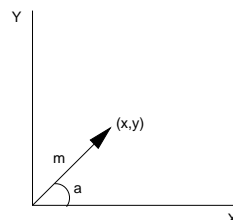
- Se libera la memoria asignada a los parámetros sobre los que existe una comunicación por **valor** (x e y). Los valores de x e y **no pasan** a x_1 y x_2 .
- Los valores de z y a **pasan** a x_3 y x_4 .

Este paso es evidente ya que los argumentos ocupan la misma posición de memoria que los parámetros en una **comunicación por dirección**, por lo que toda modificación del valor del parámetro hecha en el procedimiento repercute en el valor del argumento.

Ejemplo 7.11

Escribir un procedimiento al que se le den las dos coordenadas de un punto y devuelva el módulo y el argumento de dicho punto.

```
PROCEDURE Polar (x, y: real;
                 VAR m, a: real);
BEGIN
  m:= Sqrt (Sqr(x) + Sqr(y));
  a:= ArcTan (y/x); (* "a" está en radianes *)
END;
```



7.4 TRANSFORMACION DE FUNCIONES EN PROCEDIMIENTOS

Todas las funciones se pueden programar fácilmente como procedimientos, solamente es necesario utilizar un parámetro de salida en vez del nombre de la función. Los procedimientos pueden ampliar la información devuelta por las funciones.

Ejemplo 7.12

Escribir una función que determine si una ecuación de segundo grado posee soluciones complejas .

Análisis

Sea la ecuación: $ax^2 + bx + c = 0$

Discusión:

- Si $b^2 - 4ac > 0$ Dos soluciones reales.
- Si $b^2 - 4ac = 0$ Una solución real, raíz doble.
- Si $b^2 - 4ac < 0$ Dos soluciones complejas conjugadas.

Se ha supuesto que el coeficiente a nunca va a ser nulo, pues en ese caso estaríamos ante una ecuación de primer grado.

Se utilizarán variables booleanas para indicar si estamos en el caso de dos soluciones reales, una solución real o dos soluciones imaginarias.

Algoritmo

```

INICIO
d = b2 - 4ac
SI d < 0
  ENTONCES
    Calcular parte real e imaginaria
  SI_NO
    SI d = 0
      ENTONCES
        Calcular solución real
    SI_NO
      Calcular dos soluciones reales
  FIN_SI
FIN_SI
FIN
    
```

Codificación en Pascal

```

FUNCTION Ecu2Grado (a,b,c:real;VAR x1, x2, xi:real):boolean;
VAR
  discri:real;
  una:boolean; (* será verdadera si posee una solución real *)
    
```

SUBPROGRAMAS

```
BEGIN
discri:= Sqr(b)-4*a*c;
IF discri<0
THEN
BEGIN
x1:= -b/(2 * a);          (* Parte real *)
xi:= Sqrt(4*a*c-Sqr(b))/(2*a); (* Parte imaginaria *)
Ecua2Grado:= true;
una:= false;
END
ELSE IF discri = 0
THEN
BEGIN
x1:= -b/(2*a);
una:= true;
Ecua2Grado:= false;
END
ELSE
BEGIN
x1:= (-b + Sqrt (discri))/(2*a);
x2:= (-b - Sqrt (discri))/(2*a);
una:= false;
Ecua2Grado:= false;
END;
END; (* Ecua2Grado *)
```

El procedimiento equivalente a la función anterior necesita un parámetro de salida. Sea *ima* dicho parámetro, el código del procedimiento queda de la siguiente manera:

```
PROCEDURE Ecua2Grado(a,b,c:real;VAR ima:boolean;VAR x1, x2, xi:real);
VAR
discri:real;
una:boolean;

BEGIN
discri:= Sqr(b)-4*a*c;
IF discri<0
THEN
BEGIN
x1:= -b/(2 * a);          (* Parte real *)
xi:= Sqrt(4*a*c-Sqr(b))/(2*a); (* Parte imaginaria *)
ima:= true;
una:= false;
END
ELSE IF discri = 0
THEN
BEGIN
x1:= -b/(2*a);
una:= true;
ima:= false;
END
ELSE
BEGIN
x1:= (-b + Sqrt (discri))/(2*a);
x2:= (-b - Sqrt (discri))/(2*a);
una:= false;
ima:= false;
END;
END; (* Ecua2Grado *)
```

ANIDAMIENTO DE SUBPROGRAMAS

VARIABLES LOCALES Y GLOBALES

Si se tienen varios procedimientos, es posible declarar variables en cada procedimiento además de hacerlo en el programa principal.

Se llaman *variables y constantes globales* a las que están declaradas inmediatamente después de la cabecera *PROGRAM* y antes de cualquier declaración de procedimiento.

Se llaman *variables y constantes locales* a las declaradas inmediatamente después de la cabecera *PROCEDURE*, o *FUNCTION*.

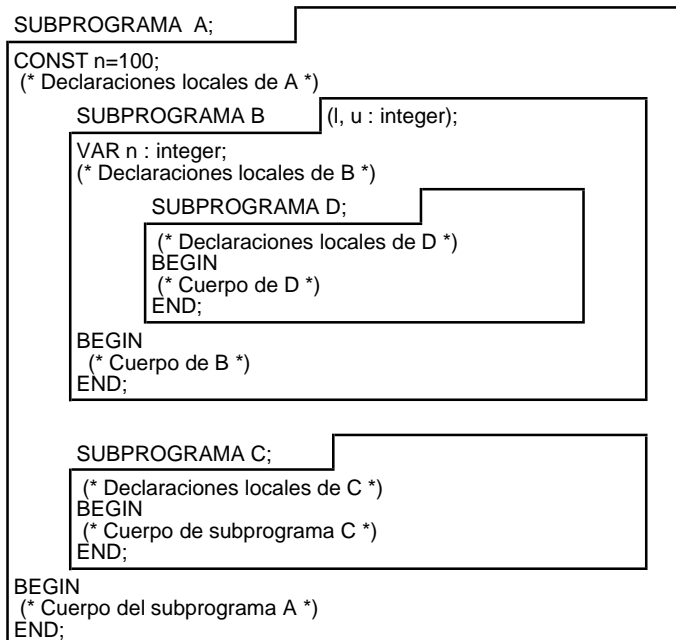
7.5 ANIDAMIENTO DE SUBPROGRAMAS

Un *subprograma* (procedimiento o función) puede contener en su interior otros subprogramas, diciéndose de estos últimos que están anidados, o que son locales a aquel.

El *nivel de anidamiento* de un subprograma es el número de subprogramas en los que está contenido. Así el programa principal tiene nivel de anidamiento 1, si dentro del anterior se define un subprograma, éste último tendrá nivel de anidamiento 2 y así sucesivamente.

DECLARACIONES LOCALES Y GLOBALES

Sea el siguiente esquema de subprogramas:



En el esquema anterior los subprogramas están anidados según la siguiente figura:

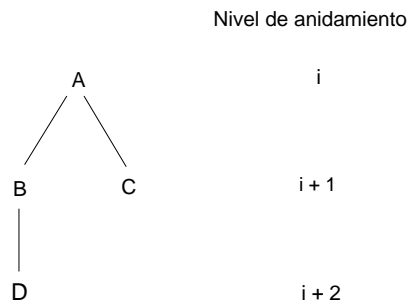


Figura 7.6 Anidamiento de subprogramas

Puesto que las declaraciones realizadas en un subprograma sólo tienen validez dentro de dicho subprograma, las declaraciones del *subprograma A*, hacen que se creen posiciones de memoria para las variables y constantes, cuando se llama al subprograma *A*. Estas variables y constantes son *locales* al *subprograma A* y a todos los *subprogramas anidados* en *A*, en el ejemplo son los subprogramas *B*, *C* y *D*.

La única **excepción** es que se declare en un subprograma anidado, otro identificador con el mismo nombre. Véase el esquema anterior.

```

SUBPROGRAMA B (l, u:integer);
...
VAR n:integer;
...

```

Entonces *n* para los *subprogramas B* y *D* será la variable declarada local en *B* y no la constante *n* global declarada en *A*.

Todo lo mostrado anteriormente se puede resumir en las tres reglas del ámbito de las declaraciones.

Si se denomina **bloque** a cada una de las cajas que se han pintado en el ejemplo anterior, se pueden enunciar las tres reglas siguientes:

- ⌘ No se puede declarar una constante o una variable, más de una vez en el mismo bloque.
- ⌘ Si se usa un identificador *I* en una sentencia *S*, búsquese el menor bloque que encierre a la vez a *S* y a una declaración de *I*, la aparición de *I* en *S* se refiere a esta declaración.
- ⌘ Un identificador sólo se puede usar dentro del bloque en el que se ha declarado.

También se pueden enunciar las reglas anteriores de la siguiente forma:

ANIDAMIENTO DE SUBPROGRAMAS

- Una variable o constante v que puede ser referenciada en un subprograma P , también puede referenciarse en un subprograma Q anidado en P , a menos que exista en Q una declaración para v .
- Las variables o constantes declaradas en un nivel de anidamiento i de un subprograma no son accesibles a otros subprogramas anidados con el mismo nivel de anidamiento i .

En el ejemplo las declaraciones de B no son accesibles desde C .

AMBITO DE LLAMADA A SUBPROGRAMAS

En un subprograma P se puede llamar a un subprograma Q si se cumple una de las reglas siguientes:

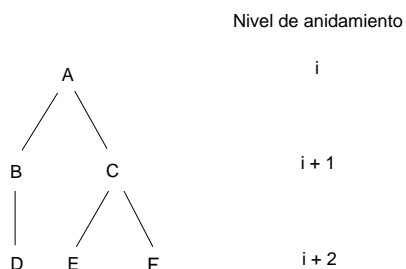
- 1ª) Q está anidado en P , es decir depende directamente de P o subprogramas anidados en P .
- 2ª) P está anidado en algún subprograma R , y Q y R están declarados en el mismo subprograma, y la declaración de R es posterior a la de Q .
- 3ª) P y Q están ambos declarados en R , suponiendo que la definición de P siga a la de Q .
- 4ª) Q puede coincidir con P . Es decir un procedimiento o función puede llamarse a sí mismo. Llamada recursiva.
- 5ª) P está anidado dentro de Q a cualquier nivel.

Ejemplo 7.13

```
PROGRAM A;  
  PROCEDURE B;  
    PROCEDURE D;  
      BEGIN  
        (* Cuerpo de D *)  
      END;  
    BEGIN  
      (* Cuerpo de B *)  
    END;  
  PROCEDURE C;  
    PROCEDURE E;  
      BEGIN  
        (* Cuerpo de E *)  
      END;  
    PROCEDURE F;  
      BEGIN  
        (* Cuerpo de F *)  
      END;  
    BEGIN  
      (* Cuerpo de C *)  
    END;  
  BEGIN  
    (* Cuerpo de A *)  
  END.
```

SUBPROGRAMAS

La estructura del programa anterior es la siguiente:



De esta forma, se puede determinar que procedimientos pueden ser llamados y cuales no según el siguiente esquema. Los números indican la regla aplicada.

Procedimiento	Puede ser llamado por				
	1 ^a	2 ^a	3 ^a	4 ^a	5 ^a
B	A	E F	C	B	D
C	A			C	E F
D	B			D	
E	C		F	E	
F	C			F	

Variables declaradas en	Pueden referenciarse en
A	A B C D E F
B	B D
C	C E F
D	D
E	E
F	F

7.6 EFECTOS LATERALES

Tal como hemos estudiado, es posible acceder y alterar el valor de las variables globales desde un subprograma.

Un procedimiento o función que altere el valor de una variable no local, se dice que tiene **efectos laterales**. Los efectos laterales van en contra de los principios de la *programación estructurada*.

EFFECTOS LATERALES

Ejemplo 7.14

```
PROGRAM EfectoLateral1 (input, output);
VAR
  a, b, c : real;
{-----}

PROCEDURE Inicializa (VAR m, n:real);
BEGIN
  a:=0; m:=0; n:=0
END;

{***** Programa principal *****)}

BEGIN
  Read (a,b,c);
  Inicializa (b,c);
  Writeln (a,b,c);
END.
```

El resultado de la ejecución de este programa son *tres ceros*. Puede ser lo que se pretendía, pero es *mal hábito* de programación, pues cuando se ha declarado una variable como parámetro, se supone que su valor puede cambiar. Sin embargo, cuando la variable pertenece a otro subprograma o al programa principal y no figura como parámetro, no se suele esperar que cambie, y puede ser causa de error.

Por ejemplo si se utilizase `Inicializa` otra vez en el mismo programa debido a cuestiones de mantenimiento, es muy posible que quien lo use olvide que ese procedimiento también afecta a la variable `a`.

INCONVENIENTES DE LOS EFECTOS LATERALES

- Dificultan la comprensión, corrección y mantenimiento de los programas, pues las variables no locales se modifican en subprogramas.
- Si se usan variables no locales dentro de subprogramas, y éstos se llaman desde distintas partes de un programa, puede ocurrir que las variables no locales usadas, no tengan el mismo identificador.
- Pueden darse situaciones extrañas, tal como la mostrada en el ejemplo 7.15, donde se altera aparentemente la propiedad commutativa de la suma.

Ejemplo 7.15

```
PROGRAM EfectoLateral2 (input, output);
VAR
  z, x : integer;
{-----}
```

SUBPROGRAMAS

```
FUNCTION Loca(y:integer):integer;
BEGIN
  x:=x +2;          (* x es una variable global *)
  Loca:=y + Sqr(y);
END;  (* Loca *)

{***** Programa principal *****)

BEGIN
  x:=7; z:=Loca(2) + x;
  Writeln (z);
END.
```

El resultado de la ejecución del programa anterior es 15

Con la sentencia de asignación

$z := \text{Loca}(2) + x$ el resultado es $6 + 9 = 15$

Pero si se cambia la sentencia de asignación de z por la siguiente:

$z := x + \text{Loca}(2)$ el resultado es $7 + 6 = 13$

Es decir que el valor de z depende del orden en que se evalúa la suma, ya que cada vez que se llama a la función se altera el valor de la variable global x.

- Si se utilizan procedimientos se pueden dar casos de indeterminación.

Ejemplo 7.16

```
PROGRAM EfectoLateral3 (input, output);
VAR
  x, y: integer;

{-----}

PROCEDURE Camelo (VAR a,b:integer);
BEGIN
  a:=8;
  x:=a + 2;
  b:= Sqr(a);
END; (* Camelo *)

{***** Programa principal *****)

BEGIN
  Camelo(x,y);
END.
```

Resulta difícil averiguar el valor que tiene la x cuando se devuelve el control al programa principal, sin observar el subprograma.

a vale 8
x vale 10

- Efecto lateral: **Alias**. Se produce cuando se pasa dos o más veces en una llamada el mismo argumento, pudiendo dar lugar a indeterminaciones.

SUBPROGRAMAS INTERNOS Y EXTERNOS. BIBLIOTECAS

Ejemplo 7.17

```
PROGRAM      EfectoAlias (input, output);
VAR
  b : integer;
{-----}

PROCEDURE   Indeter (VAR c, d : integer);
BEGIN
  c:=6;
  d:=7;
END; (* Indeter *)

{***** Programa principal *****}

BEGIN
  Indeter (b,b);
END.
```

El procedimiento devuelve los valores 6 y 7 pero la variable *b* vale 7.

METODOS PARA EVITAR LOS EFECTOS LATERALES

- ✘ Pasar todos los argumentos por valor o por dirección, y no utilizar variables globales.
- ✘ Aunque es lícito utilizar identificadores que tienen nombres idénticos a los identificadores no locales, se debe de evitar, para no crear situaciones de confusión.
- ✘ Los subprogramas deben de trabajar como cajas negras, que sólo sabemos lo que entra y lo que sale, sin tener que ocuparnos de lo que pasa dentro.
- ✘ Se pueden pasar argumentos iguales, pero asignados a variables diferentes pueden evitarnos problemas de indeterminación.

7.7 SUBPROGRAMAS INTERNOS Y EXTERNOS. BIBLIOTECAS

Este apartado es común tanto a funciones, como a procedimientos. Se utilizará el nombre de *subprograma*, que representará indistintamente a los dos tipos de subprogramas del lenguaje Pascal: funciones y procedimientos.

Subprograma interno: Un subprograma interno es aquel que se define dentro del mismo fichero o módulo que el programa principal.

Todos los subprogramas estudiados hasta este momento son de tipo interno. Los subprogramas internos pueden llevar a su vez anidados otros subprogramas que también son de tipo interno.

Subprograma externo: Un subprograma externo no está contenido dentro del mismo fichero o módulo que el programa principal. Los subprogramas externos pueden estar escritos en Pascal (por ejemplo en Turbo Pascal formando *units*). También pueden escribirse los subprogramas externos en otros lenguajes, como C y ensamblador.

Biblioteca o librería: es un conjunto de subprogramas externos agrupados y en código objeto. Estos subprogramas externos de la librería pueden utilizarse desde el módulo que realiza llamadas a la biblioteca. Las librerías se enlazan (*link*) con el código objeto del programa fuente para dar lugar al fichero ejecutable. Por ejemplo una biblioteca o librería de programas de manejo de un *plotter*.

Un subprograma externo puede ser llamado desde el programa principal o desde cualquier subprograma interno o externo, pero la cabecera tienen que ir declarada dentro del programa; es decir, *la cabecera de la definición original del subprograma seguida de la palabra reservada EXTERNAL y punto y coma (;)*.

Las variables globales no se transmiten del programa principal a los subprogramas externos. Es decir, la comunicación entre el subprograma externo y el programa que lo llama sólo se hace a través de la correspondencia argumento-parámetro, y si el subprograma es una función, devolviéndose el valor de la función asignado a su nombre. Un caso especial son la *units* de Turbo Pascal (apartado 7.12 de este capítulo).

Ejemplo 7.18

Se construye un subprograma interno para multiplicar enteros.

```
PROGRAM EjemploSupInterno (output);
VAR
  a,b,c:integer;

{-----}

FUNCTION Xmul(a,b:integer):integer;
BEGIN
  Mul:=a*b;
END;

{***** Programa principal *****}

BEGIN
  a:=40;
  b:=20;
  c:=Xmul(a,b);
  Writeln(c);
END.
```

Ejemplo 7.19

Se construye el mismo subprograma que en el ejemplo 7.18, pero como subprograma externo, y programado en lenguaje ensamblador.

Se utilizará la misma función *xMul* pero escrita en ensamblador 80x86. Esta función se debe de ensamblar con un macroensamblador, por ejemplo el Turbo Assembler (TASM) para crear un fichero con extensión *.OBJ* de la siguiente manera:

```
TASM Xmul
```

DECLARACION FORWARD

El código de la función *xMul* es el siguiente:

```
CODE SEGMENT BYTE PUBLIC
  ASSUME cs:code
  xmul proc near
    public xmul
    push bp
    mov bp, sp
    mov ax, [bp] + 4 ; coge el primer parámetro
    mul [bp] + 6    ; multiplica por el segundo
    pop bp         ; restaura bp y borra la pila
    ret 4         ; devuelve el resultado que está en ax.
  xmul endp
code ends
end
```

El código del programa que utiliza la función *xMul* anterior sería de la forma:

```
PROGRAM EjemploExternal (output);
VAR
  a, b, c : integer;
{-----}
  FUNCTION xMul(x, y: integer) :integer; EXTERNAL;
  {$L c:\Pascal\Xmul.obj} (* Necesario en Turbo Pascal *)
{***** Programa principal *****)
BEGIN
  a:=40;
  b:=20;
  c:=Xmul(a,b);
  Writeln(c);
END.
```

Si se compila y ejecuta este programa se obtendrá el mismo resultado que en el ejemplo 7.18.

7.8 DECLARACION FORWARD

Cuando una declaración de procedimiento o función especifica la directiva `forward` en vez de un bloque significa que se está realizando una *declaración forward*.

La sintaxis en notación EBNF y el diagrama sintáctico se muestran a continuación.

```
<declaración de procedimiento> ::= <cabeceraProcedure> ; FORWARD ;
<declaración de función> ::= <cabeceraFunction> ; FORWARD ;
```

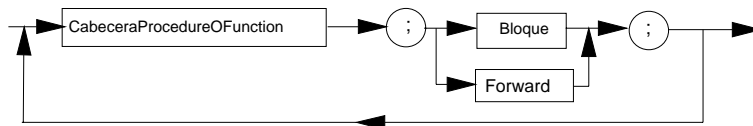


Figura 7.7 Declaración de subprograma con la directiva forward

SUBPROGRAMAS

Después de esta declaración, debe definirse el procedimiento o función con o sin la lista de parámetros formales y deben de aparecer en la misma parte de declaración de subprogramas. Es posible declarar entre ambas (declaración y definición) otros subprogramas y estos pueden llamar a los procedimientos o funciones declarados como *forward*. De esta forma se consigue recursividad mutua o indirecta.

Como utilidad práctica decir que sirve para que unos subprogramas llamen a otros sin preocuparse del orden de declaración de los mismos.

Ejemplo 7.20

```
PROCEDURE A (x:integer); Forward;
PROCEDURE B (VAR y:integer);
BEGIN
  A(y);
  Writeln(y);
END;

PROCEDURE A;
{ La lista de parámetros formales no está repetida }
BEGIN
  x := x + 2;
END;
```

Ejemplo 7.21

```
PROCEDURE A (x,y:integer); Forward;
PROCEDURE B (m,n:integer);
BEGIN
  ...
  A(2, 3);
  ...
END;

PROCEDURE A;
BEGIN
  ...
  B(5, 7);
  ...
END;
```

7.9 PROGRAMACION GRAFICA

En los programas desarrollados en los capítulos anteriores todas las operaciones de entrada y salida se realizan sobre dispositivos alfanuméricos (pantallas, impresoras, etc...). En esta sección se tratará de la programación de dispositivos gráficos. Para ello se utilizarán los subprogramas externos incluidos en las *unit Graph* y *Graph3* de Turbo Pascal. También se pueden utilizar otras bibliotecas gráficas.

SISTEMAS DE COORDENADAS

Los gráficos realizados con ordenador deben referirse a un sistema de coordenadas. Los periféricos empleados para representar los gráficos pueden clasificarse en dos grandes grupos *ráster* y *vectoriales*.

- **periféricos ráster o matriciales.** La representación gráfica se realiza en un dispositivo compuesto por elementos cuadrados o rectangulares, que están rellenos o no de un determinado color. Cada uno de los elementos más pequeños se denomina pixel. La palabra *pixel* proviene de *picture element*. Nótese que los *pixels* no son puntos, sino elementos cuadrados o rectangulares. Se denomina resolución al número de pixels horizontales por el número de pixels verticales (Figura 7.7). La mayor parte de las pantallas y las impresoras son de este tipo. Los *plotters* electrostáticos son también periféricos raster.

- **periféricos vectoriales.** La representación gráfica se realiza en un dispositivo por medio de puntos unidos por líneas. La única limitación es la precisión del dispositivo, es decir la distancia mínima a representar entre dos puntos. Los *plotters* vectoriales y algunas pantallas muy especiales son de este tipo.

En Informática Gráfica se manejan distintos sistemas de coordenadas:

- Coordenadas del periférico (*device coordinates*)
- Coordenadas normalizadas de periféricos (*normalized device coordinates*)
- Coordenadas propias del usuario (*world coordinates*)

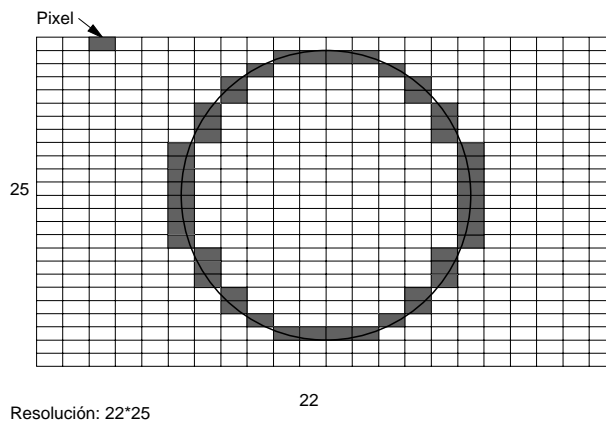


Figura 7.8 Pixels y resolución

COORDENADAS DEL PERIFERICO

El sistema de referencia se corresponde directamente con la resolución física del periférico (*device coordinates*).

Periféricos raster: las coordenadas son números enteros que corresponden con las de cada pixel, normalmente van del pixel (0,0) al pixel (MAX_RES_X-1, MAX_RES_Y-1).

Periféricos vectoriales: las coordenadas suelen ser números reales o enteros, que por ejemplo en un plotter corresponden con la posición en milímetros o pulgadas de cada punto sobre el papel de dibujo.

COORDENADAS NORMALIZADAS DE PERIFERICOS

Las coordenadas normalizadas de periféricos (*normalized device coordinates*) son independientes de las coordenadas del periférico. Las coordenadas normalizadas siempre expresan la resolución del periférico en el rango 0.0 a 1.0, es decir de la esquina superior izquierda (0.0,0.0) a la esquina inferior derecha (1.0,1.0).

COORDENADAS PROPIAS DEL USUARIO

Es el sistema de coordenadas definido por el usuario para resolver sus problemas concretos (*world coordinates*).

TRANSFORMACION DE COORDENADAS

La representación final siempre se realiza en coordenadas absolutas. El usuario o una biblioteca de gráficos tendrá que proveer un conjunto de funciones que realizan estas tareas.

Las operaciones habituales son la traslación de ejes, rotación de ejes, y cambio de escala.

ESCALADO

Habitualmente las coordenadas absolutas de los periféricos tienen distinta resolución horizontal y vertical. El cambio de escala o escalado permite ajustar las distintas resoluciones de forma que se pueda trabajar directamente con la misma resolución en los dos ejes. La transformación matemática es la siguiente:

$$\begin{aligned} X_f &= X * \text{factor_escala_X} \\ Y_f &= Y * \text{factor_escala_Y} \end{aligned}$$

Las ecuaciones anteriores pueden expresarse en forma matricial de la forma siguiente:

$$(x_f, y_f) = (x, y) \begin{pmatrix} f_x & 0 \\ 0 & f_y \end{pmatrix}$$

Las ecuaciones anteriores también pueden expresarse en coordenadas homogéneas de la forma siguiente:

PROGRAMACION GRAFICA

$$(x_f, y_f, 1) = (x, y, 1) \begin{pmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

TRASLACION

El origen de coordenadas puede trasladarse restando las coordenadas del vector de traslación (x_1, y_1) .

$$\begin{aligned} X_f &= X - x_1 \\ Y_f &= Y - y_1 \end{aligned}$$

Las ecuaciones anteriores también pueden expresarse en coordenadas homogéneas de la forma siguiente:

$$(x_f, y_f, 1) = (x, y, 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_1 & -y_1 & 1 \end{pmatrix}$$

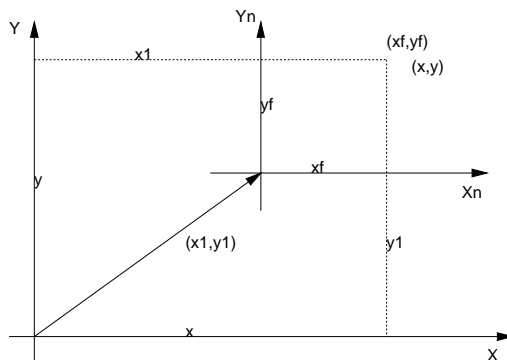


Figura 7.9 Esquema de traslación de ejes

ROTACION

Los ejes coordenados también se pueden rotar un ángulo α alrededor del origen.

$$\begin{aligned} x_f &= x \cos \alpha + y \sin \alpha \\ y_f &= -x \sin \alpha + y \cos \alpha \end{aligned}$$

Las ecuaciones anteriores también pueden expresarse en coordenadas homogéneas de la forma siguiente:

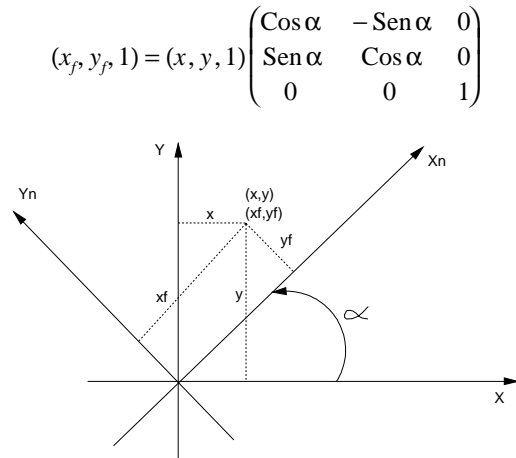


Figura. 7.10 Esquema de rotación de ejes

Ventanas gráficas (*viewports*)

Una ventana gráfica (*viewport*) es la zona de la superficie del periférico en la cual se representa el gráfico.

Recorte (*clipping*)

El recorte (*clipping*) de una imagen se produce cuando esta tiene un tamaño superior a la ventana gráfica activa en ese instante.

COLORES

Los periféricos gráficos son capaces de mostrar un conjunto finito de colores, conocido con el nombre de *paleta*. El número de colores que es capaz de manejar un periférico depende del número de *bits* almacenados por pixel (tabla 7.1).

RELACION DE ASPECTO (*aspect ratio*)

La relación de aspecto (*aspect ratio*) es un parámetro que describe la forma del *pixel* sobre un periférico. La relación de aspecto se describe mediante un número real, que es el resultado de dividir la anchura de un *pixel* por su altura.

Ejemplo: El adaptador gráfico CGA (*Color Graphics Adapter*) en modo 4 tiene una resolución de 640 por 200 *pixels*, y trabaja habitualmente en una pantalla de 9,6 pulgadas de ancho por 6,0 pulgadas de alto, así la anchura de un *pixel* es 0,015 pulgadas (9,6/640) y su altura es de 0,030 pulgadas (6,0/200). Entonces la relación de aspecto es 0,5 (0,015/0,030).

CURSOR GRAFICO

Habitualmente las funciones gráficas de una biblioteca mantienen la posición de un cursor gráfico invisible. Las coordenadas de este cursor gráfico son en *pixels*. La colocación del cursor en distintos puntos puede hacerse, por medio de funciones de movimiento relativas que los desplazan respecto de su situación actual, o por medio de funciones de movimiento absolutas que lo desplazan respecto al origen de coordenadas.

Nº de bits por pixel (n)	Nº de colores simultáneos (2 ⁿ)
1	2
2	4
4	16
8	256
12	4.096
16	65.536
20	1.048.576
24	16.777.216
32	4.294.967.296

Tabla 7.1 Relación entre el nº de colores y el nº de *bits* por *pixel*

PRIMITIVAS GRAFICAS

Las primitivas gráficas son las funciones que representan los distintos tipos de gráficos. Las primitivas habitualmente implementadas en las bibliotecas gráficas son las siguientes:

- Representación de puntos
- Representación de líneas
- Representación de rectángulos y cuadrados
- Representación de círculos, elipses y arcos
- Representación de texto en modo gráfico
- Copiar y pegar parte de una imagen gráfica

Antes de las llamadas a estas primitivas gráficas es necesario llamar a otras funciones que establecen las condiciones de uso de dichas primitivas. Las funciones de este tipo habituales en bibliotecas gráficas son las siguientes:

- Establecer modo gráfico
- Establecer colores
- Establecer tipo de líneas

SUBPROGRAMAS

- Establecer tipo de relleno de figuras cerradas
- Establecer tipo, tamaño y orientación de los textos

Otras primitivas son funciones que obtienen distintas características en un instante dado.

- Obtener periférico gráfico
- Obtener la relación de aspecto
- Obtener color de fondo
- Obtener color de dibujo actual
- Obtener información sobre colores
- Obtener el tipo de las líneas actuales
- Obtener información sobre el texto gráfico actual
- Obtener información sobre el resultado de la última operación gráfica

BIBLIOTECAS GRAFICAS DE TURBO PASCAL

Las bibliotecas gráficas varían de unos compiladores a otros, e incluso se venden por separado por compañías independientes de los fabricantes de compiladores. También pueden construirse directamente por llamadas a interrupciones de bajo nivel (véase capítulo 10).

Para utilizar la biblioteca (*unit*) de gráficos es necesario incluir después de la cabecera del programa:

```
USES  
    Graph;
```

El primer problema que se plantea es detectar el periférico (pantalla) disponible para ser utilizado, para lo cual se utiliza el siguiente mecanismo, con los subprogramas *Detect* e *InitGraph*.

```
PROGRAM IniciarGraph;  
USES  
    Graph;  
VAR  
    GraphDriver: Integer;  
    GraphMode: Integer;  
BEGIN  
    GraphDriver := Detect; (* Toma el modo de máxima resolución *)  
    InitGraph (GraphDriver, GraphMode, 'C:\TP\BGI');  
    ...  
END;
```

La función *Detect* determina el dispositivo gráfico disponible, asignándose éste a la variable *GraphDriver*.

PROGRAMACION GRAFICA

El procedimiento *initgraph()* inicializa un dispositivo gráfico y un modo gráfico, indicándose el camino (*path*) para encontrar los ficheros **.BGI* y **.CHR*, que contienen las especificaciones del modo gráfico (**.BGI*) y de las fuentes (*fonts*) de caracteres gráficos (**.CHR*). Si los ficheros necesarios no estuvieran en el directorio actual, el programa indicaría un error al inicializar gráficos. Los dispositivos gráficos están definidos por las constantes de la tabla 7.2.

Macro	Valor
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMOND	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Tabla 7.2: Macros de dispositivos gráficos

Algunos de los modos gráficos están definidos por los macros de la tabla 7.3. Las coordenadas de las distintas pantallas gráficas comienzan en la esquina superior izquierda (0,0) y acaban en la esquina inferior derecha. El valor máximo se puede obtener con dos funciones *GetMaxX* y *GetMaxY*. Así la esquina inferior derecha es (*GetMaxX, GetMaxY*). Todas las funciones gráficas están referidas a los *pixels* en este rango.

Macro	Resolución	Valor
CGA0	320 × 200	0
CGA1	320 × 200	1
CGA2	320 × 200	2
CGA3	320 × 200	3
CGAHI	640 × 200	4
VGALO	640 × 200	0
VGAMED	640 × 350	1
VGAHI	640 × 480	2
IBM8514LO	640 × 480	0
IBM8514HI	1024 × 768	1

Tabla 7.3: Macros de modos gráficos

SUBPROGRAMAS

A continuación se enumeran algunas funciones primitivas de gráficos. Estas primitivas trabajan directamente sobre las coordenadas del periférico, es decir sus coordenadas son los *pixels* de los distintos periféricos. El entorno integrado de desarrollo del compilador Turbo Pascal (IDE) permite pedir ayuda en línea de cualquiera de las siguientes funciones, tan sólo es necesario escribir el identificador de subprograma, situar el cursor encima, y pulsar las teclas **Ctrl+F1**.

- Representación de puntos: *PutPixel*
- Representación de líneas: *Line*, *LineTo*, y *LineRel*
- Representación de rectángulos, barras y polígonos: *Rectangle*, *Bar*, *Bar3D*, y *DrawPoly*, y *FillPoly*
- Representación de círculos, elipses y arcos: *Circle*, *Ellipse*, *FillEllipse*, *Arc*, *Sector*, y *PieSlice*
- Representación de texto en modo gráfico: *OutText*, y *OutTextXY*
- Copiar y pegar parte de una imagen gráfica: *GetImage*, *PutImage* y *ImageSize*

A continuación se enumeran algunos subprogramas de inicialización y establecimiento de condiciones.

- Establecer modo gráfico: *InitGraph*, *CloseGraph*, *InstallUserDriver*, y *RestoreCrtMode*
- Establecer colores de dibujo y de fondo: *SetColor*, *SetBkColor*, *SetPalette*, y *SetRGBPalette*
- Establecer tipo de líneas: *SetLineStyle*
- Establecer tipo de relleno de figuras cerradas: *FloodFill*
- Establecer tipo, tamaño y orientación de los textos: *InstallUserFont*, *SetTextStyle*, y *SetTextJustify*
- Establecer ventana gráfica: *SetViewPort*, y *ClearViewPort*
- Establecer relación de aspecto: *SetAspectRatio*

Subprogramas que obtienen distintas características en un instante dado:

- Obtener periférico gráfico: *DetectGraph*, *GetDriverName*, *GetGraphMode*, *GetMaxX*, *GetMaxY*, *GetMaxMode*, *GetModeName*, y *GetModeRange*.
- Obtener la relación de aspecto: *GetAspectRatio*
- Obtener color de fondo: *GetBkColor*
- Obtener color de dibujo actual: *GetColor*, y *GetPixel*
- Obtener información sobre colores: *GetMaxColor*, *GetPalette*, y *GetPaletteSize*
- Obtener el tipo de las líneas actuales: *GetLineSettings*

CONTROL DE PANTALLA ALFANUMERICA

- Obtener información sobre el texto gráfico actual: *GetTextSettings*
- Obtener información sobre el resultado de la última operación gráfica: *GraphResult*, y *GraphErrorMsg*

7.10 CONTROL DE PANTALLA ALFANUMERICA

El manejo de las pantallas de texto se realiza por medio de bibliotecas (*units*) incorporadas, que en los compiladores de Borland son *System* y *Crt*. Las funciones y procedimientos que incorporan pueden clasificarse en cuatro grupos:

- Definición de ventanas de texto. Se realiza con el procedimiento *Window*. Permite la definición de ventanas relativas dentro de la pantalla.
- Manipulación de pantalla. Se realiza con los procedimientos:

<i>ClrScr</i>	Borra ventana activa
<i>GotoXY</i>	Coloca el cursor en la posición especificada. La pantalla de texto admite valores de x (columnas) entre 1 y 80; y valores de y (filas) entre 1 y 25.
<i>ClrEol</i>	Borra desde el cursor hasta el final de línea actual
<i>DelLine</i>	Borra la línea sobre la que está el cursor
<i>InsLine</i>	Inserta una línea debajo de la posición actual del cursor
<i>TextMode</i>	Establece el modo texto

- Control de los atributos (colores, parpadeo,...). Se realiza con los subprogramas: *HighVideo*, *LowVideo*, *NormVideo*, *textAttr*, *TextBackground*, *TextColor*, y *TextMode*.
- Posición actual del cursor. Se obtiene con *WhereX*, y *WhereY*.

7.11 TIPOS PROCEDURALES DE TURBO PASCAL

En los diagramas sintácticos de los subprogramas se refleja que la lista de parámetros formales puede contener nombres de subprogramas, es decir, se permite el paso de funciones o procedimientos como parámetros. Esta característica de Pascal estándar, no se incluía en versiones anteriores del compilador. A partir de la versión 6.0 se incorpora por medio de los tipos procedurales.

Como su uso puede ser a veces de gran utilidad, es necesario conocer que Turbo Pascal permite que funciones y procedimientos se traten como entidades que puedan ser asignadas a variables y puedan pasarse como parámetros. Estas acciones se llevan a cabo por medio de **tipos procedurales**.

SUBPROGRAMAS

De esta forma, las siguientes declaraciones son válidas en Turbo Pascal.

```
TYPE
  Procraiz = FUNCTION (x:real): integer;
  Proclistado = PROCEDURE (a,b: real; VAR c,d:char);

VAR
  q: Procraiz;
  p: Proclistado;
```

Como ejemplo puede verse un programa completo en la sección de ejercicios resueltos (ejercicio 7.7).

7.12 UNITS DE TURBO PASCAL

El compilador Turbo Pascal permite utilizar subprogramas externos por medio de las denominadas *units* o TPU (Turbo Pascal Unit). Las TPU pueden incorporar también definiciones globales de constantes, tipos y variables. El propio compilador incorpora un conjunto de TPU estándar definidas a priori aunque también es posible crear TPU definidas por el usuario.

USO DE UNITS ESTANDAR

A continuación se muestran algunas *units* o TPU estándar que serán utilizadas en los ejemplos incorporados.

<i>Crt</i>	Conjunto de subprogramas para el manejo de pantalla, teclado, teclas especiales, colores en modo texto, ventanas y sonido
<i>Dos</i>	Permite utilizar funciones del sistema operativo, incluyendo el control de la fecha y la hora, búsqueda de directorios y ejecución de programas externos.
<i>Graph</i>	Conjunto de subprogramas, constantes y tipos que permite el manejo de gráficos sobre la mayoría de los tipos de pantalla del mercado.
<i>Graph3</i>	Conjunto de subprogramas, constantes y tipos que permite el manejo de gráficos con la biblioteca de Turbo Pascal 3.x.
<i>Overlay</i>	Permite el manejo de <i>overlays</i> . Los <i>overlays</i> son <i>units</i> que comparten un área de memoria común, reduciendo los requerimientos de memoria del programa ejecutable. Con el uso de <i>overlays</i> se pueden ejecutar programas que son mucho más grandes que el total de memoria RAM disponible.
<i>Printer</i>	Permite manejar la impresora.
<i>System</i>	Es utilizada siempre por el Turbo Pascal en tiempo de ejecución. Permite acceder y cambiar opciones avanzadas del Turbo Pascal.

Para utilizar los subprogramas, constantes y tipos de las TPU estándar debe ponerse la palabra reservada *USES* seguida del nombre de la TPU, si son varias se separan por comas. La palabra reservada *USES* debe colocarse inmediatamente detrás de la instrucción *PROGRAM*.

UNITS DE TURBO PASCAL

Ejemplo 7.22

Uso de la TPU Printer para imprimir una frase en impresora.

```
PROGRAM HolaImpresora(1st);
USES Printer;
BEGIN
  Writeln (1st, 'Hola Impresora ...');
END.
```

Ejemplo 7.23

Uso de la TPU Crt para borrar pantalla y colocar el cursor en una posición determinada en modo texto.

```
PROGRAM Pantalla (input, output);
USES Crt;
BEGIN
  ClrScr; (* Borra pantalla *)
  GotoXY(30,10);
  Write ('Comienzo a escribir en la columna 30 y en la fila 10');
END.
```

USO DE UNITS DEFINIDAS POR EL USUARIO

El usuario también puede construir sus propias units o TPU, que se compilan por separado y que pueden utilizarse desde otros programas al igual que las TPU estándar.

La estructura de una TPU tiene dos partes fundamentales, por un lado la parte denominada INTERFACE que contiene las declaraciones de constantes, tipos y subprogramas accesibles desde el exterior. Por otra parte está la parte IMPLEMENTACION en la cual está el código de todos los subprogramas. Existe también una parte opcional de inicialización. El esquema es el siguiente:

```
UNIT nombre;

INTERFACE

...
IMPLEMENTACION
...
END.
```

Se compilan en disco y crean un fichero con la extensión TPU.

Para usar las TPU desde un programa se hace de la misma forma que con las TPU estándar

```
USES nombre1, nombre2, ... , nombren;
```

La construcción de una *unit* o *TPU* se presenta en el ejercicio resuelto 7.5, y en los siguientes capítulos se utilizarán para implementar una primera aproximación a los tipos abstractos de datos.

7.13 EJERCICIOS RESUELTOS

- 7.1** Construir un programa que simule el funcionamiento de un reloj analógico en la pantalla del ordenador.

Solución. Con el fin de mostrar algunas de las posibilidades de programación gráfica que ofrecen muchas de las implementaciones comerciales del Pascal; a la vez que se ilustra la aplicación de la metodología de diseño descendente para la obtención de programas modulares mediante la descomposición de los mismos en procedimientos y funciones, se va a diseñar un programa que simule un reloj de agujas.

Se irán analizando cada una de las fases del proceso de programación, aplicadas a este problema concreto, partiendo de las especificaciones iniciales del problema (fase de análisis), hasta llegar a un hipotético mantenimiento del mismo.

Análisis

En concreto, se va a desarrollar en el entorno Turbo Pascal por lo que conviene comentar previamente algunos de los procedimientos incorporados por este compilador (extensiones respecto al estándar) que van a ser utilizados en el programa.

`Crt`: Conjunto de subprogramas para el manejo de pantalla, teclado, teclas especiales, colores en modo texto, ventanas y sonido.

`Graph3`: Biblioteca de subprogramas que permite la compatibilidad con los gráficos de la versión 3 de Turbo Pascal.

`GraphColorMode`: Prepara la pantalla para modo gráfico de 320 x 200 puntos.

`Palette(n:integer)`: Selecciona un conjunto (paleta) de cuatro colores.

`GraphBackGround(color:integer)`: Selecciona un color de fondo entre 16 posibles.

`Plot(x,y,color:integer)`: Dibuja el punto(x,y) especificado del color indicado como parámetro.

`Draw(x1,y1,x2,y2,color:integer)`: Traza una línea desde el punto (x1,y1) hasta el (x2,y2) del color indicado como parámetro.

En estos dos últimos procedimientos, si el parámetro color vale cero, se dibujará un punto (o una línea) del mismo color que el fondo elegido, lo que equivale a borrar el punto o la línea en cuestión. Ambos procedimientos toman como origen de coordenadas la esquina superior izquierda.

También queremos que nuestro reloj emita un pitido de una décima de segundo de duración y 50 Hz. de frecuencia, cada vez que la manecilla del segundero avance; y otro pitido de igual duración, pero más agudo (por ejemplo de 1200 Hz.) cada vez que transcurra un minuto.

Para ello utilizaremos los procedimientos incorporados:

EJERCICIOS RESUELTOS

`Sound(f:integer)`: Emite un sonido de la frecuencia especificada.

`NoSound`: Detiene la emisión de sonido.

`Delay(t:integer)`: Produce un retardo de t milisegundos.

El resto de las especificaciones del reloj son las siguientes:

- Radio del reloj = 95.
- Longitud aguja segundero = 90.
- Longitud aguja minuterero = 85.
- Longitud aguja horaria = 75.

Los valores vienen dados en pixels: unidad mínima de dibujo en pantalla.

Posición genérica de una aguja:

$$\begin{array}{ll} x1= -r*\cos(w) & x2= 1*\cos(w) \\ y1= -r*\sen(w) & y2= 1*\sen(w) \end{array}$$

El valor de 1 se ha especificado antes para cada una de las agujas. El valor de r vale 15 para todas las agujas.

Las agujas segundero y minuterero, deben avanzar ángulos discretos de 6° ($360/60$).

La aguja horaria avanza un ángulo de 6° cada 12 minutos de tiempo.

El reloj debe ser puesto en hora al principio de la ejecución del programa.

Una vez conocidas las especificaciones del programa y las herramientas de que disponemos, podemos pasar al diseño del algoritmo.

Algoritmo

NIVEL 0

```
Inicializar
Dibujar_Esfera
Dibujar_Agujas
REPETIR
    Esperar_Un_Segundo
    Avanzar_Agujas
    Emitir_Pitido
SIEMPRE.
```

NIVEL 1

```
Inicializar.
Preguntar la hora actual.
Leer la hora, minuto y segundo.
Activar modo gráfico.
Fijar paleta de colores a utilizar.
```

SUBPROGRAMAS

```
Dibujar_Esfera.  
(Dibujar un punto para cada una de las 12 horas)  
  PARA hora=1 HASTA 12  
    Calcular coordenadas del punto.  
    Dibujar punto.  
  FIN_PARA
```

```
Dibujar_Agujas.  
  PARA aguja=segundero HASTA horaria  
    Trazar_Aguja (aguja).  
  FIN_PARA
```

```
NIVEL 2.  
  Trazar_Aguja.  
    Calcular coordenadas de los puntos  
    extremos.  
    Trazar una línea entre esos puntos.
```

```
Avanzar_Agujas.  
  Borrar_Aguja (segundero).  
  Incrementar segundo MOD 60.  
  SI segundo=0  
    ENTONCES  
      Borrar_Aguja (minutero).  
      Incrementar minuto MOD 60.  
      SI minuto MOD 12 = 0  
        ENTONCES  
          Borrar_Aguja (horaria).  
          Incrementar hora MOD 60.  
    FIN_SI  
  FIN_SI
```

```
NIVEL 2.  
  Borrar_Aguja.  
    Calcular coordenadas de los puntos  
    extremos.  
    Borra la línea que une esos puntos.
```

```
Emitir_Pitido.  
  SI segundo=0  
    ENTONCES Emitir_Sonido de 1200 Hz.  
    SI_NO Emitir_Sonido de 800 Hz.  
  FIN_SI  
  Esperar 100 milisegundos.  
  Parar_Sonido.
```

```
Esperar_Un_Segundo.
```

Por sencillez, se realizará produciendo un retardo interno mediante el procedimiento Delay comentado anteriormente:

```
Delay (tiempo)
```

EJERCICIOS RESUELTOS

siendo tiempo una constante que se determinará empíricamente.

Sería mucho más exacto haber realizado este procedimiento mediante llamadas al sistema operativo para consultar el "reloj de tiempo real"; pero el uso de estas queda fuera de las pretensiones de este texto.

Codificación del programa

```
PROGRAM Reloj (input, output);
Uses Graph3, Crt;
CONST
  pi= 3.141592654;
  frecSeg=50; {Frecuencia en Hz. a emitir cada segundo }
  frecMin=1200;{Frecuencia en Hz. a emitir cada minuto }
  tiempo=500; {Tiempo de retardo }
  longMinutero=85;
  longHoraria=75;{Longitud de cada manecilla del reloj }
  longSegundero=90;
  r= 15;
  radio=95; {Radio de la esfera del reloj }
  factorEscala= 1.2;{Para conseguir forma circular }
TYPE
  agujas= ( segundero, minutero, horario );
VAR
  seg, min, hor :integer;
  { ----- }

PROCEDURE Inicializar (VAR hor, min, seg: integer);

BEGIN { Inicializar }
  Write('Hora, Minuto, Segundo ? : ');
  Readln(hor,min,seg);
  hor:=5*hor+min DIV 12;
  GraphColorMode;      { Fija modo gráfico }
  Palette(2);
  GraphBackground(0);  { Fondo de color negro }
END; { Inicializar }
{ ----- }

PROCEDURE DibujarEsfera;

  { Dibuja la esfera del reloj }

VAR
  xr,yr:real;
  x,y,n,j,i:integer;

BEGIN { DibujarEsfera }
  FOR n:=1 TO 12 DO
  BEGIN
    xr:=radio*factorEscala*Sin(n*pi/6);
    yr:=radio*Cos(n*pi/6);
    x:=Round(xr+160); { Cambio de ejes }
    y:=Round(100-yr);
    FOR j:=0 to 2 DO { Para que los puntos sean más visibles }
      FOR i:=0 to 2 DO { se dibujan 4 puntos contiguos }
        Plot(x+i,y+j,2)
      END;
    END;
  END; { DibujarEsfera }

  { ----- }
```

SUBPROGRAMAS

```

PROCEDURE Trazar ( aguja: agujas; hor,min,seg,modo:integer);
  { Traza la aguja especificada si el parámetro "modo" vale 1. Si
  vale cero la borra. }

  VAR
    x1,x2,
    y1,y2 :integer;

PROCEDURE Calcular (l,cont: integer; VAR x1,x2,y1,y2:integer);
  { Calcula las coordenadas de los extremos de una aguja. En el
  parámetro "cont" se recibirá el segundo, minuto u hora según la aguja
  cuyos extremos se quieran calcular }

  VAR
    angulo: real;

  BEGIN { Calcular }
    angulo:=pi*cont/30;
    x1:=Round (-r*factorEscala*Sin(angulo));
    y1:=Round (-r*Cos(angulo));
    x2:=Round (l*factorEscala*Sin(angulo));
    y2:=Round (l*Cos(angulo));
  END; { Calcular }

  BEGIN { Trazar }
    CASE aguja OF
      segundero: Calcular(longSegundero,seg,x1,x2,y1,y2);
      minuterio: Calcular(longMinuterio,min,x1,x2,y1,y2);
      horario : Calcular(longHoraria,hor,x1,x2,y1,y2)
    END;
    Draw(x1+160,100-y1,x2+160,100-y2, modo*(Ord(aguja)+1));
    { El cambio de ejes se hace al pasar los parámetros a Draw }
  END; { Trazar }

  { ----- }

PROCEDURE DibujarAgujas (hor,min,seg: integer);

  { Dibuja las agujas del reloj }

  VAR
    aguja: agujas;

  BEGIN { DibujarAgujas }
    FOR aguja:=segundero TO horario DO
      Trazar(aguja,hor,min,seg, 1);
    END; { DibujarAgujas }

    { ----- }

PROCEDURE AvanzarAgujas (VAR hor,min,seg: integer);

  { Avanza las agujas del reloj }

  VAR
    aguja: agujas;
  BEGIN { AvanzarAgujas }
    Trazar(segundero,hor,min,seg,0);
    seg:=(seg+1) MOD 60;
    IF seg=0 THEN
      BEGIN
        Trazar(minuterio,hor,min,seg,0);
        min:=(min+1) MOD 60;
      END
    IF min MOD 12=0 THEN
      BEGIN
        Trazar(horario,hor,min,seg,0);
      END
    END
  END

```

EJERCICIOS RESUELTOS

```
        hor:=(hor+1) MOD 60
    END;
    END;
    DibujarAgujas (hor,min,seg);
END; { AvanzarAgujas }

{ ----- }

PROCEDURE EsperarUnSegundo;

    { Produce un retardo controlado por la constante "tiempo" para
    conseguir que las agujas avancen aproximadamente cada segundo. }
    BEGIN { EsperarUnSegundo }
        Delay (tiempo);
    END; { EsperarUnSegundo }

    { ----- }

PROCEDURE EmitirPitido (min,seg: integer);

    { Emite un sonido de 800 Hz. Cada segundo, y de 1200 Hz. Cada minuto
    exacto; ambos de 100 milisegundos de duración. }

    BEGIN { EmitirPitido }
        IF seg=0 { Se ha cumplido un minuto exacto }
        THEN Sound(frecMin)
        ELSE Sound(frecSeg);
        Delay(100); { Espera 100 milisegundos }
        NoSound;
    END; { EmitirPitido }

    { ----- }

BEGIN { Programa Principal }
    Inicializar (hor, min, seg);
    DibujarEsfera;
    DibujarAgujas (hor, min, seg);
    REPEAT
        EsperarUnSegundo;
        AvanzarAgujas (hor, min, seg);
        EmitirPitido (min, seg);
    UNTIL false; { REPETIR-SIEMPRE }
END.
{FINAL DEL PROGRAMA PRINCIPAL}
```

Prueba del programa

Probablemente, al ejecutar nuestra primera versión del programa, nos habremos encontrado con un par de anomalías que hay que corregir.

Por un lado, es posible que las agujas avancen más deprisa (o más lento) de lo deseado. Será necesario actuar sobre la constante `tiempo` hasta conseguir un valor adecuado, según se había indicado anteriormente.

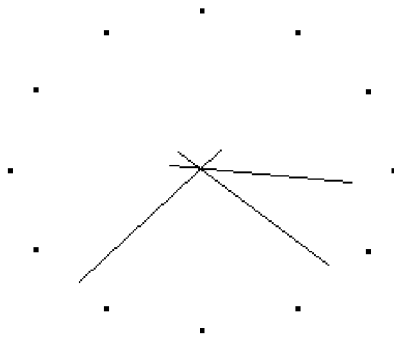
Pero, por otro lado, ha aparecido algo que no había sido previsto en la fase de análisis: si hemos calculado las coordenadas (x,y) de cada punto según las fórmulas dadas al principio, nos encontraremos con que el reloj aparece ovalado en vez de circular. Ello es debido a que la *relación de aspecto* y el ajuste de la pantalla hacen que los incrementos de x sean diferentes de los de y . Para corregirlo habrá que multiplicar

los valores de una de las coordenadas por un *factor de escala* hasta conseguir un círculo lo más perfecto posible. Esta corrección ya se ha incluido en el listado del programa mostrado anteriormente.

Mantenimiento

Quizá, tras un tiempo de uso del programa, estuviésemos interesados en ampliar algunas de las funciones del reloj. Por ejemplo para dibujar las agujas de diferente grosor; añadir la posibilidad de que el reloj tocara las *campanadas* correspondientes a cada hora en punto, o permitir que el programa se detenga al pulsar una tecla cualquiera en vez de estar ejecutándose en un bucle cerrado. Estas modificaciones resultarán más fáciles y rápidas de realizar si se ha seguido una metodología de diseño estructurada y se ha documentado adecuadamente el programa.

Ejecución



- 7.2** Escribir un programa que genere números pseudoaleatorios entre 0 y 1 a partir de una semilla introducida por teclado. Para la misma semilla se generará la misma serie de números aleatorios.

Solución

Algoritmo

```

INICIO
  Leer un entero (semilla)
  Leer número de elementos n a generar
  PARA j = 1 HASTA j = n
    Calcular valor de número aleatorio
  FIN_PARA
FIN
  
```

EJERCICIOS RESUELTOS

Codificación en Pascal

```
PROGRAM GeneracionAleatoria (input,output);
VAR
  semilla, j, n : integer;

{-----}

FUNCTION Aleatorio (VAR i:integer):real;
CONST
  k = 259;
BEGIN
  i:=k * i;
  IF i<0
  THEN i:=i+maxint+1;
  Aleatorio:=i/maxint;
END;

{***** Programa principal *****}

BEGIN
  Write ('Dame la semilla (un entero)');
  Readln (semilla);
  Write ('¿Cuántos nº quieres generar? ');
  Readln (n);
  FOR j:=1 TO n DO
    Writeln ( Aleatorio (semilla):7:5 );
  END.
```

7.3 Escribir un programa que genere quinielas con distintas probabilidades para el 1, X y el 2.

Solución

Algoritmo

```
INICIO
  Leer un entero (semilla)
  Leer probabilidad p1 del 1
  Leer probabilidad p2 del 2
  Leer probabilidad px de la x
  PARA j = 1 HASTA j = 15
    Calcular valor
  FIN_PARA
  SI valor < p1
    ENTONCES
      Escribir 1
  FIN_SI
  SI valor < p1 + px
    ENTONCES
      Escribir x
  SI_NO
    Escribir 2
  FIN_SI
FIN
```

SUBPROGRAMAS

Codificación en Pascal

```
PROGRAM Quiniela (input,output);

(* Este programa genera quinielas, con probabilidades *)
(* distintas para la X, el 1 o el 2. *)

VAR
  semilla, j, k, n : integer;
  p1, p2, px, valor: real;

{-----}

FUNCTION Aleatorio(VAR i:integer):real;
(* Genera números aleatorios entre 0 y 1, con distribución uniforme *)

CONST
  k = 259;

BEGIN
  i:=k*i;
  IF i
  THEN i:=i+maxint+1;
  Aleatorio:=i/maxint;
END;

{***** Programa principal *****}

BEGIN

  Write('Dame la semilla (un entero)');
  Readln(semilla);
  Write('¿Probabilidad del 1 en % ? ');
  Readln(p1);
  Write('¿Probabilidad del X en % ? ');
  Readln(px);
  p2:=100-p1-px;
  Write('Número de columnas que desea generar: ');
  Readln(n);
  FOR k:=1 TO n DO
    BEGIN
      FOR j:=1 to 15 DO
        BEGIN
          valor:= Aleatorio(semilla)*100;
          IF valor < p1
          THEN Write('1')
          ELSE IF valor < p1+px
          THEN Write('X')
          ELSE Write('2')
          END; (* FOR j *)
        Writeln;
      END; (* FOR k *)
    END.
  END.
```

7.4 Haciendo USO de la TPU *Graph* crear un programa en modo gráfico, que permita dibujar círculos, rectángulos, elipses para crear una tarta y escribir texto.

EJERCICIOS RESUELTOS

Solución

Algoritmo

```
INICIO
  SI no hay error al inicializar gráficos
  ENTONCES
    Dibujar rectángulo
    Dibujar un conjunto de círculos
    Dibujar un conjunto de elipses
    Dibujar una tarta
    Escribir texto
  SI_NO
    Escribir error al inicializar gráficos
  FIN_SI
FIN
```

Codificación en Pascal

```
PROGRAM PruebaDeGraficos;
USES Graph;
VAR
  DispositivoGrafico:integer;
  modoGrafico:integer;
  codigoError:integer;
  radio:integer;

{***** Programa principal *****}

BEGIN
  dispositivoGrafico := Detect; (* Detecta el tipo de pantalla *)

  (* Inicializa el modo gráfico *)
  InitGraph (dispositivoGrafico, ModoGrafico, 'C:\TP\BGI');

  (* Comprueba si existen errores *)
  codigoError := GraphResult;
  IF codigoError <> grOK
  THEN
    BEGIN
      Writeln (' Error en la inicialización de gráficos:',
              GraphErrorMsg (codigoError));
      Writeln (' FIN ');
      Halt(1);
    END;

  (* Dibuja un rectángulo *)
  Rectangle (0,0,GetMaxX,GetMaxY);

  (* Dibuja un conjunto de círculos concéntricos *)
  For radio := 1 TO 10 DO
    Circle (GetMaxX DIV 4,GetMaxY DIV 2,radio * 10);
  (* Dibuja un conjunto de elipses *)
  For radio := 1 TO 12 DO
    Ellipse(GetMaxX DIV 2,7*GetMaxY DIV 8,0,360,radio * 20, radio*2);

  (* Dibuja una tarta *)
```

SUBPROGRAMAS

```
PieSlice (4 * GetMaxX DIV 5 + 10, GetMaxY DIV 2,0,80,100);
PieSlice (4 * GetMaxX DIV 5,      GetMaxY DIV 2,80,270,100);
PieSlice (4 * GetMaxX DIV 5 + 10, GetMaxY DIV 2,270,300,100);

(* Escribe texto en modo gráfico *)

SetTextJustify (CenterText, CenterText);
SetTextStyle (DefaultFont, HorizDir, 3);
OutTextXY (GetMaxX DIV 2, GetMaxY DIV 8, 'DEMOSTRACION DE GRAFI-
COS');
SetTextStyle (GothicFont, VertDir, 2);
OutTextXY (GetMaxX DIV 20, GetMaxY DIV 2, 'V E R T I C A L');
Readln; (* Muestra la pantalla hasta que se pulsa INTRO *)
Closegraph; (* Vuelve a modo texto *)
END.
```

Ejecución



7.5 Crear una TPU que permita manejar fechas.

Solución

Algoritmo

```
INICIO
INTERFACE
  Dias_Desde_1960
  Dias_Entre_Fechas
  Igualdad_Fechas
  Posterioridad_Fechas

IMPLEMENTACION
  Dias_Desde_1960
  dias = parte_entera(30.42 * (mes - 1)) + dia
  SI mes >= 2 y mes < 8
  ENTONCES
    dias = dias + 1
  FIN_SI
  SI año MOD 4 = 0 y mes > 2
  ENTONCES
    dias = dias + 1
```

EJERCICIOS RESUELTOS

```
FIN_SI
SI (año - 60) DIV 4 > 0
  ENTONCES
    dias = dias + 1461 * ((año - 60) DIV 4) +1
FIN_SI
SI (año - 60) MOD 4 > 0
  ENTONCES
    dias = dias + 365 * ((año - 60) MOD 4) +1
FIN_SI

Dias_Entre_Fechas
Dias_Desde_1960(fecha1) - Dias_Desde_1960(fecha2)

Igualdad_Fechas
SI (dia1 = dia2) y (mes1 = mes2) y (año1 = año2)
  ENTONCES
    iguales = TRUE
  SI_NO
    iguales = FALSE
FIN_SI

Posterioridad_Fechas
posterior = FALSE
SI año1 > año2
  ENTONCES
    posterior = TRUE
  SI_NO
    SI año1 = año2
      ENTONCES
        SI (mes1 > mes2) o ((mes1=mes2) y (dia1 >
dia2))
          ENTONCES
            posterior = TRUE
        FIN_SI
      FIN_SI
    FIN_SI
FIN
```

Codificación en Pascal

```
UNIT fechas;
(* Manejo de fechas *)
INTERFACE
TYPE
  dia = 0..31;
  mes = 0..12;
  any = 00..99;
FUNCTION diasDesde1960 (d:dia; m:mes; a:any):integer;
FUNCTION diasEntreFechas (d1,d2:dia; m1,m2:mes; a1,a2:any):integer;
FUNCTION igualdad (d1,d2:dia; m1,m2:mes; a1,a2:any):boolean;
FUNCTION posterior (d1,d2:dia; m1,m2:mes; a1,a2:any):boolean;
IMPLEMENTATION

{-----}

FUNCTION diasDesde1960 (d:dia; m:mes; a:any):integer;
(* Esta función calcula el número de días transcurridos desde
el 1 de enero de 1960 hasta una fecha dada *)
```

SUBPROGRAMAS

```

VAR
  nd:integer;
BEGIN
  nd:= Trunc (30.42 * (m - 1)) + d;
  IF m = 2
    THEN nd:= nd + 1;
  IF (m > 2) AND (m < 8)
    THEN nd:= nd + 1;
  IF (a MOD 4 = 0) AND (m > 2)
    THEN nd:= nd + 1;
  IF (a - 60) DIV 4 > 0
    THEN nd:= nd + 1461 * ((a - 60) DIV 4);
  IF (a - 60) MOD 4 > 0
    THEN nd:= nd + 365 * ((a - 60) MOD 4) + 1;
  END;
  diasDesde1960:= nd
END;

{-----}

FUNCTION DiasEntreFechas (d1,d2:dia; m1,m2:mes; a1,a2:any):integer;
BEGIN
  DiasEntreFechas:= diasDesde1960 (d1, m1, a1) - diasDesde1960 (d2,
m2, a2)
END;

{-----}

FUNCTION igualdad (d1,d2:dia; m1,m2:mes; a1,a2:any):boolean;
BEGIN
  IF (d1 = d2) AND (m1 = m2)
    AND (a1 = a2)
    THEN igualdad:= TRUE
    ELSE igualdad:= FALSE;
END;

{-----}

FUNCTION posterior (d1,d2:dia; m1,m2:mes; a1,a2:any):boolean;
(* Verifica que f1, compuesta de d1,m1,a1, es posterior a f2
compuesta por d2,m2,a2 *)
BEGIN
  posterior:= FALSE;
  IF a1 > a2
    THEN posterior:= TRUE
    ELSE
      IF a1=a2
        THEN
          IF (m1 > m2) OR ((m1=m2) AND (d1 > d2))
            THEN posterior:= TRUE;
        END;
      END;
  END;
END.

```

- 7.6** Implementar un juego consistente en adivinar una clave de cuatro letras pertenecientes al rango A .. J en cinco intentos como máximo. Dicha clave será generada aleatoriamente por el ordenador.

EJERCICIOS RESUELTOS

Solución

Algoritmo

```
INICIO
  NIVEL 0
  GenerarClave
  MIENTRAS clave no acertada e intento < 5
  Leer clave
  PARA i = 1 HASTA i = 4
  Comprobar_Caracter
  FIN_PARA
  SI clave acertada
  ENTONCES
  Escribir acierto
  SI_NO
  Escribir clave generada
  FIN_SI

  NIVEL 1

  GenerarClave
  PARA i = 1 HASTA i = 4
  Generar caracteres aleatorios en el rango A .. J
  FIN_PARA
  Comprobar_Caracter
  SI los caracteres coinciden y están en la misma
posición
  ENTONCES
  Incrementar contador de aciertos
  FIN_SI
  SI los caracteres coinciden y están en diferente
posición
  ENTONCES
  Incrementar contador de aviso
  FIN_SI
FIN
```

Codificación en Pascal

```
PROGRAM Mastermind (input, output);
VAR c1, c2, c3, c4, a, b, c, d, respu : char;
    m, h, veces : integer;

(***** Generación aleatoria de la clave *****)

PROCEDURE GenerarClave (VAR c1, c2, c3, c4: char);
FUNCTION GenCar: char;

(* Genera caracteres aleatorios en el rango A..J *)

VAR n: real;
BEGIN
  n := 65 + (74 - 65) * Random;
  GenCar := chr(Round(n));
END;
```

SUBPROGRAMAS

```

{***** GenerarClave *****}

BEGIN
Randomize;
c1 := GenCar;
REPEAT
  c2 := GenCar;
UNTIL (c2 <> c1);
REPEAT
  c3 := GenCar;
UNTIL (c3 <> c1) AND (c3 <> c2);
REPEAT
  c4 := GenCar;
UNTIL (c4 <> c1) AND (c4 <> c2) AND (c4 <> c3);
END;

{-----}

PROCEDURE CompCar (car,muerto,her1,her2,her3: char; VAR nm,nh:integer);
BEGIN
  IF car = muerto
    THEN nm := nm + 1
  ELSE
    IF (car = her1) OR (car = her2) OR (car = her3)
      THEN nh := nh + 1;
END;

(***** Programa principal *****)

BEGIN
REPEAT
  GenerarClave (c1, c2, c3, c4);
  veces := 0;
  m := 0;
  WHILE (m < 4) AND (veces < 5) DO
    BEGIN
      Write ('Intento n° ', veces + 1, ' : ');
      Readln (a, b, c, d);
      a := Upcase(a);
      b := Upcase(b);
      c := Upcase(c);
      d := Upcase(d);
      m := 0;
      h := 0;
      CompCar (a, c1, c2, c3, c4, m, h);
      CompCar (b, c2, c1, c3, c4, m, h);
      CompCar (c, c3, c1, c2, c4, m, h);
      CompCar (d, c4, c1, c2, c3, m, h);
      veces := veces + 1;
      writeln (' muertos = ', m, ' heridos = ', h);
    END;
  IF m = 4
    THEN Writeln ('ACERTASTE')
  ELSE Writeln ('NO ACERTASTE. LA CLAVE ES: ', c1, c2, c3, c4);
  REPEAT
    Writeln ('¿Quieres probar otra vez? (S/n)');
    Readln (respu)
  UNTIL (respu = 's') OR (respu = 'S') OR (respu = 'n') OR (respu =
  'N');
  UNTIL (respu = 'n') OR (respu = 'N')
END.

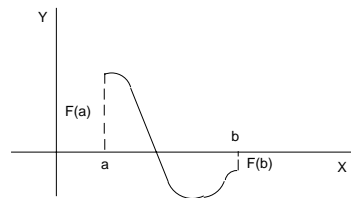
```

EJERCICIOS RESUELTOS

7.7 Crear un programa que haga uso de paso de funciones como parámetros para calcular la raíz de la función:

$$F(x) = x^5 + x + 3.0$$

según el gráfico siguiente



en el intervalo $[-2.0, 0.0]$ y con precisión $1e-7$ por el método de bisección.

Solución

Algoritmo

```
INICIO
  Nivel 0
    Llamar a la función Raiz en [a,b] y precisión esp

  Nivel 1
    Funcion
    Raiz
      m = (a+b)/2
      SI (Funcion(m) = 0) o (b - a < esp)
        ENTONCES
          raíz = m
        SI_NO
          SI (Funcion(a) * (Funcion(m) < 0)
            ENTONCES
              Buscar en la mitad izquierda del intervalo
            SI_NO
              Buscar en la mitad derecha del intervalo
          FIN_SI
        FIN_SI
      FIN_SI
    FIN
```

Codificación en Pascal

```
PROGRAM Biseccion (Input, Output);
{ Ejemplo de paso de funciones como parámetros }
USES crt;
TYPE
  procp = FUNCTION(x:real): real;

VAR
  x: real;
  f: procp;

{-----}
```

SUBPROGRAMAS

```
FUNCTION p (x:real): real; far;
{ Función a resolver }
BEGIN
  p := x*x*x*x*x+x+3.0;
END;

{-----}

FUNCTION raiz (f:procp; a,b,eps: real): real;
{ El intervalo es [a,b] y la precisión deseada esp }
VAR
  m :real;
BEGIN
  m := (a+b)/2.0;
  IF (f(m) = 0.0) OR (b - a < eps)
  THEN
    raiz := m
  ELSE
    IF (f(a) * f(m) < 0.0)
    THEN
      raiz := raiz (f,a,m,eps)
    ELSE
      raiz := raiz (f,m,b,eps);
  END;
END;

{***** Programa principal *****}

BEGIN
  Clrscr;
  f := p;
  x := raiz(f, -2.0, 0.0, 1e-7);
  Writeln ('La raíz aproximada es ', x);
  Writeln ('El valor de la función en ese punto es ',f(x));
  Writeln;
  Writeln ('Pulse una tecla para finalizar ');
  Readln;
END.
```

7.14 CUESTIONES Y EJERCICIOS PROPUESTOS

7.8 Escribir subprogramas que realicen las siguientes funciones matemáticas:

ArcSen(x)	ArcCos(x)	Th(x)	Ch(x)
Sh(x)	ArgSh(x)	ArgCh(x)	ArgTh(x)

7.9 Realizar un programa que sea una calculadora científica, con gran cantidad de funciones tanto matemáticas como estadísticas.

7.10 Añadir a la calculadora del ejercicio anterior, la posibilidad de que el programa anterior opere con números complejos.

7.11 Añadir a la calculadora del ejercicio anterior, la posibilidad de que calcule integrales y derivadas en un punto.

CUESTIONES Y EJERCICIOS PROPUESTOS

- 7.12** Escribir un procedimiento, denominado *BANNER*, para imprimir tus iniciales en letras de gran tamaño. Consiste en crear cada letra grande a base de usar líneas, con la misma letra, para construirla.
- 7.13** Escribir un procedimiento que lea una fecha de entrada y un número n para calcular la nueva fecha después de transcurrir n días.
- 7.14** Escribir un programa que utilice el procedimiento del ejercicio anterior para determinar las fechas de pagos de una empresa.
- 7.15** Escribir una función de tipo boolean que indique si una fecha es correcta o no, teniendo en cuenta el número de días de cada mes, y si el año es bisiesto o no.

7.16 Dada la declaración:

```
VAR a : integer;  
    x,y,z : real;  
  
PROCEDURE Nombre (x,y: integer; VAR z: real);
```

Indicar si las siguientes llamadas al procedimiento son correctas o incorrectas y por qué:

- a) Nombre (x, x, z);
- b) Nombre (x, y, a);
- c) Nombre (x-y, 5, z);
- d) Nombre (x, y, z-x);
- e) Nombre (x-a, y, a);

7.17 Escribir la salida que genera el siguiente programa

```
PROGRAM cuestion4;  
Uses  
    crt;  
var  
    a: integer;  
    x,y,z : real;  
  
PROCEDURE nombre (x,y :integer; VAR z: real);  
BEGIN  
    Writeln ('x = ',x);  
    Writeln ('y = ',y);  
    Writeln ('z = ',z:4:2);  
END;
```

SUBPROGRAMAS

```
BEGIN
  ClrScr;
  a := 1; x := 2.1; y := 3.2; z := 4.3;
  Writeln ('a');
  nombre (x,x,z);
  Writeln ('b');
  nombre (x,y,a);
  Writeln ('c');
  nombre(x-y,5,z);
  Writeln('d');
  nombre (x,y,z-x );
  Writeln('e');
  nombre (x-a,y,a);
  Readln;
END.
```

- 7.18** Crear una *unit (TPU)* que permita diseñar carátulas, cabeceras, menús y formatos de salida de datos a pantalla e impresora. Usar esta *unit* en una nueva versión del ejercicio 7.9.
- 7.19** Crear una *unit (TPU)* para el manejo cómodo de cualquier pantalla gráfica, de forma que trabaje directamente en un sistema de coordenadas de tipo real definido por el usuario, con el origen en el centro de la pantalla, y cuyos valores máximos son dados al inicializar gráficos. Usar esta *unit* para realizar una nueva versión del programa que simula un reloj analógico (ejercicio 7.1).
- 7.20** Diseñar y construir una *unit* con todas las funciones trigonométricas. Han de tenerse en cuenta los casos de indeterminación y los infinitos.
- 7.21** Diseñar y construir una *unit* con todas las funciones hiperbólicas.
- 7.22** Diseñar y construir una *unit* con todas las funciones necesarias para operar con números complejos.
- 7.23** Diseñar y construir una *unit* con funciones estadísticas (media, mediana, moda, varianza y desviación típica).
- 7.24** Volver a construir la calculadora científica avanzada de los ejercicios 7.9, 7.10 y 7.11 con las *units* diseñadas en los ejercicios 7.18, 7.20, 7.21 y 7.22.

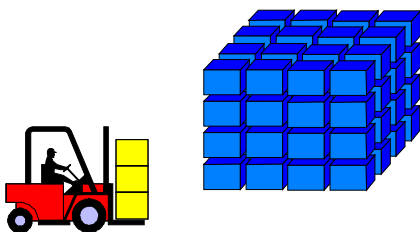
7.15 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Los subprogramas se estudian en todos los libros de introducción a la programación en los distintos lenguajes, en algunas obras también se utiliza el término *modularización*, y a la técnica de diseño de subprogramas *diseño modular*. Las técnicas de diseño modular indican que cada módulo debe ir provisto en su cabecera de *listas de importaciones* y de *listas de exportaciones*. La lista de importaciones serían los nombres de los subprogramas externos y parámetros que va a utilizar dicho módulo. La lista de exportaciones son los subprogramas del módulo que serán usados como subprogramas externos de otros módulos. El lenguaje que implementa explícitamente este concepto de módulo es el *MODULA-2*, que es un descendiente del lenguaje Pascal, y que también fue diseñado por *N. Wirth*. En Pascal estándar se puede suplir esta característica con comentarios en la cabecera de cada subprograma. En Turbo Pascal las *units (TPU)* tienen una sección de *INTERFACE* que define su comunicación con el exterior o lista de exportaciones. La lista de importaciones tan sólo se reduce a la lista de *units* que aparecen detrás de la cláusula *USES*. Sobre modularización puede consultarse el capítulo correspondiente en el libro *Curso de programación* de *J. Castro, F. Cucker, X. Messeguer, A. Rubio, Ll. Solano, y B. Valles* (McGraw-Hill, 1993).

El concepto de recursividad y el diseño de algoritmos recursivos puede profundizarse en el capítulo titulado *Algoritmos recursivos* del libro *Algoritmos + estructuras de datos = programas* de *Niklaus Wirth* (Ed. del Castillo, 1980).

Sobre programación gráfica puede usarse como libro de introducción el titulado *Gráficas por computadora* de *D. Hearn y M. P. Baker* en la editorial *Prentice-Hall* (1988). Contiene una introducción a los sistemas de gráficos, y a su programación con ejemplos en Pascal estándar. Como texto de introducción también se pueden recomendar las obra: *The art of computer graphics programming*, de *W. J. Mitchell, R. S. Liggett, y T. Kvan* (Ed. *Van Nostrand Reinhold*, 1987); tiene gran cantidad de ejemplos simples y criterios de diseño gráfico en Pascal estándar. Para la programación de gráficos usando las características propias del compilador Turbo Pascal se recomiendan las obras: *Advanced Graphics Programming in Turbo Pascal*, de *R.T. Stevens y C.D. Watkins* (Ed. M&T Books, 1992), y en el caso de programación de fractales *Fractal Programming in Turbo Pascal* de *R.T. Stevens* (Ed. M&T Books, 1992).

El diseño de subprogramas externos en lenguaje ensamblador con Turbo Pascal requiere el conocimiento del lenguaje ensamblador de la familia de microprocesadores 80x86. El compilador *Borland Pascal* incorpora el *Turbo Assembler*, con toda su documentación. Como texto con gran cantidad de ejercicios se recomienda la obra titulada *8088-8086/8087 programación ensamblador en entorno MS-DOS* de *M.A. Rodríguez-Roselló* (Ed. Anaya 1987). También se puede consultar como libro de iniciación el titulado *Lenguaje ensamblador para microcomputadoras IBM, para principiantes y avanzados* de *J. T. Godfrey* (Prentice-Hall, 1991).



CAPITULO 8

ESTRUCTURA DE DATOS ARRAY

CONTENIDOS

- 8.1 Introducción
- 8.2 Arrays bidimensionales
- 8.3 Operaciones con arrays completos
- 8.4 Arrays multidimensionales
- 8.5 Arrays empaquetados
- 8.6 Cadenas de caracteres
- 8.7 Extensión string
- 8.8 Concepto de tipo abstracto de datos (TAD)
- 8.9 Los arrays como tipos abstractos de datos.
- 8.10 Aplicaciones al Cálculo Numérico.
- 8.11 Extensiones del compilador Turbo Pascal
- 8.12 Cuestiones y ejercicios resueltos
- 8.13 Ejercicios propuestos
- 8.14 Ampliaciones y notas bibliográficas

INTRODUCCION

8.1 INTRODUCCION

Hasta este capítulo sólo se han estudiado los tipos simples: integer, char, boolean, tipos enumerados y tipos subrango. Todos los datos de tipo simple tienen como característica común que cada variable representa a un solo dato individual.

En este capítulo se estudiará un tipo de estructura de datos, que en el lenguaje de programación Pascal se conoce como *ARRAY*. En algunos libros la palabra *array* se traduce por arreglo, matriz, o tabla, en este libro no se traducirá.

Se entiende por *estructura de datos* la composición de tipos simples de una forma determinada. Las estructuras de datos permiten manipular grandes cantidades de información más fácilmente.

La estructura de datos de tipo *ARRAY* es una generalización de los conceptos de vector y matriz de Matemáticas. En Matemáticas un vector se puede representar por sus componentes de la siguiente forma:

$$V = (V_1, V_2, V_3, \dots, V_i, \dots, V_n)$$

donde la componente 1 es V_1 (se lee: V sub 1), la componente 2 es V_2 , etc...

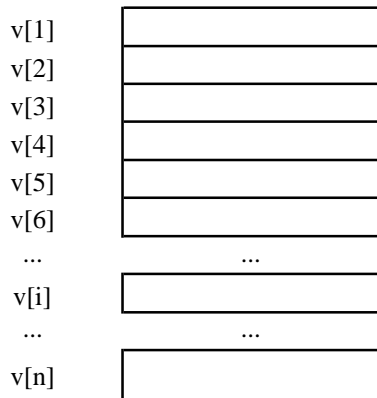


Figura 8.1 Representación de un vector en Informática

En Informática se pueden estructurar los tipos de datos simples como si fuesen las componentes de un vector. En la figura 8.1 se representan los componentes o elementos del *array* como rectángulos huecos. Dentro de cada hueco se meterán cada uno de los valores simples del *array*.

Cada una de las componentes del vector es un dato de tipo simple. En un determinado vector todos los componentes son del mismo tipo, llamado *tipo base*. Los subíndices del *array* no tienen que ser obligatoriamente números enteros, pueden ser de cualquier *tipo ordinal*, tal y como se muestra en la figura 8.2.

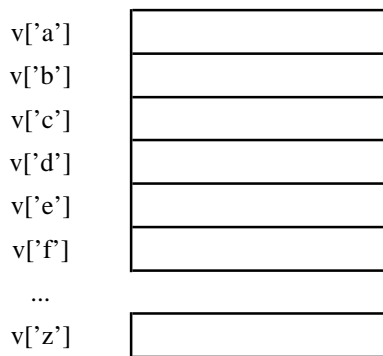


Figura 8.2 Array con tipo subíndice *char*

Los *arrays* se representan internamente en la memoria del ordenador como una sucesión consecutiva de los elementos simples que los componen. El ordenador accede a la posición de memoria donde se almacena un elemento de un *array* por el *identificador* de éste, *v* y por el *subíndice*, *i*. En Informática a los subíndices también se les denomina *índices*. Los subíndices de los componentes del vector, en muchos lenguajes de programación (entre ellos Pascal y C) se escriben entre corchetes [] y no entre paréntesis (como por ejemplo en FORTRAN y BASIC)¹⁵.

Los diagramas sintácticos que definen el tipo *array* son los de la figura 8.3. La notación *EBNF* del tipo *ARRAY* es:

```

<tipo array>      ::= ARRAY [<tipo índice> { , <tipo índice> } ] OF
                   <tipo componente>
<tipo índice>    ::= <tipo ordinal>
<tipo componente> ::= <tipo>

```

Ejemplo 8.1

Para operar con vectores de números reales, podríamos usar la siguiente definición de tipo:

```

TYPE
  valores = ARRAY [1..100] OF real;
VAR
  a, b, c: valores;

```

Esta definición de *ARRAY* es equivalente a la siguiente:

¹⁵ El motivo del uso del corchete con arrays es para que el compilador pueda diferenciar más fácilmente entre la llamada a una función y el uso de un elemento de un array. Por ejemplo la llamada a una función *a* con el argumento 1 es *a(1)*; y el componente 1 del array *a* es *a[1]*.

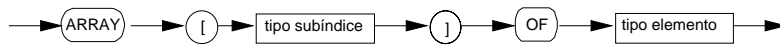
INTRODUCCION

```
VAR a: ARRAY [1..100] OF real;  
    b: ARRAY [1..100] OF real;  
    c: ARRAY [1..100] OF real;
```

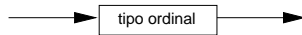
que todavía se puede poner de forma más escueta:

```
VAR a, b, c : ARRAY [1..100] OF real;
```

Tipo Array:



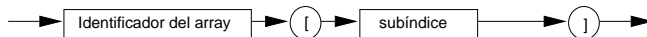
Tipo Subíndice:



Tipo elemento:



Variable con subíndice:



Subíndice:

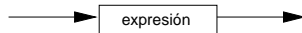


Figura 8.3 Diagrama sintáctico del tipo ARRAY

Los elementos de un *array*, se seleccionan por su *subíndice*. En Pascal, un subíndice puede ser cualquier expresión que dé un valor del tipo subíndice anterior o igual al límite superior declarado, y posterior o igual al límite inferior declarado¹⁶.

Ejemplo 8.2

Sea la declaración:

```
VAR x,y: ARRAY [1..100] OF integer;
```

16 El compilador Turbo Pascal tiene la directiva **{SR}** que comprueba rangos en tiempo de ejecución. En el caso de los arrays verifica si los subíndices están siempre entre los límites superior e inferior del array.

ESTRUCTURA DE DATOS ARRAY

Las variables con subíndice pueden aparecer en cualquier sitio en el que aparezcan variables simples, como se indica a continuación:

- a) `x[y[1] -j] := 3;`
- b) `x[i] := 2;`
- c) `x[i-2] := y[j];`
- d) `x[i] := Sqrt(y[j]);`

Como ejemplo práctico de utilización de *arrays*, resolveremos los siguientes ejercicios, que realizan operaciones básicas con *arrays*. En todos ellos utilizaremos las siguientes declaraciones:

```
CONST
  m = 100;
TYPE
  indice = 1..m;
  vector = ARRAY[indice] OF real;
```

Ejemplo 8.3

Escribir un subprograma para leer de teclado los elementos de un *array*.

Solución. Si conocemos el número de elementos del *array*, esta operación se suele realizar con un bucle *FOR*, recorriendo todas las posiciones del *array*, desde el límite inferior del tipo subíndice, hasta la última posición utilizada, que será igual al número de elementos. La estructura no suele rellenarse completamente de datos, y hay que tener la precaución de no acceder a las posiciones que no se han rellenado de datos, pues contienen valores imprevisibles.

Algoritmo

```
ACCION LeerVector ES:
  INICIO
    Leer el número de elementos, m
    DESDE i:=1 HASTA m HACER
      Leer elemento i, w[i]
    FIN_DESDE
  FIN
```

Codificación en Pascal

```
PROCEDURE LeerVector(VAR w: vector; VAR m:indice);
VAR i: indice;
BEGIN
  Write('¿Nº de elementos?');
  Readln(m);
  FOR i:=1 TO m DO
    BEGIN
      Write ('Elemento nº ',i,': ');
```


INTRODUCCION

```
    Readln(w[i]);
  END;
END;
```

Notas: Obsérvese que ambos parámetros, el vector, *w*, y el número de elementos, *m*, se declaran por dirección, pues tienen que ser devueltos al punto de llamada. Si no conocemos a priori el número de elementos, utilizaremos una estructura *WHILE* o *REPEAT*, con condición de parada, y contaremos el número de elementos introducidos, que suele ser útil para otras tareas.

Ejemplo 8.4

Modificar el procedimiento anterior, suponiendo que no se conoce de antemano el número de elementos a introducir en el *array*, sino que se detendrá la entrada de datos al introducir el número 0.0.

Algoritmo

```
ACCION LeerVector2 ES:
  INICIO
    i:=0 (* inicializar17 *)
  REPETIR
    i := i+1 (* incrementar contador *)
    Leer elemento i, w[i]
  HASTA w[i] = 0.0
  Eliminar último elemento, de parada (m:=i-1)
  FIN
```

Codificación en Pascal

```
PROCEDURE LeerVector(VAR w: vector; VAR m: indice);
VAR i: indice;
BEGIN
  i:=0;
  REPEAT
    i := i+1;
    Write ('Elemento n° ',i,'(0 para parar): ');
    Readln(w[i]);
  UNTIL w[i]=0.0
  m := i-1; (* Descontamos el último, de parada *)
END;
```

17 Todas las variables que se usen sin inicializarse previamente a un valor contienen **basura** (valores correspondientes a lo almacenado en binario en la posición de memoria de dicha variable). Es frecuente escuchar a un estudiante, que su programa funciona correctamente la primera vez que se ejecuta, y falla las veces siguientes. O también que su programa funciona unas veces y otras no. En estos casos debe comprobarse si hay variables sin inicializar. Pues ocurre que la primera ejecución pudo realizarse nada más encender el ordenador, estando todas las posiciones de memoria sin utilizar (habitualmente con valor cero). Las siguientes ejecuciones las posiciones de memoria ya contienen valores correspondientes a otros programas ejecutados.

Nota: Cuando no hay valores especiales de los datos que podamos utilizar para detener la entrada de los mismos, se realiza una pregunta, y se controla el final del proceso por el valor leído para la respuesta, generalmente en una variable caracter.

Ejemplo 8.5

Escribir un subprograma para escribir por pantalla los elementos de un *array*.

Solución. Esta operación se realiza casi siempre mediante un bucle *FOR*, pues el número de elementos siempre se puede conocer, contando los datos al introducirlos. En este caso el número de elementos es parámetro de entrada, declarado por valor. El vector también es parámetro de entrada, pero puede ser conveniente declararlo por dirección para ahorrar memoria, dependiendo del tamaño de la estructura.

Algoritmo

```
ACCION EscribeVector ES:
  INICIO
    DESDE i:=1 HASTA m HACER
      Escribir elemento i, w[i]
    FIN_DESDE
  FIN
```

Codificación en Pascal

```
PROCEDURE EscribeVector (w: vector; m:indice);
VAR i: indice;
BEGIN
  Writeln;
  Writeln('Elementos del vector:');
  Writeln('      N°           ELEMENTO');
  Writeln('-----');
  FOR i:=1 TO m DO writeln (i:8,w[i]:15:2);
  Writeln;
  Writeln ('Pulsa <intro> para continuar');
  Writeln;
  readln;
END;
```

Ejemplo 8.6

Realizar un subprograma que calcule la media y la desviación típica de un conjunto de números reales almacenados en un *array*.

Solución. Se podría realizar una función de tipo real para calcular la media, y otra función que utilice la anterior para la desviación típica. También se puede escribir un procedimiento, y declarar parámetros por dirección para devolver ambos valores. Recordemos el ejemplo 6.8 del capítulo 6, que calculaba la media de un conjunto de datos introducidos por teclado, utilizando una estructura *FOR*.

INTRODUCCION

La solución se obtiene aplicando las fórmulas:

Media:

$$x_m = \frac{\sum_{i=1}^n x_i}{n}$$

Desviación típica:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - x_m)^2}{n^2}}$$

Algoritmo

```
ACCION Estadística ES:
  NECESITA:
    vector x
    n° de elementos, n
  DEVUELVE:
    xm, σ
  INICIO
    s := 0      (* Inicializar acumulador del sumatorio *)
    DESDE contador:=1 HASTA n HACER
      s := s + x[contador]
    FIN_DESDE
    xm := s/n
    s := 0      (* usamos la misma variable auxiliar como acumu-
lador *)
    DESDE contador:=1 HASTA n HACER
      s := s + (x[contador]-xm)2
    FIN_DESDE
    σ :=  $\frac{\sqrt{s}}{n}$ 
  FIN
```

Codificación en Pascal

```
PROCEDURE Estadistica (x: vector; n: indice; VAR media, sigma: real);
VAR
  contador: indice;
  suma: real;
BEGIN
  suma := 0;
```

ESTRUCTURA DE DATOS ARRAY

```
FOR contador := 1 TO n DO
  suma := suma + x[contador];
media := suma / n;

(* Utilizamos la misma variable auxiliar como acumulador *)

suma := 0;
FOR contador := 1 TO n DO
  suma := suma + Sqr (x[contador] - media ) ;
sigma := Sqrt (suma)/n;

END;
```

Ejemplo 8.7

Efectuar las operaciones anteriores mediante funciones para el cálculo de la suma, media y desviación típica.

Solución. No repetiremos otra vez la fase de análisis y el algoritmo correspondiente, similares a los del ejemplo anterior.

Codificación en Pascal

```
FUNCTION Media(VAR v: vector; num: indice): real;
  (*****)
  FUNCTION Suma(VAR v: vector; num: indice): real;
  VAR
    s: real;
    contador: indice;
  BEGIN
    s := 0;
    FOR contador := 1 TO num DO
      s := s + v [contador];
    Suma := s;
  END;
  (*****)
  BEGIN
    Media := Suma(v, num) / num;
  END;
  (*****)

FUNCTION Sigma(VAR v: vector; num: indice): real;
VAR
  sumatorio, med: real;
  contador: indice;
BEGIN
  sumatorio := 0;
  med := Media(v, num);
  FOR contador := 1 TO num DO
    sumatorio := sumatorio + Sqr(v[contador] - med);
  Sigma := Sqrt(sumatorio)/num;
END;
```

Notas: Se ha utilizado una función local dentro de la función `Media` para el cálculo de la suma de los elementos del vector. También podríamos haber escrito la función `Media` como local, dentro de `Sigma`. Un ejemplo de utilización de estos subprogramas sería:

```
Writeln('La media es ', Media(x,n):6:3);
Writeln('La desviación típica es ', Sigma(x,n):6:3);
```

INTRODUCCION

También se podría hacer todo el cálculo en una única función, y con un sólo bucle utilizando la fórmula siguiente para la desviación típica.

$$\sigma = \sqrt{\frac{\sum_{i=1}^n x_i^2}{n} - \frac{\left(\sum_{i=1}^n x_i\right)^2}{n^2}}$$

Deben colocarse dos acumuladores: uno para los valores de x y otro para los valores de x^2 .

Ejemplo 8.8

Realizar un subprograma que ordene de mayor a menor los elementos de un *array* por el método de la burbuja.

Solución: Este método consiste en tomar el elemento que ocupa la última posición del *array* y compararlo con el elemento que ocupa la posición penúltima.

Si están mal ordenados se cambia el orden. Si están bien ordenados se dejan como están. El siguiente paso es comparar el elemento penúltimo con su predecesor, y así sucesivamente se comparan elementos sucesivos dos a dos. Con ésto se logra que los elementos colocados al final del *array* asciendan hasta donde su valor lo permita. Cuando terminamos las comparaciones, el mayor elemento ocupará la primera posición.

El proceso se repite otra vez, esta vez sin tocar la primera posición, desde el último hasta el segundo, con lo cual el mayor de los restantes quedará en segunda posición. Repetiremos ahora las comparaciones desde el último al tercero. Cuando solo nos queden dos elementos por comparar (el penúltimo y el último), tendremos el *array* ordenado.

Este método hace que el elemento "más ligero" suba hasta arriba, al igual que una burbuja en un líquido, de ahí su nombre.

El subprograma recibe como parámetros el vector y el n° de elementos, y devuelve el vector ordenado al punto de llamada. El vector tiene que pasarse por dirección al procedimiento de ordenación, dado que entra desordenado y sale ordenado.

Algoritmo

Para el diseño del algoritmo aplicamos la técnica de diseño descendente. Podemos descomponer el problema en dos niveles de abstracción:

NIVEL 0:

```
ACCION Ordena ES:  
  NECESITA: vector w  
           n° de elementos, m  
  DEVUELVE: vector w ordenado descendentemente  
INICIO
```

ESTRUCTURA DE DATOS ARRAY

```
DESDE j := 2 HASTA m HACER
  DESDE i := m descendiendo HASTA j HACER
    SI w[i-1] > w[i]
      ENTONCES Intercambiar (w[i-1], w[i]);
    FIN_SI;
  FIN_DESDE i;
FIN_DESDE j;
FIN
```

NIVEL 1:

```
ACCION Intercambiar (a, b) ES
  INICIO
    aux := a
    a := b
    b := aux
  FIN
```

Codificación en Pascal

```
PROCEDURE Ordena ( VAR w: vector; m:indice);
VAR
  i,j: indice;
  aux: real;
BEGIN
  Write ('Ordenando');
  FOR j:=2 TO m DO
    FOR i:= m DOWNTO j DO
      IF w[i-1] > w[i] THEN
        BEGIN
          Write('.');
          aux := w[i-1];
          w[i-1] := w[i];
          w[i] := aux;
        END;
    END;
  Writeln;
END;
```

Nota: Los procedimientos estándar de escritura (*Write*) sirven para comprobar que se está ejecutando el programa, ya que no hay otras sentencias de salida a pantalla.

Ejemplo 8.9

Escribiremos ahora un programa que utilice los subprogramas anteriores, para ver un ejemplo de cómo utilizarlos.

Solución. Utilizando los subprogramas construidos, leeremos los elementos del vector, calcularemos su valor medio y su desviación típica, y ordenaremos el vector descendientemente. Escribiremos los elementos del vector después de leerlos y después de la ordenación.

INTRODUCCION

Algoritmo

NIVEL 0:

INICIO

```
LeerVector (w, m);
EscribeVector (w, m);
Estadística (w, m,  $x_m$ ,  $\sigma$ );
Escribir  $x_m$ ,  $\sigma$ ;
Ordena (w, m);
EscribeVector (w, m);
```

FIN

NIVEL 1:

Los algoritmos de las acciones LeerVector, EscribeVector, Estadística y Ordena, son los construidos en los ejemplos anteriores, 8.3, 8.5, 8.6 y 8.8

Codificación en Pascal

```
PROGRAM Vectores(input, output);
CONST
  m = 100;
TYPE
  indice = 1..m;
  vector = ARRAY[indice] OF real;
VAR
  v: vector;
  n: indice;
  med, sig: real;
(*-----*)
PROCEDURE LeerVector(VAR w: vector; VAR m:indice);
VAR i: indice;
BEGIN
  Write('¿Nº de elementos?');
  Readln(m);
  FOR i:=1 TO m DO
    BEGIN
      Write ('Elemento nº ',i,': ');
      Readln(w[i]);
    END;
END;
(*-----*)
PROCEDURE EscribeVector (w: vector; m:indice);
VAR i: indice;
BEGIN
  Writeln;
  Writeln('Elementos del vector:');
  Writeln('      N°          ELEMENTO');
  Writeln('-----');
  FOR i:=1 TO m DO writeln (i:8,w[i]:15:2);
  Writeln;
  Writeln ('Pulsa <intro> para continuar');
  Writeln;
  readln;
END;
(*-----*)
```

ESTRUCTURA DE DATOS ARRAY

```
PROCEDURE Estadistica (x: vector; m: indice; VAR media, sigma: real);
VAR
    contador: indice;
    suma: real;
BEGIN
    suma := 0;
    FOR contador := 1 TO m DO
        suma := suma + x[contador];
    media := suma / m;
    (* Utilizamos la misma variable auxiliar para el cálculo de sigma *)

    suma := 0;
    FOR contador := 1 TO m DO
        suma := suma + Sqr (x[contador] - media ) ;
    sigma := Sqrt (suma/m);

END;
(*-----*)
PROCEDURE Ordena ( VAR w: vector; m:indice);
VAR i,j: indice;
    aux: real;
BEGIN
    Write ('Ordenando');
    FOR j:=2 TO m DO
        FOR i:= m DOWNTO j DO
            IF w[i-1] > w[i] THEN
                BEGIN
                    Write('.');
                    aux := w[i-1];
                    w[i-1] := w[i];
                    w[i] := aux;
                END;
        END;
    Writeln;
END;
(*-----*)
BEGIN (* PROGRAMA PRINCIPAL *)
    Writeln('PROGRAMA EJEMPLO DE UTILIZACION DE ARRAYS');
    Writeln;
    Writeln('LECTURA DE LOS ELEMENTOS DEL VECTOR');
    LeerVector(v, n);
    Writeln('ESTADISTICA DE SUS DATOS:');
    Estadistica(v, n, med, sig);
    Writeln(' Valor medio...', med:6:2);
    Writeln(' Desviación típica...', sig:6:2);
    Writeln;
    Writeln(' ORDENACION POR EL METODO DE LA BURBUJA:');
    Ordena(v, n);
    Writeln('El vector ordenado es:');
    EscribeVector(v, n);
    Write('Pulsa <Intro> para acabar...');
    Readln;
END.
```

8.2 ARRAYS BIDIMENSIONALES

Hasta ahora sólo se han estudiado *arrays unidimensionales* o vectores. Sin embargo se pueden definir en Pascal *arrays multidimensionales*.

En el caso de dos dimensiones la estructura es de la forma de una matriz de Matemáticas:

ARRAYS BIDIMENSIONALES

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

Aquí se ha representado una matriz de m filas y n columnas.

Según el diagrama sintáctico, los elementos de un *array* pueden ser a su vez del tipo *ARRAY*. Por lo tanto un *array multidimensional* puede definirse de la siguiente manera (caso de 2 dimensiones):

```
VAR x: ARRAY [1..10] OF ARRAY [1..10] OF real;
```

La forma anterior es equivalente a

```
TYPE fila = ARRAY [1..10] OF real;
      tabla = ARRAY [1..10] OF fila;
VAR x: tabla;
```

siendo otra forma más abreviada:

```
VAR x: ARRAY [1..10, 1 .. 10] OF real;
```

Esta notación abreviada es la que más se asemeja a la notación matemática. También está contemplada en el diagrama sintáctico del tipo *ARRAY*.

Los distintos componentes del *array* declarado anteriormente, se pueden manejar como variables subindicadas de la forma:

$x[i, j]$ o $x[i][j]$

o para indicar un elemento concreto:

```
x[1,1]      o      x[1][1]
x[2,5]      o      x[2][5]
x[7,90]     o      x[7][90]
```

Se pueden utilizar los operadores válidos para las variables simples con las variaciones subindicadas, que denotan los elementos de un *array*. Por ejemplo:

```
x[i,7] := x[2,5] + 23.77
a[i] := x[5,7] - x[j,j]
```

donde $a[i]$ es un elemento de un *array* unidimensional de reales, x es un *array* bidimensional de reales, e i, j son subíndices de tipo ordinal.

Ejemplo 8.10

Escribir por pantalla los elementos de una matriz de números reales.

Solución. Cuando se conoce el número de filas y columnas de un *array* bidimensional, es muy frecuente utilizar un doble bucle *FOR* para recorrer la estructura, ya sea para la lectura o escritura de los elementos de matriz, o en general para aquellas operaciones que precisen recorrer uno a uno y ordenadamente todos sus elementos. Generalmente el bucle exterior recorre las filas, y el bucle interior recorre, para cada fila, todos los elementos desde la primera columna hasta la última.

Algoritmo

```

ACCION EscribeMatriz ES:
  NECESITA: Matriz t
           N° de filas, nf, y n° de columnas, nc
  PRODUCE: Salida por pantalla de la matriz (números rea-
les)
  INICIO
    DESDE i:=1 HASTA nf HACER
      DESDE j:= 1 HASTA nc HACER
        Escribir t[i,j]
      FIN_DESDE j
    FIN_DESDE i
  FIN

```

Codificación en Pascal

Supuestas realizadas las declaraciones

```

CONST
  numFil = 10;
  numCol = 10;
TYPE
  indiceFil = 1..numFil;
  indiceCol = 1..numCol;
  fila = ARRAY [indiceFil] OF real;
  tabla = ARRAY [indiceCol] OF fila;
VAR
  x: tabla;

```

podríamos escribir por pantalla los elementos de x mediante el siguiente subprograma:

```

PROCEDURE EscribeMatriz (VAR t: tabla; nf: indiceFil; nc: indiceCol);
VAR
  i: indiceFil;
  j: indiceCol;

```

ARRAYS BIDIMENSIONALES

```
BEGIN
FOR i:=1 TO nf DO
  BEGIN
  Write ('|');
  FOR j:=1 TO nc DO
    Write (t[i,j]:5:1, ' ');
  Writeln('|');
  END;
END;
```

Notas: Obsérvese que el parámetro t se declara por dirección, aunque es un dato de entrada, con el fin de ahorrar memoria. El siguiente ejemplo incluye subprogramas para leer y manipular los datos.

Ejemplo 8.11

Realizar un programa que a partir de la facturación mensual de una empresa nos determine:

- a) La facturación anual de cada año
- b) La facturación media mensual

Solución: La estructura del programa se basa en almacenar los datos en una matriz de tantas filas como años y de doce columnas, que representan los meses:

	enero	febrero	...	diciembre
1980	$x_{1,1}$	$x_{1,2}$...	$x_{1,12}$
1981	$x_{2,1}$	$x_{2,2}$...	$x_{3,12}$
...

La suma de las filas nos dará la facturación anual. La suma de las columnas dividida por el número de años nos da la facturación media mensual.

Para facilitar el acceso a los datos, utilizamos como tipos subíndice los años y los meses:

```
CONST
  m = 12;
TYPE
  anual = 1900..2000;
  meses = 1..m;
  factura = ARRAY [anual, meses] OF real;
```

De este modo, para referirnos por ejemplo a la facturación de mayo de 1980 escribiremos:

$x[1980,5]$

Podríamos utilizar un tipo enumerado con los nombres de los meses, pero complicaríamos la escritura de los índices en el apartado **b)**. Recordemos que las variables de tipo enumerado no se pueden leer ni escribir directamente.

ESTRUCTURA DE DATOS ARRAY

Algoritmo

NIVEL 0:

INICIO

 LeeDatos (Lee años inicial y final, y datos de la matriz)

 ListaFactAnual (Calcula y lista facturaciones anuales)

 ListaMediaMensual (Calcula y lista las medias mensuales)

FIN

NIVEL 1:

ACCION LeeDatos ES:

 NECESITA:

 DEVUELVE: años inicial y final, matriz de facturaciones

 INICIO

 Leer año inicial

 Leer año final

 DESDE i:= año inicial HASTA final HACER

 DESDE j:=1 HASTA 12 (meses) HACER

 Leer x[i,j]

 FIN_DESDE j

 FIN_DESDE i

 FIN

ACCION ListaFactAnual ES:

 NECESITA: años inicial y final, matriz de facturaciones

 PRODUCE: Cálculo y listado de facturaciones anuales

 INICIO

 Escribir cabecera;

 DESDE i:= año inicial HASTA final HACER

 Inicializar suma := 0;

 DESDE j:=1 HASTA 12 (meses) HACER

 suma := suma + t[i,j]

 FIN_DESDE j

 Escribir año i, suma (facturación del año i)

 FIN_DESDE i

 FIN

ACCION ListaMediaMensual ES:

 NECESITA: años inicial y final, matriz de facturaciones

 PRODUCE: Cálculo y listado de medias mensuales

 INICIO

 Escribir cabecera;

 DESDE j:=1 HASTA 12 (meses) HACER

 Inicializar suma := 0;

 DESDE i:= año inicial HASTA final HACER

 suma := suma + t[i,j] (j cte, varía el año i)

 FIN_DESDE i

 suma := suma/nº de años (final-inicial+1)

 Escribir mes j, suma (media del mes j)

 FIN_DESDE j

 FIN

ARRAYS BIDIMENSIONALES

Codificación en Pascal

```
PROGRAM Facturacion (input,output);
CONST
  m = 12;
TYPE
  anual = 1900..2000;
  meses = 1..m;
  factura = ARRAY [anual, meses] OF real;
VAR
  x: factura;
  anio1, anio2: anual;
  (*****)
PROCEDURE LeeDatos(VAR f: factura; VAR inicial, final: anual);
VAR
  i: anual;
  j: meses;
BEGIN
  Writeln;
  Writeln('***** FACTURACION *****');
  Writeln;
  Write('Introduzca el primer año del que desea realizar el estudio ');
  Readln(inicial);
  Write('Introduzca el último año a estudiar ');
  Readln(final);
  FOR i := inicial TO final DO
    FOR j := 1 TO m DO
      BEGIN
        Write('Facturación del mes ',j:2,' del año ',i:4,' = ');
        Readln(x[i,j])
      END
    END;
  (*****)
PROCEDURE ListaFactAnual(VAR f: factura; VAR inicial, final: anual);
VAR
  i: anual;
  j: meses;
  suma: real;
BEGIN
  (* CALCULO Y LISTADO DE LA FACTURACION ANUAL *)
  Writeln;
  Writeln('          AÑO          FACTURACION ANUAL');
  Writeln('          *****          *****');
  Writeln;
  FOR i:=inicial TO final DO
    BEGIN
      suma:=0;
      FOR j:=1 TO m DO suma := suma + f[i,j];
      Writeln (i:17, suma:30:3)
    END;
  END;
  (*****)
PROCEDURE ListaMediaMensual(VAR f: factura; VAR inicial, final: anual);
VAR
  i: anual;
  j: meses;
  suma: real;
BEGIN
  (* CALCULO DE LA FACTURACION MEDIA MENSUAL *)
  Writeln;
  Writeln('          MES          FACTURACION MEDIA');
  Writeln('          *****          *****');
  Writeln;
  FOR j:=1 TO m DO
    BEGIN
      suma:=0;
```

ESTRUCTURA DE DATOS ARRAY

```
FOR i:=inicial TO final DO suma := suma + x[i,j];
suma := suma/(final-inicial+1);
Writeln(j:17, suma:30:3)
END
END;
(*****
BEGIN      (* Programa principal *)
  LeeDatos(x, anio1, anio2);
  ListaFactAnual(x, anio1, anio2);
  ListaMediaMensual(x, anio1, anio2);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

8.3 OPERACIONES CON ARRAYS COMPLETOS

Existen ciertos tipos de operaciones que pueden efectuarse sobre un *array* completo, afectando del mismo modo a todos los elementos del *array*.

Para realizar estas operaciones se deben de definir los *arrays* con los que se va a operar como de un mismo tipo.

Ejemplo 8.12

Con las siguientes declaraciones:

```
TYPE
  tabla = ARRAY [1.. 100, 1..25] OF real;
VAR
  a, b, c, d : tabla;
  fila, columna: integer ;
```

se puede realizar la operación de asignación de todos los elementos de un *array* a otro *array*, con una sola sentencia:

```
a := b;
```

Nótese que la sentencia anterior es equivalente a un doble bucle anidado:

```
FOR fila := 1 TO 100 DO
  FOR columna := 1 TO 25 DO
    a[fila, columna] := b[fila, columna];
```

Operaciones no permitidas

- No se pueden introducir *arrays* en expresiones numéricas o booleanas:

```
c := a + b;
```

Esto es incorrecto, pues *a*, *b*, *c* son del tipo *ARRAY*.

- Tampoco se pueden leer *arrays* completos con una sola sentencia *Read* o *Readln*:

OPERACIONES CON ARRAYS COMPLETOS

```
Read(a)
```

Esto es incorrecto, pues *a* es del tipo *ARRAY*.

Para leer los *arrays*, se suelen utilizar uno o varios bucles (según la dimensión) para leer elemento a elemento.

- Los descriptores de tipo tienen que ser exactamente iguales para hacer las asignaciones. Así, si se tienen las siguientes declaraciones:

```
TYPE
  vector = ARRAY [1..10] OF real;
VAR
  a: vector;
  b: ARRAY [1..10] OF real;
```

la asignación de la forma

```
b := a
```

no está permitida y dará un mensaje de error.

- No se pueden asignar constantes a un *array*. Por lo tanto no está permitido hacer:

```
a := 27.3
```

Asignación de los elementos de la fila de un *array* a un *array unidimensional*

Se pueden mezclar *arrays* de distintas dimensiones, tal y como se muestra en el ejemplo siguiente:

Ejemplo 8.13

```
PROGRAM Pantallazos;
CONST
  anchoPantalla = 80;
  longitudPantalla = 24;
TYPE
  columna = 1..anchoPantalla;
  fila = 1..longitudPantalla;
  linea = ARRAY [columna] OF char;
  pantalla = ARRAY [fila] OF linea;
VAR
  lineal, linea2: linea;
  videoPantalla: pantalla;
  m, i : fila;
  n, j : columna;
BEGIN
  ...
  ...
  FOR m:=1 TO anchoPantalla DO
    lineal[m] := Chr(m+64);          (* Bucles para rellenar de datos *)
  FOR n:=1 TO longitudPantalla DO   (* la matriz *)
    videoPantalla[n] := lineal;
  ...
```

```

    ...
    linea2 := videoPantalla [10];
    ...
    ...
END.

```

Con la asignación:

```
videoPantalla[n] := linea1;
```

copiamos los 80 elementos del *array* unidimensional *linea1* a la fila *n* del *array* bidimensional *videopantalla*. Es una asignación de un *array* fila a una fila de un *array* bidimensional.

Con la asignación:

```
linea2 := videoPantalla [10];
```

Se han asignado los 80 elementos de la fila 10 del *array* *videoPantalla* al *array* unidimensional *linea2*.

Por supuesto, operaciones similares pueden realizarse con *arrays* unidimensionales y columnas de un *array* bidimensional, siempre que las variables a asignar sean exactamente del mismo tipo.

8.4 ARRAYS MULTIDIMENSIONALES

El lenguaje Pascal no pone límite a las dimensiones de un *array* (recordemos el diagrama sintáctico del tipo *ARRAY*, fig. 8.3), por lo que se permiten *arrays* con más de dos dimensiones. El número de dimensiones de un *array* es el mismo que el número de índices que se necesitan para referenciar una celda particular. Por ejemplo un *array tridimensional* se representa en la figura 8.4, donde cada celda representa al elemento $[i, j, k]$ del *array*.

El número de los elementos de un *array* multidimensional es el producto del número de elementos de cada dimensión. Así un *array* de $100 \times 100 \times 100$ tiene 1.000.000 de elementos, que puede ser superior a la capacidad de muchos microordenadores. Se puede observar que los *arrays* son muy ávidos de memoria. La estructura de datos *array* en Pascal es una estructura estática de datos, dado que su tamaño máximo se debe definir en tiempo de compilación, reservándose la memoria necesaria como si se fuese a ocupar hasta el tamaño máximo. En el capítulo 12 se estudiarán las estructuras dinámicas de datos, que no necesitan definir su tamaño máximo en tiempo de compilación, y se pueden ajustar a las necesidades exactas definidas en tiempo de ejecución.

ARRAYS MULTIDIMENSIONALES

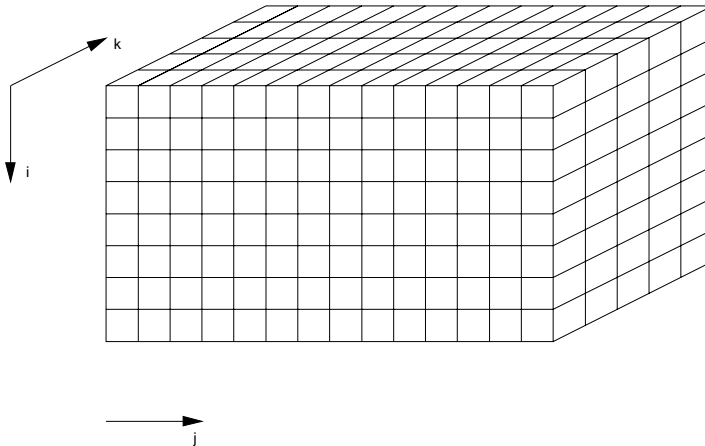


Figura 8.4 Array tridimensional

Ejemplo 8.14

Un *array tetradimensional* tendría cuatro índices para indicar la posición de cada elemento.

```
a [i, j, k, l]
b [2, 3, 4, 7]
```

Un *array pentadimensional* tendría cinco índices para indicar la posición de cada elemento:

```
c [i, j, k, l, m]
d [2, 3, 7, 1, 9]
```

Es decir un *array n-dimensional* tendría n índices para indicar la posición de cada elemento:

```
e [a1, a2, a3, ..., an]
```

Para manejar *arrays multidimensionales* se suelen utilizar bucles *FOR* anidados. Para leer, escribir, o recorrer ordenadamente todos los elementos de un *array n-dimensional*, se suelen anidar n bucles *FOR*, como veremos en el siguiente ejemplo.

Ejemplo 8.15

Escribir un programa que calcule la facturación media de una empresa para un determinado mes, si la empresa en cuestión fabrica 5 artículos y almacena los datos de venta por años, meses y artículos.

ESTRUCTURA DE DATOS ARRAY

Solución. El problema se reduce a leer los datos y calcular la media para el mes pedido, análogamente a los cálculos del ejemplo 8.11. Utilizaremos tipos subíndices con significado semántico para facilitar la referencia a las celdas del *array tridimensional* (ver declaraciones).

Algoritmo

```
INICIO
  DESDE i := añoInicial HASTA añoFinal HACER (recorre años)
    DESDE j:=1 HASTA 12 HACER (recorre meses)
      DESDE k:='A' HASTA 'E' HACER (recorre artículos)
        Leer a[i,j,k];
      FIN_DESDE k;
    FIN_DESDE j;
  FIN_DESDE i;
  Leer mes en estudio;
  suma := 0;
  DESDE i := añoInicial HASTA añoFinal HACER
    DESDE k:='A' HASTA 'E' HACER
      suma := suma + a[i,j,k];
    FIN_DESDE k;
  FIN_DESDE i;
  suma := suma / n° de años
  Escribir suma
FIN
```

Codificación en Pascal

```
PROGRAM Tridimensional (input, output);
TYPE
  filas = 1978..1988;
  columnas = 1..12;
  fondo = 'A'..'E';
  matriz = ARRAY [filas,columnas,fondo] OF real;
VAR
  i: filas;
  j: columnas;
  k: fondo;
  s: real;
  a: matriz;
BEGIN
  (* Lectura de la matriz *)

  FOR i := 1978 TO 1988 DO
    FOR j := 1 TO 12 DO
      FOR k := 'A' TO 'E' DO
        BEGIN
          Write ('Dame la facturación del artículo ');
          Write (k:1, ' del mes ',j:2, ' del año ',i:4, ': ');
          Readln (a[i,j,k])
        END;
      END;
    END;
  END;

  (* Cálculo de la facturación media de un determinado mes *)

  Write ('Dame el mes del que se desea la facturación media: ');
  Readln (j);
  s := 0.0;
  FOR i := 1978 TO 1988 DO
    FOR k := 'A' TO 'E' DO
      s:= s + a[i,j,k];
    END;
  END;
```

ARRAYS EMPAQUETADOS

```
Writeln ('La facturación media del mes ',i:2,' es ',s/8:7:1)
END.
```

8.5 ARRAYS EMPAQUETADOS

Con el fin de lograr un uso más eficiente de la memoria algunos *arrays* pueden definirse con la opción *PACKED* (empaquetamiento), con lo cual en una determinada cantidad de memoria se almacena más información.

Esto realmente depende del tipo de ordenador, pues por ejemplo en la mayoría de los microcomputadores solo cabe un carácter en una posición de memoria, con lo que el empaquetamiento no ahorra memoria. Donde sí es realmente efectivo es en grandes ordenadores, en los cuales cada palabra de memoria puede albergar cuatro o más caracteres (figura 8.5).

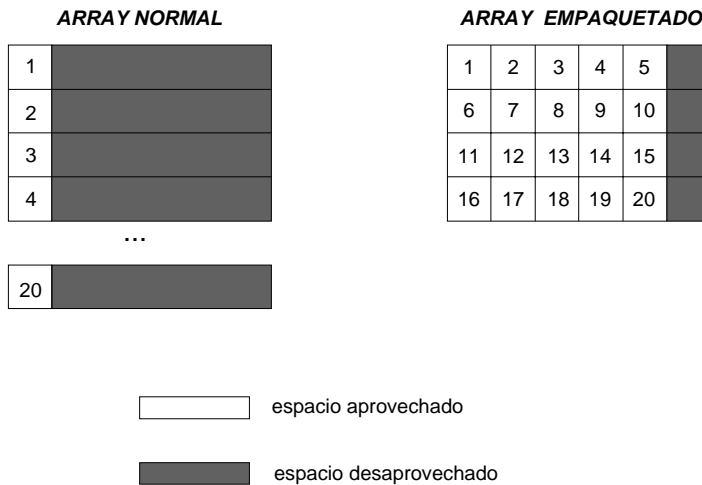


Figura 8.5 Ahorro de memoria con un array empaquetado

La forma de utilizar la opción de empaquetamiento de *arrays* en Pascal consiste en anteponer la palabra reservada *PACKED* en la definición del *array*. Por ejemplo:

```
TYPE empaquetado = PACKED ARRAY [1..20] OF char;
```

El empaquetamiento es realmente efectivo con *arrays* de tipo *char*, booleano, enumerado o con datos de tipo subrango.

Si bien es cierto que el empaquetamiento ahorra memoria, hay sin embargo un compromiso entre espacio ocupado y velocidad de proceso, ya que las operaciones de empaquetamiento y desempaquetamiento implicadas requieren un tiempo adicional. Así pues no está claro que método es el más ventajoso; y la elección de una u otra opción dependerá del tipo de aplicación.

En algunas circunstancias especiales, como por ejemplo en programas que contienen gran número de asignaciones de *arrays* completos, o en los que se transfieren *arrays* como parámetros por valor a procedimientos o funciones, el empaquetamiento es beneficioso en ambos aspectos: espacio y velocidad.

Los *arrays* empaquetados tienen una restricción, y es que sus elementos no pueden ser pasados como parámetros actuales (argumentos) a un procedimiento o función. Sin embargo todo el *array* completo sí puede pasarse de la forma habitual.

Por ejemplo, si *a* es un *array* empaquetado de caracteres no podemos hacer:

```
Read(a[i]);
```

sino:

```
Read(ch);
a[i] := ch;
```

siendo *ch* una variable de tipo *char*. Nótese que *Read* es un procedimiento incorporado por el lenguaje Pascal estándar¹⁸.

8.6 CADENAS DE CARACTERES

Una *cadena de caracteres* es un *array* empaquetado unidimensional de caracteres. La importancia y peculiaridades de las cadenas justifican su tratamiento separado en este epígrafe una vez presentado el tipo *array*.

La importancia de las cadenas reside en que pueden almacenar palabras o frases enteras, siendo éstas interpretadas como elementos simples desde el punto de vista de la programación.

En cuanto a las peculiaridades de las cadenas, el Pascal permite realizar ciertas operaciones que no están permitidas para otros tipos de *arrays*.

El concepto de cadena se extiende tanto a variables como a constantes. Así la siguiente declaración:

```
CONST titulo = 'La Eneida';
```

hace que el identificador *titulo* represente una *constante cadena* de longitud 9. Y la declaración:

```
VAR nombre: PACKED ARRAY [1..10] OF char;
```

¹⁸ El compilador Turbo Pascal admite el uso de la palabra reservada *PACKED*, pero no tiene ningún efecto.

CADENAS DE CARACTERES

hace que `nombre` sea una *variable cadena* capaz de almacenar palabras de hasta 10 caracteres. No se puede almacenar más caracteres en una cadena de los que permita su longitud. Si el valor que hay que almacenar es menor que la longitud de la cadena, el resto se rellenará por el programador con blancos, en caso contrario puede contener *basura* (caracteres correspondientes a los valores en binario de dichas posiciones de memoria dejados por programas que se ejecutaron anteriormente).

Operaciones con cadenas

Al igual que los *arrays* sin empaquetar, las cadenas del mismo tipo pueden asignarse unas a otras con una sola sentencia:

```
TYPE
  cadena = PACKED ARRAY [1..8] OF char;
VAR
  nombre, palabra: cadena;
```

las siguientes asignaciones son válidas:

```
nombre := 'PEPE   ' ; (* obsérvese el relleno de blancos *)
palabra := nombre;
```

sin embargo éstas no son válidas:

```
nombre := 'PEPE';      (* menos de 8 caracteres *)
nombre := 'INMACULADA'; (* más de 8 caracteres *)
```

Las cadenas de igual longitud pueden compararse entre sí mediante los operadores relacionales. Las cadenas se comparan según el orden de los caracteres en el código utilizado (ASCII generalmente), comparándose los caracteres de ambas cadenas uno a uno de izquierda a derecha. La tabla con el código ASCII se muestra en el anexo I.

Por ejemplo si hacemos las asignaciones:

```
nombre := 'Fernando';
palabra := 'Feroz   ';
```

las siguientes expresiones son todas *true*:

```
nombre < palabra      (* pues 'n' < 'o' *)
palabra <> 'Ventanal'
palabra = 'Feroz   '
' Cadena' >= 'Anterior' (* pues 'C' > 'A' *)
```

Otra peculiaridad asociada a las cadenas es que pueden escribirse directamente mediante una sentencia *Write* o *Writeln*. Así por ejemplo:

```
Writeln(nombre);
```

escribiría:

```
Fernando
```

Sin embargo no se puede decir lo mismo respecto a la lectura mediante *Read* y *Readln*. En este caso se deberá utilizar un bucle y proceder como se indica a continuación:

```
FOR i:= 1 TO 8 DO
BEGIN
  Read (ch);
  nombre[i] := ch;
END;
```

Algunos compiladores de Pascal (por ejemplo Turbo Pascal) permiten hacer *Read(nombre[i])* directamente.

Ejemplo 8.16

Construir una función que dada una cadena de caracteres de longitud dada, la explore comenzando por el final, y se detenga al encontrar el primer carácter distinto de blanco, dando la longitud de la cadena hasta dicho carácter.

Solución. Utilizaremos una estructura *WHILE*, con doble condición de parada: encontrar un carácter distinto de blanco, o llegar al principio de la cadena.

Algoritmo

```
Función LonCadena
NECESITA: cadena, a y longitud, lonMax
DEVUELVE: longitud real de la cadena
INICIO
  i := lonMax + 1;
  buscar := verdad;
  MIENTRAS (i>1) Y (buscar) HACER
    i := i-1;
    buscar := (a[i]=blanco);
  FIN_MIENTRAS;
  LonCadena := i;
FIN
```

Codificación en Pascal

```
FUNCTION LonCadena(a: cadena; lonMax:integer): integer;
(* El tipo cadena se ha definido previamente, como un array empaquetado
de tamaño lonMax de caracteres *)
CONST
  blanco = ' ';
VAR
  i: integer;
  buscar: boolean;
BEGIN
  i := lonMax + 1;
  buscar := true;
  WHILE (i>1) AND buscar DO
    BEGIN
      i := i-1;
      buscar := (a[i]=blanco);
    
```

CADENAS DE CARACTERES

```
END;  
lonCadena := i;  
END;
```

Para ilustrar el uso de la función anterior supongamos las siguientes sentencias en el programa principal:

```
b := 'LOS CABAL '  
c := lonCadena(b, lonMax);
```

Trás estas sentencias la variable *c* valdrá 9.

Ejemplo 8.17

Construir una función que dadas dos cadenas, de forma que una sea parte de otra, nos indique la posición donde se comienza a encontrar la subcadena.

Solución. Utilizaremos una estructura *WHILE*, también con doble condición de parada: encontrar la subcadena, o llegar a una posición en la que ya no quepa la subcadena en el resto de cadena sin explorar (longitud de la cadena - longitud de la subcadena + 1). Esta vez recorreremos la cadena del principio al final.

Algoritmo

NIVEL 0:

Función Busca

```
NECESITA: cadenas a y subA, lonMax  
UTILIZA: Función LonCadena  
DEVUELVE: posición de subA en a (0 si no se encuentra)  
INICIO  
  i := 1;  
  hallado := falso;  
  lt := LonCadena(a,lonMax)  
  lm := LonCadena(subA,lonMax)  
  MIENTRAS ( i < (lt-lm+1) ) Y NO(hallado) HACER  
    SI a[i] = subA[1] (encontramos primer caracter  
                      de subA)  
      ENTONCES  
        Inicializar j:=2; (recorre subA)  
                    k:=i+1; (recorre a)  
                    buscar := verdad  
        MIENTRAS (j ≤ lm) Y buscar HACER  
          SI (a[k] = subA[j])  
            ENTONCES  
              j := j+1;  
              k := k+1  
            SI_NO buscar := falso;  
          FIN_SI;  
        FIN_MIENTRAS;  
      FIN_SI;
```

ESTRUCTURA DE DATOS ARRAY

```
        i:=i+1;
    FIN_MIENTRAS;
    SI hallado
        ENTONCES Busca := i-1 (porque se incrementa
                               al final del bucle)
        SI_NO Busca := 0;
    FIN_SI;
FIN
NIVEL 1:
```

El algoritmo de LonCadena se detalla en el ejemplo 8.16.

Codificación en Pascal

```
FUNCTION Busca (a, subA: cadena ; lonMax: integer ): integer;
VAR
    i, j, k, lt, lm : integer;
    buscar, hallado : boolean;
BEGIN
    i:=1;
    hallado := false;
    lt:= LonCadena (a, lonMax ); (* Ver ejemplo anterior *)
    lm:= LonCadena (subA, lonMax);
    WHILE (i<=lt-lm +1) AND (NOT hallado) DO
        BEGIN
            IF a[i] = subA[1]
            THEN
                BEGIN
                    j:=2 ;
                    k:=i+1;
                    buscar := true;
                    WHILE (j <= lm) AND buscar DO
                        IF a[k] = subA[j]
                        THEN
                            BEGIN
                                j := j+1;
                                k := k+1
                            END
                        ELSE buscar := false ;
                    hallado := buscar;
                END;
            i := i+1;
        END;
        IF hallado
        THEN Busca:= i-1
        ELSE Busca:= 0;
    END; (* Busca *)
```

La aplicación de la función puede hacerse de la siguiente manera:

```
...
a := 'LOS CABALE';
b := 'CAB      ';
c := 'AA      ';
d := Busca(a, b, 10);
Writeln('d = ',d);
e := Busca(a, c, 10);
Writeln('e = ',e);
...
```


CADENAS DE CARACTERES

Este fragmento de programa escribirá:

```
d = 5
e = 0
```

Ejemplo 8.18

Realizar un procedimiento que una dos cadenas dando como resultado otra cadena, que si excede de la longitud máxima se trunca.

Solución. Utilizaremos un procedimiento, ya que el resultado de una función no puede ser de tipo estructurado. Llamaremos:

```
n1 = longitud de la cadena 1 (cade1)
n2 = longitud de la cadena 2 (cade2)
m = longitud de la cadena resultante (destino); m será n1+n2 O lonMax.
```

Usaremos dos bucles *FOR* para copiar los elementos de cada cadena en la resultante, el primero desde 1 hasta n_1 , y el segundo desde n_1+1 hasta m .

Algoritmo

NIVEL 0:

```
ACCION Concatena ES:
  NECESITA: cade1, cade2, lonMax
  UTILIZA: Función LonCadena
  DEVUELVE: cadena resultante, destino
  INICIO
    n1 := Loncadena(cade1, lonMax);
    n2 := Loncadena(cade2, lonMax);
    m := n1 + n2;
    SI m > lonMax ENTONCES m := lonMax;
    FIN_SI;
    DESDE i:=1 HASTA n+1 HACER
      destino[i] := cade1[i];    (copiamos cade1)
    FIN_DESDE;
    j := 1;    (j recorrerá cade2)
    DESDE i:=n1+1 HASTA m HACER
      destino[i] := cade2[j];    (copiamos cade2)
      j:=j+1;
    FIN_DESDE;
  FIN;
```

NIVEL 1:

El algoritmo de LonCadena se detalla en el ejemplo 8.16.

Codificación en Pascal

```

PROCEDURE Concatena (cade1, cade2: cadena; lonMax: integer;
                    VAR destino: cadena);
VAR
  i, j, n1, n2, m: integer;
BEGIN
  n1 := LonCadena ( cade1, lonMax);
  n2 := LonCadena ( cade2, lonmax);
  m := n1 + n2;
  IF m>lonMax THEN m:= lonMax;
  FOR i:=1 TO n1 DO
    destino [i]:= cade1 [i];
  j:= 1;
  FOR i:= n1+1 TO m DO
    BEGIN
      destino [i] := cade2 [j];
      j:=j+1
    END;
END; (* Concatena *)

```

Suponiendo que se ejecutan las sentencias:

```

...
a := 'AEIOU   ' ;
b := 'VOCAL   ' ;
Concatena(a,b,10,c);
Writeln('c = ',c);
Concatena(a,a,10,c);
Writeln('c = ',c);
...

```

Se escribirá:

```

c = AEIOUVOCAL
c = AEIOUAEIOU

```

8.7 EXTENSION STRING

Aunque el Pascal estándar no incorpora el tipo predefinido *STRING*, la mayoría de las implementaciones prácticas sí lo incorporan y puede considerarse casi como estándar.

Un *string* puede definirse como una cadena de caracteres de longitud variable. Es decir: una cadena de caracteres que puede contener desde cero (cadena vacía) hasta un determinado número máximo de caracteres.

La sintaxis más generalizada para el tipo *STRING* se muestra a continuación:

```
string [n]
```

donde *n* es un número entero positivo que indica la longitud máxima que puede tener el *string*.

Así, las variables:

```

VAR
  nombre: string [20];
  linea : string [80];

```

EXTENSION STRING

son dos *strings* de longitudes máximas respectivas 20 y 80.

La sintaxis que aquí presentamos es la más aceptada, si bien pueden existir variaciones de unos compiladores a otros. Por ejemplo algunos utilizan paréntesis en vez de corchetes para especificar la longitud máxima. En otros, como por ejemplo en el compilador VAX-Pascal, las diferencias son más acusadas, ya que este tipo de datos se define como `VARYING [n] OF char;`.

Como vemos, al definir un tipo *string* únicamente se especifica la longitud máxima que puede alcanzar. Se supone que los índices van desde 1 hasta el valor especificado como longitud máxima. En Turbo Pascal si no se especifica el valor máximo toma 255 por defecto.

Así, `nombre[1]` contendrá el primer carácter del *string* `nombre`.

Las principales diferencias con las *cadena de caracteres* (`PACKED ARRAY [1..m] OF char`), que acabamos de estudiar, son:

- A una variable de tipo *string* se le puede asignar una cadena de caracteres de cualquier longitud, siempre que no exceda la máxima especificada. En el caso de que fuera de mayor longitud, los caracteres sobrantes se perderían.
- El tipo *string* en Turbo Pascal tiene como valor máximo 255, sin embargo las *cadena de caracteres* su valor máximo sólo depende de la memoria disponible por el ordenador en tiempo de compilación. En el capítulo 12 se estudiarán otro tipo de cadenas (tipo *PChar*) que incorpora Turbo Pascal, cuyas características más sobresalientes son: a) que son dinámicas, es decir su tamaño se puede definir en tiempo de ejecución; y b) su tamaño máximo tan sólo depende de la memoria disponible en tiempo de ejecución.

Ejemplo 8.19

Con la sentencia

```
nombre := '';
```

Asignamos a `nombre` la cadena vacía. Su longitud es cero.

Con la sentencia

```
nombre := 'Pepe';
```

La longitud actual del *string* `nombre` es 4.

Con la asignación:

```
linea := nombre;
```

Asignamos a `linea` el valor de `nombre`. Observe que esto es válido aunque ambos *strings* son de diferente longitud máxima.

- Una variable de tipo *string* puede leerse con una sola sentencia *Read* o *Readln*. Podemos leer con una sola sentencia el *string* nombre:

```
Readln (nombre);
```

- Los *strings* se pueden comparar entre sí, y con cadenas de caracteres (*PACKED ARRAY*). Dos *strings* serán iguales si su longitud *actual* es la misma y su contenido es el mismo.

Las diferentes implementaciones del Pascal incluyen gran cantidad de procedimientos y funciones incorporadas para el manejo de *strings*, que permiten realizar operaciones tales como concatenación, copia, búsqueda, etc... Remitimos al lector al manual correspondiente de su implementación del Pascal. En el caso del compilador Turbo Pascal las características propias de su tipo *string* se indican en el apartado 8.11 de este capítulo, donde está el epígrafe *El tipo string en Turbo Pascal*.

8.8 CONCEPTO DE TIPO ABSTRACTO DE DATOS (TAD)

En la sección 3.2, *Los Datos y sus tipos*, del capítulo 3, introducimos el concepto de *Tipo Abstracto de Datos (TAD)*, como un *modelo matemático* con una serie de *operaciones* definidas en ese modelo. Ahora estamos en condiciones de profundizar en este concepto, pues sabemos lo que es una *estructura de datos*, y sabemos construir subprogramas para operar con dicha estructura de datos.

Un *Tipo Abstracto de Datos* o *TAD*, es una declaración de datos (generalmente una estructura de datos), *encapsulada* (integrada en una unidad) con todas las operaciones necesarias para manipular ese tipo de datos, de manera que los datos y las operaciones se puedan utilizar sin conocer en detalle como han sido construidos.

La *encapsulación* no consiste en la mera unión de los datos y las operaciones asociadas a ellos. Además, el acceso a los datos se realiza a través de las operaciones suministradas en el TAD. No se manipulan directamente los datos, como hemos hecho hasta ahora, sino que se construyen subprogramas para cada operación necesaria: Inicializar, efectuar cálculos, ordenar, modificar, mostrar, etc.

Otra característica mencionada en la definición de TAD expuesta en el capítulo 3 es la *ocultación de información*. Consiste en no mostrar a los programadores usuarios del TAD los detalles de su instrumentación, sino únicamente lo que necesitan para utilizar el TAD. Se dice que un tipo de datos es *opaco* si es utilizable sin saber como están almacenados los datos en el hardware. Una operación es *opaca* si podemos utilizarla sin saber como funciona internamente. Por otra parte las operaciones deben tener una sintaxis sencilla, para ser usadas con facilidad.

CONCEPTO DE TIPO ABSTRACTO DE DATOS (TAD)

¿Cuales son las ventajas de la utilización de TADs? Podemos citar la extensibilidad, y la facilidad de reutilización y modificación del código. Actualmente, los ordenadores se utilizan prácticamente para todo, resultando imposible enumerar todas sus aplicaciones. Permiten describir la solución del problema, en la misma nomenclatura del problema a resolver, acercando el mundo real al código del programa que lo simula. Esto exige una enorme variedad de tipos de datos y operaciones asociadas, que ningún lenguaje de programación puede suministrar listos para ser utilizados. Los buenos lenguajes de programación lo que suministran son herramientas para que el programador pueda construirse sin dificultad sus tipos de datos y las correspondientes operaciones asociadas, es decir, sus TADs. Los TADs son *extensiones* del lenguaje de programación, que facilitan su adaptación a nuevas aplicaciones.

Por poner un ejemplo que nos resulte familiar, podríamos considerar como TAD el tipo *string* de Turbo Pascal con los ocho subprogramas que suministra este compilador para las operaciones básicas a realizar con cadenas: concatenar dos cadenas; calcular la longitud de una cadena; extraer una subcadena de una cadena; insertar una cadena en una posición dada dentro de otra cadena; borrar una parte de la cadena; calcular la posición del primer carácter de una subcadena dentro de una cadena; convertir valores numéricos en cadenas y viceversa. Junto con estas ocho operaciones tenemos que incluir los procedimientos standard *Read*, *Readln*, *Write* y *Writeln*, que nos permiten leer y escribir cadenas de tipo *string*.

En la programación convencional (procedimental), las estructuras de datos y los subprogramas para su manipulación se definen por separado. El tipo *string* y sus operaciones asociadas no se ajustan exactamente a la definición formal que hemos dado de TAD. La *encapsulación* es incompleta, ya que podemos acceder directamente a un *string* y sus elementos, además de hacerlo a través de los subprogramas mencionados. Por otro lado, existen operaciones básicas de manipulación de cadenas no suministradas por el lenguaje: pasar una cadena a mayúsculas o minúsculas, comparar cadenas, búsqueda y sustitución, ...

En Turbo Pascal se pueden construir TADs mediante *unidades de compilación*, o *units* (*TPU*). Esto no quiere decir que todas las *units* representen TADs. Como estudiamos en el capítulo 7, sección 7.11, una *unit* es un módulo que engloba definiciones y declaraciones de constantes, tipos y variables, junto con procedimientos y funciones que operan con ellos. Estos módulos no se pueden ejecutar solos, pero pueden compilarse y distribuirse a programadores clientes, que los utilizarán dentro de sus programas, sin necesidad de conocer los detalles de su construcción (*ocultación de información*).

Recordando otra vez el capítulo anterior, las *units* de Turbo Pascal constan de tres partes:

- *Parte de Interfaz*: Formada por la parte del TAD accesible al programador usuario, es decir, *pública*. Contiene definiciones y declaraciones de constantes, tipos y variables, además de cabeceras de procedimientos y funciones (operaciones del TAD). Todos los identificadores de esta parte son accesibles al programa que utiliza la *unit* (mediante una cláusula *uses*).

- *Parte de Implementación:* Es la parte *privada* (oculta) del TAD. Contiene la implementación de los procedimientos y funciones declarados en la parte de interfaz, y además declaraciones privadas, no accesibles al programa que utiliza la *unit*. Estas declaraciones no se limitan solo a etiquetas, constantes, tipos y variables, también pueden existir procedimientos y funciones solo utilizables dentro del mismo TAD.
- *Parte de Inicialización:* Es opcional, y consta de una serie de sentencias para inicializar el TAD, que se ejecutan una sola vez, al ser llamada la *unit* por el programa cliente mediante la sentencia *uses*.

En el capítulo 13, *Programación orientada a objetos*, veremos otro enfoque de los TADs. Adelantaremos nada más que los datos y los subprogramas (llamados *métodos*), se engloban en un tipo especial denominado *objeto* (object), con una parte *pública*, accesible a programadores usuarios, y otra parte *privada*, no accesible, con la implementación de los métodos. En las secciones 13.3 y 13.4 profundizaremos en los conceptos de *encapsulamiento* y *ocultación de información*.

8.9 LOS ARRAYS COMO TADs

Para familiarizarnos con los nuevos conceptos que acabamos de introducir, vamos a construir un TAD para manejar vectores de números reales, que incluya las definiciones y declaraciones necesarias para construir la estructura de datos, y algunas operaciones básicas cuyos algoritmos nos sean conocidos.

Ejemplo 8.20

Construir mediante una *unit* un *TAD* para manipular vectores de números reales, que incluya subprogramas para leer los elementos de un vector, escribirlos, ordenarlos en ambos sentidos, y realizar los siguientes cálculos: suma, media, desviación típica, elemento máximo y elemento mínimo.

Solución. Construiremos una *unit* que llamaremos `TadArray`. En la parte de interface incluiremos las declaraciones necesarias para construir la estructura (que son las utilizadas en los ejemplos 8.3 hasta 8.9) y las cabeceras de los procedimientos y funciones que realizarán las operaciones pedidas. En la parte de implementación escribiremos el código de dichos subprogramas. Con fines didácticos repetimos en esta parte las cabeceras completas de los subprogramas, aunque sea redundante. Las fases de análisis y diseño de algoritmos para estas operaciones se incluye en los ejemplos 8.3, 8.5, 8.7, y 8.8. El código del algoritmo de ordenación se ha mejorado, permitiendo elegir entre ordenación ascendente o descendente, así como ordenar solo un intervalo del vector.

LOS ARRAYS COMO TADs

Codificación en Pascal

```
Unit TADarray;

INTERFACE
CONST
  m = 100;
TYPE
  indice = 1..m;
  vector = ARRAY[indice] OF real;
VAR
  i,j: indice;
(*-----*)
PROCEDURE LeerVector(VAR w: vector; VAR m:indice);
PROCEDURE EscribeVector (w: vector; m:indice);
FUNCTION Suma(VAR v: vector; num: indice): real;
FUNCTION Media(VAR v: vector; num: indice): real;
FUNCTION Sigma(VAR v: vector; num: indice): real;
FUNCTION maximo(VAR v: vector; num: indice): real;
FUNCTION minimo(VAR v: vector; num: indice): real;
PROCEDURE OrdenacionBurbuja (VAR valores: vector;
                             limInf,limSup: indice;
                             desAscendente: boolean);
(*-----*)
IMPLEMENTATION
PROCEDURE LeerVector(VAR w: vector; VAR m:indice);
BEGIN
  Write('¿Nº de elementos?');
  Readln(m);
  FOR i:=1 TO m DO
    BEGIN
      Write ('Elemento nº ',i,': ');
      Readln(w[i]);
    END;
END;
(*-----*)
PROCEDURE EscribeVector(w: vector; m:indice);
BEGIN
  Writeln;
  Writeln('Elementos del vector:');
  Writeln('      N°           ELEMENTO');
  Writeln('-----');
  FOR i:=1 TO m DO writeln (i:8,w[i]:15:3);
  Writeln;
  Writeln ('Pulsa <intro> para continuar');
  Writeln;
  readln;
END;
(*-----*)
FUNCTION Suma(VAR v: vector; num: indice): real;
VAR
  s: real;
BEGIN
  s := 0;
  FOR i:=1 TO num DO
    s := s + v [i];
  Suma := s;
END;
(*-----*)
FUNCTION Media(VAR v: vector; num: indice): real;
BEGIN
  Media := Suma(v, num) / num;
END;
(*-----*)
```

ESTRUCTURA DE DATOS ARRAY

```

FUNCTION Sigma(VAR v: vector; num: indice): real;
VAR
    sumatorio, med: real;
BEGIN
    sumatorio := 0;
    med := Media(v, num);
    FOR i:=1 TO num DO
        sumatorio := sumatorio + Sqr(v[i] - med);
    Sigma := Sqrt(sumatorio)/num;
END;
(*-----*)
FUNCTION minimo(VAR v: vector; num: indice): real;
VAR
    min: real;
BEGIN
    min := 1E+38; (* Inicializamos el mínimo a un valor mayor que todos
                  los que pueda contener el vector *)
    FOR i:=1 TO num DO
        IF v[i] < min THEN min := v[i];
    minimo := min;
END;
(*-----*)
FUNCTION maximo(VAR v: vector; num: indice): real;
VAR
    max: real;
BEGIN
    max := -1E+38; (* Inicializamos el máximo a un valor menor que todos
                   los que pueda contener el vector *)
    FOR i:=1 TO num DO
        IF v[i] > max THEN max := v[i];
    maximo := max;
END;
(*-----*)
PROCEDURE OrdenacionBurbuja (VAR valores: vector;
                             limInf,limSup: indice;
                             desAscendente: boolean);
(* Este procedimiento ordena el vector valores mediante el
   algoritmo de la burbuja. Valores es el vector de entrada/salida;
   limInf y limSup son los índices entre los cuales se ordena;
   desAscendente es true si se desea orden ascendente *)
(*-----*)
PROCEDURE Intercambia (VAR a, b : real);
(* Intercambia los valores de las variables a y b *)
VAR
    temp : real; (* almacenamiento temporal *)
BEGIN (* Intercambia *)
    temp:=a;
    a:=b;
    b:=temp;
END; (* Intercambia *)
(*-----*)
BEGIN (* OrdenacionBurbuja *)
    FOR j:=limSup DOWNTO limInf+1 DO
        FOR i:=limInf TO j-1 DO
            IF ((valores[i] > valores [j]) AND (desAscendente))
                THEN Intercambia (valores[i], valores[j])
            ELSE
                IF (valores [i] < valores [j]) AND (NOT desAscendente)
                    THEN Intercambia (valores[i], valores[j]);
        END; (* OrdenacionBurbuja *)
    END.

```

Veamos como utilizar la *unit* `TadArray` dentro de un programa *cliente*:

APLICACIONES AL CALCULO NUMERICO

```
PROGRAM VectorReales(input, output);

(* Programa ejemplo de utilización del TAD array de reales *)

Uses TADArray;
VAR x: vector;
    n: indice;
    ch:char;
    menorAmayor:boolean;
BEGIN
  LeerVector(x,n);
  EscribirVector(x,n);
  Writeln('Cálculos realizados con los elementos del vector:');
  Writeln('Suma =           ', Suma(x,n):8:2);
  Writeln('Valor medio =       ', Media(x,n):8:2);
  Writeln('Desviación típica =   ', Sigma(x,n):8:2);
  Writeln('Valor máximo =       ', Maximo(x,n):8:2);
  Writeln('Valor mínimo =       ', Minimo(x,n):8:2);
  Write('¿Desea ordenación ascendente o descendente (a/d)? ');
  Readln(ch);
  IF ch IN ['a','A']
    THEN OrdenacionBurbuja(x, 1, n, true)
    ELSE
      IF ch IN ['d','D']
        THEN OrdenacionBurbuja(x, 1, n, false)
        ELSE Writeln('ORDENACION NO REALIZADA');
  EscribirVector(x,n);
END.
```

Obsérvese que este programa ni siquiera necesita una declaración de la estructura de datos `vector` para funcionar, ya que dicha declaración está incluida en la parte pública (*interface*) de la *unit*. El programa no accede a los elementos del vector `x` directamente, sino a través de los subprogramas declarados en la parte de *interface*.

8.10 APLICACIONES AL CALCULO NUMERICO

RESOLUCION DE SISTEMAS DE ECUACIONES LINEALES POR EL METODO DE GAUSS-JORDAN

Para utilizar este método escribiremos el sistema a resolver en notación matricial. El método se basa en la siguiente propiedad de los sistemas de ecuaciones lineales: Si sustituimos una ecuación de un sistema por una combinación lineal de ella y otra, el sistema resultante es equivalente al original. Aprovechando esta propiedad, se van haciendo ceros por columnas hasta diagonalizar la matriz de los coeficientes. En este momento, la matriz solución es el vector de términos independientes. Para entenderlo, desarrollemos paso a paso la resolución de un sistema de tres ecuaciones con tres incógnitas.

Sea el sistema compatible determinado:

$$\begin{aligned} 2x_1 + 4x_2 - 5x_3 &= -5 \\ x_1 + x_2 + x_3 &= 6 \\ -x_1 + 2x_2 - 2x_3 &= -3 \end{aligned}$$

En notación matricial tenemos:

ESTRUCTURA DE DATOS ARRAY

$$\begin{vmatrix} 2 & 4 & -5 \\ 1 & 1 & 1 \\ -1 & 2 & -2 \end{vmatrix} \mathbf{x} \begin{vmatrix} x_1 \\ x_2 \\ x_3 \end{vmatrix} = \begin{vmatrix} -5 \\ 6 \\ -3 \end{vmatrix}$$

Que también se puede escribir así:

$$(A) \times (X) = (B)$$

donde (A) es la matriz de los coeficientes, (X) es el vector de las incógnitas, y (B) es el vector de los términos independientes.

Otra notación es:

$$\sum a_{ij} \cdot x_j = b_j$$

Comenzamos por formar la *matriz aumentada*, añadiendo a la matriz de los coeficientes una columna con el vector de los términos independientes:

$$\begin{vmatrix} 2 & 4 & -5 & -5 \\ 1 & 1 & 1 & 6 \\ -1 & 2 & -2 & -3 \end{vmatrix}$$

Paso 1

- Normalizamos la primera fila dividiendo todos sus elementos por el *pivote*, a_{11} . En cada fila el *pivote* es el elemento de la diagonal principal. Para *normalizar* (hacer unos en la diagonal principal), habrá que dividir cada fila por el pivote correspondiente.

$$\begin{vmatrix} 1 & 2 & -5/2 & -5/2 \\ 1 & 1 & 1 & 6 \\ -1 & 2 & -2 & -3 \end{vmatrix}$$

- Hacemos ceros en la primera columna, sumando a cada fila (excepto la del pivote) la primera multiplicada por un factor. Para cada fila hay que elegir este factor de manera que se anule el elemento de la columna del pivote.

$$\begin{vmatrix} 1 & 2 & -5/2 & -5/2 \\ 0 & -1 & 7/2 & 17/2 \\ 0 & 4 & -9/2 & -11/2 \end{vmatrix} \begin{array}{l} \leftarrow \text{(fila 2)} + (-1)(\text{fila 1}) \\ \leftarrow \text{(fila 3)} + 1(\text{fila 1}) \end{array}$$

Paso 2

Repetimos el proceso desde la segunda fila, tomando como pivote $a_{22} = -1$

- Dividimos por el pivote la segunda fila, para que quede $a_{22} = 1$

$$\begin{vmatrix} 1 & 2 & -5/2 & -5/2 \\ 0 & 1 & -7/2 & -17/2 \\ 0 & 4 & -9/2 & -11/2 \end{vmatrix}$$

APLICACIONES AL CALCULO NUMERICO

- Hacemos ceros en la segunda columna:

$$\left| \begin{array}{ccc|c} 1 & 0 & 9/2 & 29/2 \\ 0 & 1 & -7/2 & -17/2 \\ 0 & 0 & 19/2 & 57/2 \end{array} \right| \begin{array}{l} \leftarrow (\text{fila } 1) + (-2)(\text{fila } 2) \\ \leftarrow (\text{fila } 3) + (-4)(\text{fila } 2) \end{array}$$

Paso 3

- El último pivote es a_{33} . Normalizamos la tercera fila dividiendo por el pivote:

$$\left| \begin{array}{ccc|c} 1 & 0 & 9/2 & 29/2 \\ 0 & 1 & -7/2 & -17/2 \\ 0 & 0 & 1 & 3 \end{array} \right|$$

- Hacemos ceros en la tercera columna:

$$\left| \begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array} \right| \begin{array}{l} \leftarrow (\text{fila } 1) + (-9/2)(\text{fila } 3) \\ \leftarrow (\text{fila } 2) + (7/2)(\text{fila } 3) \end{array}$$

Hemos transformado el sistema de ecuaciones original en otro equivalente, cuya matriz de coeficientes es la matriz unidad:

$$\left| \begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array} \right| \times \begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array} = \begin{array}{c} 1 \\ 2 \\ 3 \end{array}$$

Es decir:

$$\begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array} = \begin{array}{c} 1 \\ 2 \\ 3 \end{array} \rightarrow \begin{array}{l} x_1 = 1 \\ x_2 = 2 \\ x_3 = 3 \end{array}$$

Este método es laborioso para resolver manualmente sistemas de tres o más ecuaciones, pero es muy sistemático, por lo que resulta fácil elaborar, a partir de él, un algoritmo que funcione con sistemas compatibles determinados de n ecuaciones con n incógnitas. Se puede observar que son necesarios n pasos, uno por cada fila o ecuación. Para cada paso se repite la siguiente secuencia de acciones:

- Normalizar (obtener un 1 en la diagonal principal) la fila actual, dividiendo todos sus elementos por el pivote, a_{ii} .
- Hacer ceros en la columna del pivote. Se logra recorriendo todas las filas (excepto la del pivote), y sumando a cada fila la fila del pivote multiplicada por un factor elegido de tal manera que se anule el elemento de la columna del pivote.

Antes de escribir el algoritmo resultante, hay que examinar los posibles problemas que pueden surgir *¿Son válidas todas las operaciones a realizar?* Se trata de simples operaciones aritméticas. La única operación que puede fallar es la normalización de las filas, en caso de que algún pivote sea nulo. Hay varias maneras de solucionar esta dificultad. Si un pivote es cero o muy pequeño, basta intercambiar la fila correspondiente con otra inferior. Hay que tener en cuenta que los ceros (o números muy pequeños) pueden aparecer en la diagonal debido a las operaciones aritméticas que se van realizando con las filas. Un método de evitar divisiones por cero es, en cada paso, reorganizar las filas de manera que el coeficiente de mayor magnitud quede en la diagonal (de pivote). Si el sistema tiene solución única, de esta manera el pivote nunca será cero. Con esta precaución, si un pivote se anula, será porque el sistema no es compatible y determinado, en cuyo caso no se puede resolver por este método.

Ejemplo 8.21

Escribir un programa que resuelva sistemas de ecuaciones lineales por el método de Gauss-Jordan. Deberá calcular además el determinante de la matriz de los coeficientes.

Solución. La resolución del sistema ya ha sido analizada en el desarrollo matemático anterior. Para el cálculo del determinante, nos basamos en tres propiedades de los determinantes:

- Si multiplicamos una fila por una constante y sumamos el resultado a otra fila, el valor del determinante no cambia. Lo mismo ocurre con columnas.
- Si intercambiamos dos filas o dos columnas, el determinante cambia de signo.
- Si multiplicamos una fila o una columna por una constante, el determinante de la nueva matriz será el producto de la constante por el determinante anterior.

Teniendo en cuenta estas propiedades, si desarrollamos el determinante por el método de los adjuntos, será el producto de los pivotes una vez diagonalizada la matriz (antes de normalizar). Cada vez que intercambiamos dos filas, buscando el pivote máximo, habrá que cambiar de signo el determinante.

Algoritmo

NIVEL 0

INICIO

```
Leer Sistema (coeficientes del sistema de ecuaciones)
Diagonalizar la matriz ampliada del sistema
    (calculando a la vez el determinante)
Escribir el valor del determinante
Escribir columna n+1, que contiene las soluciones
```

FIN

NIVEL 1. Sólo desarrollaremos el algoritmo para diagonalizar la matriz, pues los otros se refieren a tareas ya estudiadas.

APLICACIONES AL CALCULO NUMERICO

ACCION *Diagonalizar es:*

NECESITA: Matriz aumentada del sistema. N° de ecuaciones.
MODIFICA: La matriz del sistema.
PRODUCE: Las soluciones del sistema, que quedan en la columna n+1 de la matriz de coeficientes.
El determinante.

INICIO

```
determinante := 1; (acumulará un producto)
DESDE iPiv :=1 HASTA n HACER
  BuscaPivote; (Intercambia fila iPiv con aquella que
                tenga el máximo valor, en valor
                absoluto, en la columna del pivote)
  pivote := aiPiv,iPiv;
  determinante := determinante * pivote;
  Normalizar fila del pivote; (divide por el pivote para
                              que quede un 1 en la
                              diagonal principal)
  DESDE i:=1 HASTA n HACER (Hacemos ceros en la columna
                          del pivote)
    SI i ≠ iPiv
      ENTONCES
        factor:= -ai,iPiv;
        Sustituir fila i por (fila i + factor * fila iPiv);
      FIN_SI;
FIN;
```

Codificación en Pascal

```
PROGRAM GaussJordan (input, output);
CONST
  max=10;
  necmax=10;
TYPE
  vector = ARRAY[0..max] OF real;
  matriz = ARRAY[1..necmax] OF vector;
VAR
  nec:integer; (* n° de ecuaciones *)
  sistema:matriz; (* dimensiones: nec x nec+1 *)
  d: real; (* columna n+1: términos independientes *)
  i: integer;
(*-----*)
PROCEDURE Diagonalizar(VAR det: real; VAR a: matriz; n: integer);
(* Convierte la matriz en triangular, con lo que las soluciones al
sistema son los elementos de la última columna.*)
VAR
  i,j,iPiv: integer;
  pivote, factor: real;
(*-----*)
PROCEDURE Normalizar(VAR v:vector; piv:real);
VAR
  j: integer;
BEGIN
  FOR j:=1 TO n+1 DO
    v[j] := v[j]/piv;
  END;
(*-----*)
```

ESTRUCTURA DE DATOS ARRAY

```

PROCEDURE Combilineal(VAR v1,v2:vector; fact:real);
VAR
  j:integer;
BEGIN
  FOR j:=1 TO n+1 DO
    v1[j] := v1[j] + fact * v2[j];
  END;
  (*-----*)
PROCEDURE Intercambiar(VAR v1, v2: vector);
VAR
  aux: vector;
BEGIN
  aux := v1;
  v1 := v2;
  v2 := aux;
END;
  (*-----*)
PROCEDURE BuscaPivote(VAR a:matriz; fila: integer; VAR det: real);
VAR
  h: integer;
BEGIN
  FOR h:= fila+1 TO n DO
    IF Abs(a[h,fila]) > Abs(a[fila,fila])
      THEN
        BEGIN
          Intercambiar(a[fila],a[h]);
          (* Al intercambiar dos filas, el determinante cambia de signo *)
          det := - det;
        END;
  END;
  (*-----*)
BEGIN (* Diagonalizar *)
  det := 1;
  FOR iPiv:=1 TO n DO
    BEGIN
      BuscaPivote(a,iPiv,det);
      pivote := a[iPiv,iPiv];
      (* Determinante = producto de los pivotes *)
      det := det * pivote;
      (* Dividimos la fila del pivote por el pivote *)
      Normalizar(a[iPiv],pivote);
      FOR i:=1 TO n DO (* Hacemos ceros en la columna del pivote *)
        IF i<> iPiv
          THEN
            BEGIN
              factor := -(a[i,iPiv]);
              CombiLineal(a[i],a[iPiv],factor);
            END
      END;
  END;
  (*-----*)
PROCEDURE LeeSistema(VAR x:matriz;VAR n:integer);
VAR
  i,j: integer;
BEGIN
  Write('¿N° de ecuaciones (1..10)?');
  Readln(n);
  (* Para que el sistema sea compatible y determinado: n° ec. = n° incóg. *)
  WriteLn('Introducción de los coeficientes del sistema:');
  FOR i:=1 TO n DO
    BEGIN
      FOR j:=1 TO n DO
        BEGIN
          Write('Coeficiente ',i:2,', ',j:2,': ');
          Readln(x[i,j]);
        END;
      Write('Término independiente de la ecuación ',i:2,': ');
    END;
  END;

```

APLICACIONES AL CALCULO NUMERICO

```
        Readln(x[i,n+1]);
    END;
END;
(*-----*)
BEGIN
    LeeSistema(sistema, nec);
    Diagonalizar(d, sistema, nec);
    Writeln('El determinante de la matriz de coeficientes es ', d:8:3);
    Writeln('Las soluciones del sistema son:');
    FOR i:=1 TO nec DO
        Writeln('X', i, ' = ', sistema[i,nec+1]:8:2);
    Writeln('Pulse <Intro> para terminar...');
    Readln;
END.
```

METODO DE CUADRATURA DE GAUSS DE INTEGRACION NUMERICA

Muchos algoritmos de cálculo numérico utilizan la técnica de aproximar la función en estudio (más o menos complicada) por un polinomio en un determinado intervalo, y a continuación se trabaja con el polinomio para derivar, integrar, etc. Esta técnica es usada en los métodos de integración numérica más conocidos: Método del trapecio, regla de Simpson y método de Romberg. Este último método es mejor que los anteriores, al ser aplicable a cualquier función cuya variación sea suave en cualquier intervalo finito que no contenga singularidades. Todos estos métodos dividen el intervalo de trabajo en una serie de puntos equiespaciados. Si prescindimos de esta última restricción, obtenemos una serie de algoritmos de integración numérica, conocidos en conjunto con el nombre de *Cuadratura de Gauss* (*cuadratura* es el nombre técnico de la integración numérica).

Los métodos clásicos de integración aproximan una integral por la suma de los valores que toma la función integrando en una serie de puntos equiespaciados, multiplicados por unos coeficientes llamados *factores de peso*, w_i :

$$\int_a^b f(x)dx \approx \sum_{i=0}^n w_i \cdot f(x_i) \quad (1)$$

donde $x_i = a + i\Delta x_i$.

Los pesos w_i dependen del algoritmo utilizado, pues vienen determinados por el polinomio empleado para el ajuste de la función.

Gauss sugirió que la precisión del método de cálculo aumentaría significativamente eligiendo, tanto los puntos en que se calcula la función, como los factores de peso, de forma que la precisión total sea óptima. El proceso debería dar un resultado exacto al aplicarlo a un polinomio del mayor grado posible.

Si en la aproximación de la integral utilizamos los puntos $x_0, x_1, x_2, \dots, x_n$, tenemos que determinar $2n+2$ parámetros ($n+1$ puntos x_i y sus correspondientes factores de peso, w_i). Si un polinomio de grado n tiene $n+1$ coeficientes, el algoritmo de Gauss con $n+1$ puntos debe dar un resultado exacto para cualquier polinomio de grado $2n+1$ o menor.

En general, los métodos de cuadratura de *Gauss* se desarrollan para integrales en el intervalo $(-1, 1)$. Si tenemos que resolver una integral en un intervalo distinto, (a, b) , bastará con realizar un cambio de variable antes de utilizar Gauss.

El objetivo del método es determinar los parámetros $x_0, x_1, x_2, \dots, x_n; w_0, w_1, w_2, \dots, w_n$ para que la expresión:

$$\int_{-1}^1 f(x)dx \approx w_0 \cdot f(x_0) + w_1 \cdot f(x_1) + w_2 \cdot f(x_2) + \dots + w_n \cdot f(x_n) \quad (2)$$

tenga la precisión óptima.

Por ejemplo, para $n=1$ (aproximación de primer orden), tenemos que calcular cuatro parámetros: x_0, x_1, w_0, w_1 . La ecuación anterior se reduce a:

$$\int_{-1}^1 f(x)dx \approx w_0 \cdot f(x_0) + w_1 \cdot f(x_1) \quad (3)$$

La ecuación debe ser exacta para un polinomio de grado 3 ($2n+1$) o menor. Aplicando esta condición a las funciones $1, x, x^2$, y x^3 tenemos:

$$\int_{-1}^1 1dx = w_0 + w_1 \quad (4)$$

$$\int_{-1}^1 x dx = w_0 \cdot x_0 + w_1 \cdot x_1 \quad (5)$$

$$\int_{-1}^1 x^2 dx = w_0 \cdot x_0^2 + w_1 \cdot x_1^2 \quad (6)$$

$$\int_{-1}^1 x^3 dx \approx w_0 \cdot x_0^3 + w_1 \cdot x_1^3 \quad (7)$$

Estas cuatro ecuaciones son no lineales en los cuatro parámetros. Escribiendo en forma matricial las ecuaciones (4) y (7) tenemos:

$$\begin{pmatrix} x_0 & x_1 \\ x_0^3 & x_1^3 \end{pmatrix} \times \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = 0$$

Descartando la solución trivial ($w_0 = w_1 = 0$), el determinante de la matriz de orden dos tiene que ser nulo:

$$x_0 \cdot x_1^3 - x_1 \cdot x_0^3 = 0$$

donde, si $x_0 \neq 0$ y $x_1 \neq 0$, tenemos:

$$x_1^2 - x_0^2 = 0$$

o, lo que es lo mismo:

$$x_1 = \pm x_0$$

La solución $x_1 = x_0$ no nos sirve, ya que partimos de la suposición inicial del cálculo de la función en dos puntos distintos. Por tanto, obtenemos $x_1 = -x_0$. Sustituyendo en la ecuación (5) nos queda:

$$w_1 = w_0$$

Llevando este valor a las ecuaciones (4) y (6):

$$w_0 + w_1 = 2 \rightarrow w_0 = w_1 = 1$$

$$w_0 \cdot x_0^3 + w_1 \cdot x_1^3 = \frac{2}{3} \rightarrow x_1 = -x_0 = \frac{1}{\sqrt{3}}$$

Sustituyendo en la ecuación (3) obtenemos la expresión correspondiente a la cuadratura de Gauss de primer orden:

$$\int_{-1}^1 f(x) dx \approx f\left(\frac{-1}{\sqrt{3}}\right) + f\left(\frac{+1}{\sqrt{3}}\right) \quad (8)$$

Se puede comprobar que esta aproximación tan sencilla da un resultado exacto para la integral de cualquier polinomio de grado menor o igual a tres. Si $f(x)$ no es polinómica de grado menor o igual a tres, la ecuación (8) nos da una aproximación del valor real. El grado de aproximación depende de cómo la función $f(x)$ se aproxime a un polinomio de grado menor o igual a tres.

Para aumentar la precisión del resultado en la ecuación (3), hay que aumentar el número de puntos, n , en que se evalúa la función $f(x)$. Para cada elección de n hay que determinar todos los parámetros w_i y x_i , de forma que se obtenga un resultado exacto para polinomios de grado menor o igual a $2n+1$. Estos valores han sido calculados y tabulados desde $n=2$ hasta $n=95$. En la tabla 8.1 se resumen los valores de estos parámetros para distintos valores de n .

Ejemplo 8.22

Construir una función para resolver integrales por el método de cuadratura de Gauss, con 20 puntos, en el intervalo $(-1, 1)$. Escribir un programa sencillo que la utilice.

Solución: La fase de análisis sería el desarrollo matemático anterior. Aplicaremos la ecuación (2), con los parámetros de la tabla 8.1 correspondientes a $n=20$.

ESTRUCTURA DE DATOS ARRAY

n	i	x_i	w_i
2	0	-0.5773502692	1.0000000000
	1	0.5773502692	1.0000000000
3	0	-0.7745966692	0.5555555556
	1	0.0	0.8888888889
	2	0.7745966692	0.5555555556
5	0	-0.9061798459	0.2369268850
	1	-0.5384693101	0.4786286705
	2	0.0	0.5688888889
	3-4	¹⁹	
10	0	-0.9739065285	0.0666713443
	1	-0.8650633667	0.1494513492
	2	-0.6794095683	0.2190863625
	3	-0.4333953941	0.2692667193
	4	-0.1488743390	0.2955242247
5-9	Véase nota a pie de página		
20	0	-0.9931285992	0.0176140071
	1	-0.9639719273	0.0406014298
	2	-0.9122344283	0.0626720483
	3	-0.8391169718	0.0832767416
	4	-0.7463319065	0.1019301198
	5	-0.6360536807	0.1181945320
	6	-0.5108670020	0.1316886384
	7	-0.3737060887	0.1420961093
	8	-0.2277858511	0.1491729865
	9	-0.0765265211	0.1527533871
10-19	Véase nota a pie de página		

Tabla 8.1 Puntos de muestreo y factores de peso para cuadraturas de Gauss

Algoritmo

Desarrollaremos solamente el algoritmo de la función que calcula la integral, ya que el programa es solamente un ejemplo de utilización de dicha función. Para poder utilizar una función como parámetro, definimos un *tipo procedural* y declaramos una variable de ese tipo, que será pasada como argumento a la función.

Para almacenar los parámetros de la tabla 8.1 utilizamos dos *arrays*, uno para los pesos, w_i y otro para los puntos de evaluación de la función, x_i . Los índices de ambos *arrays* variarán de 0 a 19.

¹⁹ Los puntos de muestro están situados de forma simétrica con respecto a cero, de forma que $x_{n-k-1} = -x_k$. Los factores de peso son los mismos para los puntos simétricos, de forma que $w_{n-k-1} = w_k$.

APLICACIONES AL CALCULO NUMERICO

FUNCION IntegraGauss

NECESITA: Función a integrar
MODIFICA: Nada
PRODUCE: Integral de la función en el intervalo (-1, 1)

```
INICIO
  Inicializamos el valor de los parámetros (tabulados):
  x0 := valor tabla;   w0 := valor tabla;
  x1 := valor tabla;   w1 := valor tabla;
  x2 := valor tabla;   w2 := valor tabla;
  ...
  x19 := valor tabla;  w19 := valor tabla;
  Calculamos el valor de la ecuación (2)
  s := 0;
  DESDE i:=0 HASTA 19 HACER
    s := s + wi * F(xi);
  FIN_DESDE;
  IntegraGauss := s;
FIN
```

Codificación en Pascal

```
PROGRAM EjemploIntegracionGauss(input, output);
TYPE
  tipoFuncion = FUNCTION(x: real): real;
VAR
  p: tipoFuncion;
  x, int: real;
  (*****)
FUNCTION poli(x: real):real; far;
BEGIN
  poli := 4*x*x*x + 2*x - 1;
END;
  (*****)
FUNCTION IntegraGauss(VAR F: tipoFuncion): real;
CONST
  n = 20;
TYPE
  indice = 0..19;
  vector = ARRAY[indice] OF real;
VAR
  x, w: vector;
  s: real; (* auxiliar para acumular suma de wi*f(xi) *)
  i: integer;
BEGIN
  (* Inicialización de parámetros *)
  x[0] := - 0.9931285992; w[0] := 0.0176140071;
  x[1] := - 0.9639719273; w[1] := 0.0406014298;
  x[2] := - 0.9122344283; w[2] := 0.0626720483;
  x[3] := - 0.8391169718; w[3] := 0.0832767416;
  x[4] := - 0.7463319065; w[4] := 0.1019301198;
  x[5] := - 0.6360536807; w[5] := 0.1181945320;
  x[6] := - 0.5108670020; w[6] := 0.1316886384;
  x[7] := - 0.3737060887; w[7] := 0.1420961093;
  x[8] := - 0.2277858511; w[8] := 0.1491729865;
  x[9] := - 0.0765265211; w[9] := 0.1527533871;
  x[10] := 0.0765265211; w[10] := 0.1527533871;
  x[11] := 0.2277858511; w[11] := 0.1491729865;
  x[12] := 0.3737060887; w[12] := 0.1420961093;
```

ESTRUCTURA DE DATOS ARRAY

```
x[13] := 0.5108670020; w[13] := 0.1316886384;
x[14] := 0.6360536807; w[14] := 0.1181945320;
x[15] := 0.7463319065; w[15] := 0.1019301198;
x[16] := 0.8391169718; w[16] := 0.0832767416;
x[17] := 0.9122344283; w[17] := 0.0626720483;
x[18] := 0.9639719273; w[18] := 0.0406014298;
x[19] := 0.9931285992; w[19] := 0.0176140071;
(* Aplicación de la fórmula de cuadratura de Gauss *)
s := 0;
FOR i:=0 TO 19 DO
  s := s + w[i] * F(x[i]);
IntegraGauss := s;
END;
(*****
BEGIN
  p := poli;
  int := IntegraGauss(p);
  Writeln('Integración de un polinomio por Gauss:');
  Write('La integral entre -1 y 1 de P(x) = 4x3 + 2x -1 ');
  Writeln('vale: ', int:10:6);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

8.11 EXTENSIONES DEL COMPILADOR TURBO PASCAL

FUNCIONES LOW Y HIGH

Las funciones *Low* y *High* son aplicables a un identificador de tipo *array*. Devuelven respectivamente los límites inferior y superior del tipo índice del *array*.

FUNCION SIZEOF

La función *SizeOf* es aplicable a un argumento de tipo *array*, y devuelve el tamaño del argumento en bytes.

PALABRA RESERVADA PACKED

El compilador Turbo Pascal admite la utilización de la palabra reservada *PACKED*, pero no tiene ningún efecto, es ignorada por el compilador. Se admite por razones de portabilidad de los programas, pero no tiene sentido su utilización, ya que el compilador Turbo Pascal está diseñado para ser utilizado en ordenadores personales, en los que el empaquetamiento no ahorra memoria. Sin embargo, el empaquetamiento se produce automáticamente siempre que es posible.

PARAMETROS ABIERTOS (*Open parameters*)

Los parámetros abiertos permiten el uso de *strings* y *arrays* de tamaño variable como parámetros de un procedimiento o función.

• Parámetros string abiertos

Se declaran usando el identificador *OpenString*, que se utiliza para definir un tipo especial de *string*, usado únicamente en la declaración de parámetros *string*. Con un parámetro de este tipo, el argumento (o parámetro actual) correspondiente puede ser una variable de cualquier tipo *string*. Dentro del procedimiento o función, la longitud máxima del parámetro formal será la misma del parámetro actual (argumento) correspondiente.

Ejemplo 8.23

```
PROCEDURE AsignaString (VAR S: OpenString);
BEGIN
  S := '1234567890ABCDEF';
END;
```

Al ser *s* un *parámetro string abierto*, se pueden pasar argumentos de cualquier tipo *string* al subprograma *AsignaString*, como se indica a continuación:

```
VAR
  S1: string[10];
  S2: string[20];
BEGIN
  AsignaString(S1);      (* S1 = '1234567890' *)
  AsignaString(S2);      (* S2 = '1234567890ABCDEF' *)
END;
```

Dentro de *AsignaString* la longitud máxima de *s* es la del parámetro actual. En la primera llamada al subprograma, la asignación trunca el *string* porque la longitud de *s1* es 10.

• Parámetros array abiertos

Se declaran como un *array* normal, sin especificar tipo subíndice, usando la sintaxis:

```
ARRAY OF T;
```

T debe ser un identificador de tipo, y el parámetro actual correspondiente debe ser un *array* cuyos elementos sean de tipo *T*, o una variable de tipo *T*. Dentro del procedimiento o función, el parámetro formal se comporta como si hubiera sido declarado como

```
ARRAY [0..n-1] OF T
```

siendo *n* el número de elementos del parámetro actual. Si el parámetro actual es una variable de tipo *T*, es tratado como si fuese un *array* con *un elemento* de tipo *T*.

No está permitida una asignación a un *parámetro array abierto* completo, ni en general operaciones con estos *arrays* completos. Hay que operar con sus elementos.

Ejemplo 8.24

Veamos como utilizar *parámetros array abiertos*:

ESTRUCTURA DE DATOS ARRAY

```
PROCEDURE Inicializar(VAR a: ARRAY OF real);
VAR i: word;
BEGIN
  FOR i := 0 TO High(a) DO a[i] := 0;
END;
```

Ejemplo 8.25

El procedimiento `Inicializar` puede utilizarse con cualquier *array* de reales, porque `a` se declara como *parámetro array abierto*. Lo mismo sucede con la función `Suma` del ejemplo siguiente:

```
FUNCTION Suma(VAR a: ARRAY OF real): real;
VAR i: word;
    s: real;
BEGIN
  s := 0;
  FOR i := 0 to High(a) DO
    s := s + a[i];
  Suma := s;
END;
```

EL TIPO STRING EN TURBO PASCAL

El compilador Turbo Pascal incorpora el tipo *string*, como ayuda al manejo de cadenas de caracteres. Para definirlo utiliza la palabra reservada *STRING*, según el siguiente esquema:

```
TYPE <Identificador de tipo> = STRING (<vacío> | [< Cte. entera >]);
```

La constante entera es opcional, y debe estar en el rango 1..255. Su propósito es definir la longitud máxima de la cadena. Si no aparece, la longitud máxima por defecto es 255. Los *strings* se caracterizan además, porque se conoce en cada momento su longitud dinámica o longitud actual, según se explica a continuación.

Las diferencias entre las cadenas de caracteres y el tipo *string* en Turbo Pascal son:

- Una variable declarada como *array empaquetado de caracteres*, sólo es compatible con variables declaradas exactamente con el mismo tipo. Sin embargo, las variables declaradas de un tipo *string* pueden asignarse a otras de distintos tipos *string*, independientemente de su longitud o identificador de tipo.
- Cuando se asigna una cadena, si la cadena fuente es mayor que la cadena destino, la cadena destino contendrá una versión truncada de la cadena fuente después de la asignación.
- Una variable de tipo *string* no tiene que rellenarse completamente, tal y como ocurre con las variables declaradas como *arrays empaquetados de caracteres*.
- Cuando se escribe una variable de tipo *string*, con una sentencia *Write* o *Writeln*, el número de caracteres escritos está determinado por su longitud actual, y no por la longitud máxima definida.
- Una variable declarada como de tipo *array empaquetado de caracteres*, puede asignarse a variables de tipo *string*.

EXTENSIONES DEL COMPILADOR TURBO PASCAL

- Una variable de tipo *string* puede leerse directamente con sentencias *Read* o *Readln*.
- El espacio asignado a una variable de tipo *string* por el Turbo Pascal es un byte mayor que la longitud indicada en la definición de su tipo. El byte adicional se utiliza para almacenar la longitud actual de la cadena. Este byte de longitud se almacena en el elemento cero del *string*. Es decir, el elemento 0 del *string* es: `Chr(long)`, siendo `long` la longitud actual del *string*.

Ejemplo 8.26

```
PROGRAM EjemploDeString (output) ;
  TYPE
    string80 = STRING [80];
  VAR
    saludo: string80;
  BEGIN
    saludo := 'Hola soy un string' ;
    Writeln (saludo , Ord (saludo[0]))
  END.
```

La ejecución de este programa es:

```
Hola soy un string18
```

Ejemplo 8.27

También se puede trincar el *string*, tal y como se muestra a continuación:

```
PROGRAM EjemploDeStringTruncado (output);
  TYPE
    string 80 = STRING [80];
  VAR
    saludo : string80;
  BEGIN
    saludo := 'Hola soy un string' ;
    saludo[0]:=Chr (4);
    Write(saludo, Ord(saludo[0]))
  END.
```

La salida del programa será:

```
Hola4
```

Subprogramas incorporados por el Turbo Pascal para el manejo de strings

En la tabla 8.2 se presentan los procedimientos incorporados por el compilador Turbo Pascal para un manejo más eficiente del tipo *string*.

Sintaxis y parámetros

Las declaraciones de estos subprogramas se indican a continuación, con el fin de ilustrar los parámetros utilizados y sus tipos.

```
Concat (S1, S2, ..., Sn: STRING): STRING;
```

```
Copy (cadena: STRING; posición, longitud: integer): STRING;
```

ESTRUCTURA DE DATOS ARRAY

```

Insert (subcadena: STRING; VAR cadena: STRING; posición: integer);
Length (cadena: STRING): integer;
Delete (VAR cadena: STRING; posición, longitud: integer);
Pos (subcadena, cadena: STRING): integer;
Str (i: integer; VAR cadena: STRING);
Str (r: real; VAR cadena: STRING);
Val (cadena: STRING; VAR r: real; VAR codigo: integer);
Val (cadena: STRING; VAR i, codigo: integer);

```

El parámetro `codigo` se utiliza para indicar el éxito o fallo de la operación, pues no toda cadena se puede convertir en un número. Si toma el valor 0 la conversión es correcta, en caso contrario no.

Identificador	Tipo de subprograma	Descripción
<i>Concat</i>	Función	Concatena dos o más <i>string</i>
<i>Copy</i>	Función	Extrae una subcadena de una cadena
<i>Insert</i>	Procedimiento	Introduce una subcadena en una cadena
<i>Length</i>	Función	Da la longitud actual de la cadena
<i>Delete</i>	Procedimiento	Borra una parte de la cadena
<i>Pos</i>	Función	Devuelve la posición de una subcadena en una cadena
<i>Str</i>	Procedimiento	Convierte un tipo numérico a un tipo <i>string</i>
<i>Val</i>	Procedimiento	Convierte un tipo <i>string</i> a un tipo numérico

Tabla 8.2 Subprogramas para manejo de *strings* en Turbo Pascal

Operador concatenación "+"

Su efecto es similar a la función *Concat*, pero quizá su forma de aplicación sea más intuitiva.

EXTENSIONES DEL COMPILADOR TURBO PASCAL

Ejemplo 8.28

Para concatenar tres cadenas podemos poner:

```
cad4 := cad1 + cad2 + cad3;
```

donde *cad1*, *cad2*, *cad3*, *cad4* son variables de tipo *STRING*; *cad1*, *cad2* y *cad3* también podrían ser expresiones o constantes de tipo *STRING*.

Si en vez del operador +, se utilizase la función *Concat*, la sentencia de asignación sería:

```
cad4 := Concat (cad1, cad2, cad3);
```

Función UpCase

Convierte a mayúscula un carácter en minúscula, con excepción de la ñ y las vocales acentuadas. Solo trabaja con las letras que están por debajo del carácter 127 de la tabla ASCII. Su sintaxis es:

```
UpCase (carácter: char): char;
```

Ejemplo 8.29

El siguiente programa se ha escrito con la finalidad de ilustrar la utilización de los subprogramas de la tabla 8.2, y el operador concatenación.

```
PROGRAM ManejoStrings (input, output);
CONST
  blanco = ' ';
VAR
  nom, apel, ape2, alias, restos: string[15];
  nombreCompleto, edadS: string[60];
  lugar: 1..60;
  valida, edad, resto: integer;
BEGIN
  Write('Introduzca su primer apellido: ');
  Readln (apel);
  Write('Introduzca su segundo apellido: ');
  Readln (ape2);
  Write('Introduzca su nombre: ');
  Readln (nom);
  nombreCompleto := Concat(nom, blanco, apel, blanco, ape2);
  Writeln('Su nombre completo es: ');
  Writeln(nombreCompleto);
  Write('Sus iniciales son: ');

  (* Extraemos las iniciales con Copy *)
  Writeln(Copy(nom,1,1), Copy(apel,1,1), Copy(ape2,1,1));
  Writeln('Vamos a cambiarle el nombre de pila. ');
  Write('¿Cómo desea llamarse? ');
  Readln(alias);

  (* eliminamos el nombre original *)
  Delete(nombreCompleto, 1, Length(nom));
```

ESTRUCTURA DE DATOS ARRAY

```
(* insertamos el nuevo nombre *)
Insert(alias, nombreCompleto, 1);
Writeln('Ahora usted se llama ', nombreCompleto);

(* Probamos ahora a usar la función Pos *)
Write('La mayoría de los nombres completos en España ');
Writeln('contienen la cadena "ez".');
lugar := pos('ez', nombreCompleto);
IF lugar=0 THEN Writeln('El suyo es una excepción')
    ELSE Writeln('En el suyo aparece en la posición ', lugar);

(* Para finalizar convertimos strings a números y viceversa *)
Writeln('Por último, calcularemos su edad en lustros. ');
REPEAT
  Write('¿Cuántos años tiene usted? ');
  Readln(edadS); (* Leemos la edad como una cadena de caracteres *)
  Val(edadS, edad, valida); (* Transformamos string en número entero *)
  If valida <> 0 THEN Writeln('Se ha confundido tecleando. ');
UNTIL valida = 0;
resto := edad MOD 5;
edad := edad DIV 5;
Str(edad, edadS); (* Transformamos nº en string *)
Str(resto, restoS);
edadS := edadS + blanco + 'lustros y ' + restoS + blanco + 'años';
Write('Su edad es: ');
Writeln(edadS);
Readln;
END.
```

8.12 CUESTIONES Y EJERCICIOS RESUELTOS

8.1 Indíquense las definiciones de tipo correctas e incorrectas, señalando, para las correctas el número de elementos del *array*.

```
TYPE
  cosas = (lapiz, papel, pluma, pincel);
  m1 = ARRAY['a'..'d', -1..2] OF 'a'..'z';
  m2 = ARRAY[cosas, 1..4] OF cosas;
  m3 = ARRAY[integer] OF char;
  m4 = ARRAY[1.0..2.5] OF real;
```

Solución

La declaración de *m1* es sintácticamente correcta. El array tiene 16 elementos.

La declaración de *m2* es sintácticamente correcta. El array tiene también 16 elementos.

La declaración de *m3* es sintácticamente correcta, pero de difícil manejo, pues es una estructura de mucha ocupación de memoria. El array tiene 65.535 elementos de tipo *char*, y necesita un byte por elemento.

La declaración de *m4* es incorrecta sintácticamente, ya que el tipo índice tiene que ser un tipo ordinal; puede ser cualquiera de los tipos simples estudiados, excepto el tipo *real*. Además no está permitido un tipo subrango o intervalo cuyo tipo base sea *real*.

CUESTIONES Y EJERCICIOS RESUELTOS

8.2 Sea la declaración:

```
TYPE
  cad = PACKED ARRAY OF char;
VAR
  x,y: cad;
```

Escribir una función booleana, llamada por `Invert(a,b)` que devuelva `true` si `a` es la cadena invertida de `b` y `false` en caso contrario. Ejemplo:

ANTES_ es inverso a _SETNA.

Solución

```
FUNCTION Invert(x, y: cadena):boolean;
CONST
  l=6;
VAR
  i, m: integer;
  inv: boolean;
BEGIN
  inv := true;
  i := 1;
  WHILE (i<=l) AND inv DO
    IF a[i] <> b[l+1-i]
      THEN inv := false;
  Invert := inv;
END;
```

8.3 La tabla siguiente representa las variaciones de peso (en Kgs) de cuatro personas a lo largo de los meses de un año:

	EVA	ANA	ROSA	MARIA
Enero	+1.3	+0.4	-1.6	-0.8
Febrero	+0.5	+1.5	-1.2	+0.4
...
Diciembre	-0.5	-1.5	+2.2	+1.7

Si declaramos los tipos:

```
TYPE
  mujeres = (Eva, Ana, Rosa, Maria);
  meses = (ene, feb, mar, abr, may, jun, jul,
          ago, sep, oct, nov, dic)
```

Se pregunta:

- ¿Cómo se declararía dicha tabla en Pascal?
- Escribir un bucle que permita explorar la tabla

ESTRUCTURA DE DATOS ARRAY

Solución

```
a) TYPE
    tabla = ARRAY[meses, mujeres] OF real;
VAR
    pesos: tabla;
```

```
b) VAR
    i: meses;
    j: mujeres;
    ...
BEGIN
    ...
    FOR i := ene TO dic DO
        FOR j := Ana TO Maria DO
            Explorar(pesos[i,j]);
    ...
```

Explorar(pesos[i,j]); puede ser una sentencia, o un subprograma con un parámetro de tipo real.

8.4 Sea la declaración

```
TYPE
    mat = ARRAY [1..10, 1..10] OF char;
VAR
    a: mat;
```

Escribir un trozo de programa que determine sobre la matriz a:

- Si existe alguna 'A' en la diagonal principal.
- Si existe algún '+' en la fila 5.
- Cuántas 'e' hay en la columna 6

Solución

```
VAR
    a: mat;
    existeA, existeMas: boolean;
    i, j, e: integer;
    ...
BEGIN
    ...
    (* Verificamos existencia de 'A' en la diagonal *)
    existeA := false;
    existeMas := false;
    e := 0;
    FOR j := 1 TO 10 DO
        BEGIN
            IF a[j,j] = 'A'
                THEN existeA := true;
            IF a[5,j] = '+'
                THEN existeMas := true;
            IF j=6
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
        THEN
          FOR i:=1 TO 10 DO
            IF a[i,j] = 'e'
              THEN e := e + 1;
            END;
          END;
        IF existeA
          THEN Writeln('Existe alguna 'A' en la diagonal principal')
          ELSE Writeln('No existe ninguna 'A' en la diagonal principal')
        IF existeMas
          THEN Writeln('Existe algun '+' en la fila 5')
          ELSE Writeln('No existe ningun '+' en la fila 5')
        Writeln('El número de 'e' en la columna 6 es ', e:2);
        ...
      END;
```

8.5 Sea la declaración

```
TYPE
  notas = (doo, re, mi, fa, sol, la, si);
  mat = ARRAY [notas] OF notas;
VAR
  x, y: mat;
```

- a) ¿Es correcta? Si no lo es, modifíquela para que sea correcta.
- b) Escribir un procedimiento llamado por la sentencia `Invertir(x,y)`, que devuelva en `y` el valor de `x` invertido. Se debe utilizar una estructura repetitiva en el procedimiento. Por ejemplo:
Para `x = doo doo si mi fa fa re`
Se debe obtener: `y = re fa fa mi si doo doo`

Solución

- a) Es correcta.
- b)

```
PROCEDURE Invertir(a: notas; VAR b: notas);
VAR i, j: notas;
BEGIN
  j := si;
  FOR i := doo TO si DO
    BEGIN
      b[i] := a[j];
      j := Pred(j);
    END;
  END;
```

8.6 Sea la declaración:

```
TYPE
  t = ARRAY[-1000..1000] OF 'a'..'z';
VAR
  x: t;
```

ESTRUCTURA DE DATOS ARRAY

Se pide escribir un procedimiento que tras la llamada `Control(x)` nos devuelva el número de elementos del `array x` igual a cada letra minúscula. Declarar parámetros y variables pensando que el procedimiento no debe devolver nada, solo imprimir el resultado pedido.

Solución

```
PROCEDURE Control(VAR v: t);
VAR
  letra: char;
  i: -1000..1000;
  cont: integer;
BEGIN
  cont := 0;
  FOR letra := 'a' TO 'z' DO
    BEGIN
      cont := 0;
      FOR i:=-1000 TO 1000 DO
        IF (v[i]=letra) THEN cont := cont + 1;
        Writeln('LETRA ', letra:1, ' aparece ', cont:3, ' veces');
      END;
    END;
  END;
```

8.7 Los dos algoritmos siguientes están diseñados para calcular la posición de un dato de tipo entero (almacenado en la variable `buscado`), dentro de un vector de números enteros (representado por la variable `v`, de tipo `array` de enteros). El resultado se almacena en la variable de tipo entero `posicion` (que tomará el valor 0 si el dato buscado no es un elemento del `array v`). Otras variables auxiliares usadas son:

`i`: contador para recorrer secuencialmente el vector (tipo entero)
`m`: Número de elementos del vector (tipo entero), conocido
`encontrado`: variable auxiliar de tipo lógico.

a) Acción Cálculo Posición es:

```
INICIO
  posicion := 0;
  DESDE i:=1 HASTA m HACER
    SI v[i]=buscado
      ENTONCES posicion := i;
    FIN_SI;
  FIN_DESDE;
FIN
```

b)

```
INICIO
  encontrado := falso;
  i := 0;
  MIENTRAS (i<m) Y NO(encontrado) HACER
    i := i + 1;
    SI v[i]=buscado ENTONCES encontrado := cierto;
    FIN_SI;
  FIN_MIENTRAS;
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
SI encontrado
  ENTONCES posicion := i
  SI_NO    posicion := 0;
FIN_SI;
FIN
```

1. ¿Cuál es el más eficiente? ¿Por qué?
2. ¿Cómo modificar el algoritmo más eficiente para el caso de que los elementos del vector y la variable buscado sean números reales? ¿Por qué?

Solución

1. Es más eficiente la versión b), ya que el proceso de búsqueda se detiene en cuanto que se encuentra el valor buscado, mientras que el algoritmo de la versión a) recorre todo el vector hasta el final, aunque encuentre antes el valor buscado. La versión a) se usaría para encontrar la última aparición de `buscado` en el vector.
2. En el caso de números reales, no debemos efectuar la comparación mediante el operador `=` (igualdad). Los números reales pueden no tener representación exacta, y los errores representacionales se propagan al operar con ellos, dando lugar a errores computacionales. La aritmética real no es exacta. Si los números a comparar difieren en la última cifra decimal, el resultado de la comparación será *false*, aunque probablemente no sea el resultado esperado por nosotros. Lo que se debe hacer es mirar si el valor absoluto de su diferencia es muy pequeño:

```
SI Abs(v[i]-buscado) < epsilon
  ENTONCES encontrado:=cierto;
```

siendo `epsilon` la precisión requerida.

8.8 Escribir un programa que realice el producto escalar y vectorial de dos vectores.

Solución

```
PROGRAM Vectores (input,output);
VAR
  s: real;
  x: integer;
  a,b,c: ARRAY [1..3] OF real;
BEGIN
  s:=0; (* acumulador para el producto escalar *)
  (* Lectura de las componentes *)
  FOR x:=1 TO 3 DO
    BEGIN
      Write('ESCRIBE LA COORDENADA ',x,' DE LOS DOS VECTORES :');
      Readln(a[x],b[x]);
    END;
  (* Producto escalar *)
  FOR x:=1 TO 3 DO s:=s+(a[x]*b[x]);
  (* Producto vectorial *)
```

ESTRUCTURA DE DATOS ARRAY

```
c[1]:=a[2]*b[3]-a[3]*b[2];
c[2]:=a[3]*b[1]-a[1]*b[3];
c[3]:=a[1]*b[2]-a[2]*b[1];
Writeln;
Writeln ('EL PRODUCTO ESCALAR ES :',S:10:2);
Writeln;
Writeln('EL PRODUCTO VECTORIAL ES : (' , c[1]:9:2, ', ' ,
      c[2]:9:2,',',c[3]:9:2,')');
Write('Pulsa <Intro> para volver al editor...');
Readln;
END.
```

- 8.9** Escribir un programa que determine el máximo y el mínimo de un conjunto de números reales.

Solución

```
PROGRAM MaxMin (input,output);
CONST
  m = 100 ;
TYPE
  valores = ARRAY [1..m] OF real;
VAR
  n,i: integer;
  x: valores;
  max,min: real;
BEGIN
  Writeln('CALCULO DEL MAXIMO Y MINIMO DE UN CONJUNTO DE VALORES');
  Writeln('*****');
  Writeln;
  Write('Introduzca el número de valores a leer ');
  Readln(n);
  Writeln;
  Write('Dame el primer valor ');
  Readln(x[1]);
  max:=x[1];
  min:=x[1];
  FOR i:=2 TO n DO
    BEGIN
      Write('Dame el valor ',i:3,' = ');
      Readln(x[i]);
      IF x[i]>max
        THEN max:=x[i];
      IF x[i]<min
        THEN min:=x[i];
    END;
  Writeln;
  Write('El valor máximo es ',max:10:4);
  Writeln(' y el mínimo es ',min:10:4);
  Writeln;
  Writeln(' N° VALOR');
  Writeln('*****');
  FOR i:=1 TO n DO
    Writeln(i:3,x[i]:14:4);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

- 8.10** Realizar un programa que determine la máxima precipitación mensual y la temperatura media mensual de un año determinado en una estación metereológica dada.

CUESTIONES Y EJERCICIOS RESUELTOS

Solución

```
PROGRAM Estacion (input, output);
TYPE
  meses = (enero, febrero, marzo, abril, mayo,
           junio, julio, agosto, septiembre,
           octubre, noviembre, diciembre);
  meteoro = ARRAY [meses] OF real;
VAR
  lluvia, temp: meteoro;
  mes: meses;
FUNCTION Maximo (v: meteoro): real;
VAR
  max : real;
BEGIN
  max := v[enero];
  FOR mes:= febrero TO diciembre DO
    IF v[mes] > max THEN max := v[mes];
  Maximo := max;
END;
BEGIN (*Programa principal*)
  FOR mes := enero TO diciembre DO
    BEGIN
      Write('Dame precip. y temperatura del mes ', Ord(mes)+1, ':');
      Readln (lluvia [mes], temp [mes]);
    END;
  Writeln ('Precipitación máxima mensual: ', Maximo (lluvia):4:1);
  Writeln ('Temperatura máxima de las medias mensuales:',
           Maximo(temp):4:1);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

En este ejemplo:

`max` es una variable *local*, con validez solamente dentro de la función.
`lluvia, temp, mes` son variables *globales* con validez en todo el programa.

- 8.11** Escribir un programa para calcular el valor medio de los elementos de un vector utilizando subprogramas.

Solución

```
PROGRAM ValorMedioDeUnVector (input, output);
CONST
  n = 10;
TYPE
  vector = ARRAY [1..n] OF integer;
VAR
  v: vector;
  i: integer;
  med: real;
PROCEDURE Media (v: vector; VAR med: real);
VAR
  j: integer;
```

ESTRUCTURA DE DATOS ARRAY

```
BEGIN
med := 0.0;
FOR j:=1 TO n DO
  med:= med + v[j];
med:= med/n;
END;
BEGIN (* Programa principal *)
FOR i:=1 TO n DO
  BEGIN
  Write ('Introducir componente', i:3, ': ');
  Readln (v[i])
  END;
Media (v, med);
Writeln ('La media vale ', med:8:2);
Write('Pulse <Intro> para volver al editor...');
Readln;
END.
```

8.12 Escribir un procedimiento que realice una búsqueda secuencial en un vector.

Análisis: La manera más simple de realizar una búsqueda en un vector es comenzar por el primer elemento e ir examinando secuencialmente cada uno de sus elementos, hasta encontrar el valor buscado o el último elemento. Generalizando, el siguiente procedimiento realiza la búsqueda en una porción del vector, entre las posiciones `limInf` y `limSup`.

Solución

```
PROCEDURE BusquedaSec(valores: vector; limInf,limSup: integer;
  valorBuscado: elementoVector;
  VAR posicion: integer; VAR hallado: boolean);
(* Este procedimiento busca en el vector valores, el valor
  contenido en valorBuscado. Si se encuentra hallado es true,
  y si no se encuentra false *)
BEGIN
hallado := false;
posicion := limInf;
WHILE (posicion <= limSup) AND (NOT hallado) DO
  IF (valores [posicion] = valorBuscado)
    (* Si los elementos del vector son reales hay que
    compararlos de otra manera *)
    THEN hallado := true
    ELSE posicion := posicion + 1;
END;
```

8.13 Escribir un procedimiento que realice una búsqueda dicotómica de un vector ordenado.

Análisis: Supongamos que queremos buscar el teléfono de un amigo en la guía telefónica. El método anterior no nos interesa, pues sería muy largo, y podemos ahorrar tiempo aprovechando que la guía está ordenada alfabéticamente. Abriremos la guía por donde esperamos encontrar el apellido de nuestro amigo, y según el resultado rechazaremos para la búsqueda la porción de guía anterior o posterior a la

CUESTIONES Y EJERCICIOS RESUELTOS

página actual. Repetiremos el proceso reduciendo en cada paso el área de búsqueda, hasta encontrar el nombre buscado, o descubrir que no está en la guía. Este método se llama *búsqueda binaria o dicotómica*, y se aplica a secuencias ordenadas de datos.

Área de búsqueda inicial: Intervalo (bajo..alto). Generalmente será (1..última posición).

El área de búsqueda se inicializa en la llamada, por el valor que tomen los parámetros *alto* y *bajo*. Inicializamos *hallado* a *false*, e *indiceMedio* a (bajo + alto) DIV 2. La variable *indiceMedio* es el subíndice más cercano a la mitad del área de búsqueda. En cada paso se reduce el área de búsqueda y se calcula el nuevo *indiceMedio*, hasta que encontremos el *valorDeseado* o el área de búsqueda quede vacía, lo cual sucede cuando *bajo* > *alto*.

Algoritmo

```
INICIO
  hallado := false;
  indiceMedio = (bajo + alto) DIV 2;
  MIENTRAS no (hallado) Y (alto >= bajo) HACER
    SI valores[indiceMedio] = valorDeseado
      ENTONCES hallado = TRUE  -> Termina la búsqueda
    SI_NO Reducimos área de búsqueda:
      SI valores [indiceMedio] > valorDeseado
        ENTONCES alto := indiceMedio-1
        SI_NO bajo := indiceMedio+1;
      FIN_SI;
      Calculamos nuevo indiceMedio:
        indiceMedio = (bajo + alto) DIV 2;
    FIN_SI;
  FIN_MIENTRAS;
FIN
```

Codificación en Pascal

```
PROCEDURE BusquedaBinaria (valores:vector; bajo, alto:integer;
  valorDeseado:elementoVector;
  VAR indiceMedio:integer;
  VAR hallado:boolean);
(* Este procedimiento busca en el vector valores el valor contenido
  en la variable ValorDeseado. Si se encuentra, la variable
  hallado toma el valor verdadero y la variable IndiceMedio es el
  índice de dicho valor.Si el valor no se encuentra hallado toma el
  valor false *)
BEGIN
  hallado := false;
  indiceMedio := (alto+bajo) DIV 2;
  WHILE (NOT hallado) AND (alto >= bajo) DO
    BEGIN
      IF (valores [indiceMedio] = valorDeseado)
        THEN hallado := true
      ELSE IF valores [indiceMedio] > valorDeseado
        THEN alto := indiceMedio-1
```

ESTRUCTURA DE DATOS ARRAY

```
        ELSE bajo := indiceMedio+1;
        indiceMedio := (bajo+alto) DIV 2;
    END;
END;
```

- 8.14** Escribir un procedimiento que ordene los elementos de un vector por el método de la burbuja.

Solución

```
PROCEDURE OrdenacionBurbuja (VAR valores: vector;
                             limInf,limSup: integer;
                             desAscendente: boolean);
(* Este procedimiento ordena el vector valores mediante el
  algoritmo de la burbuja. Valores es el vector de entrada/salida;
  limInf y limSup son los índices entre los cuales se ordena;
  desAscendente es true si se desea orden ascendente *)
VAR
  i,j: integer; (* índices de los bucles *)
(*-----*)
PROCEDURE Intercambia (VAR a, b : elementoVector);
(* Intercambia los valores de las variables a y b *)
VAR
  temp : elementoVector; (* almacenamiento temporal *)
BEGIN (* Intercambia *)
  temp:=a;
  a:=b;
  b:=temp;
END; (* Intercambia *)
(*-----*)
BEGIN (* OrdenacionBurbuja *)
  FOR j:=limSup DOWNTO limInf+1 DO
    FOR i:=limInf TO j-1 DO
      IF ((valores[i] > valores [j]) AND (desAscendente))
        THEN Intercambia (valores[i], valores[j])
      ELSE
        IF (valores [i] < valores [j]) AND (NOT desAscendente)
          THEN Intercambia (valores[i], valores[j]);
        END; (* OrdenacionBurbuja *)
      END;
```

- 8.15** Escribir un procedimiento que escriba una hilera de asteriscos, especificando cuantos se desean.

Solución

```
PROCEDURE Barras (x:integer);
VAR
  i: integer;
BEGIN
  IF x = 0
    THEN Writeln ( ' ' )
  ELSE
    BEGIN
      FOR i:=1 TO (x-1) DO
        Write ( '* ' );
        Writeln ( '* ' );
      END;
    END; (* Barras *)
END;
```

CUESTIONES Y EJERCICIOS RESUELTOS

Cuando se llame a este procedimiento, se pondrá:

```
...
Barras (5);
...
```

y la ejecución será:

```
* * * * *
```

- 8.16** Realizar un programa que construya un diagrama de barras, utilizando el procedimiento anterior.

Solución

```
PROGRAM DiagramaDeBarras (input, output);
TYPE
  vector = ARRAY [1..100] OF real;
VAR
  a: vector;
  m, i: integer;
  max: real;
  (*****
PROCEDURE Barras (x:integer);
VAR
  i: integer;
BEGIN
  IF x = 0
  THEN Writeln (' ')
  ELSE
  BEGIN
    FOR i:=1 TO (x-1) DO
      Write ('* ');
      Writeln ('* ');
    END;
  END; (* Barras *)
  (*****
PROCEDURE Leer (n:integer; VAR v: vector);
VAR i:integer;
BEGIN
  FOR i:=1 TO n DO
  BEGIN
    Write ('Dame el valor N° ', i:2, ' = ');
    Readln (v[i]);
  END;
END; (* Leer *)
  (*****
PROCEDURE Maximo (n:integer; VAR v:vector; VAR mayor:real);
VAR
  i: integer;
  aux: real;
BEGIN
  aux := v[1];
  FOR i:=2 TO n DO
    IF v[i]>aux THEN aux:=v[i];
  mayor := aux;
END; (* Maximo *)
  (*****
```

ESTRUCTURA DE DATOS ARRAY

```
PROCEDURE Escala (mayor: real; n: integer; VAR v: vector);
  (* Este procedimiento tiene en cuenta que se escribe en 70
     columnas, pero para pintar las barras sólo se usan 60 pues
     se dejan 10 para escribir el índice *)
CONST
  columns = 60;
VAR
  i:integer;
BEGIN
  FOR i:=1 TO n DO
    v[i]:= v[i] * columns / mayor;
  END; (* Escala *)
  (*****)
BEGIN (* Programa principal *)
  Write ('Introduzca el nº de valores: ');
  Readln (m);
  Leer (m,a);
  Maximo (m,a,max);
  FOR i:=1 TO 10 DO
    Write (' ');
  FOR i:=1 TO 49 DO
    Write ('-');
  Writeln ('-');
  FOR i:=1 TO m DO
    BEGIN
      Write (i:9, ' ');
      Barras (Round (a[i]))
    END;
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

NOTA: No se pueden representar valores negativos.

El resultado de la ejecución será:

```
1 * * * *
2 * *
3 *
4
5 * * * * * *
...
```

8.17 Dadas las siguientes declaraciones globales:

```
TYPE
  vector = ARRAY [1..100] OF integer;
VAR
  v: vector;
  n,x: integer;
```

a) Escribir un procedimiento que, llamado por la sentencia `Ordena(v,n)`, ordene de menor a mayor los n elementos del *array* v .

b) Escribir una función llamada por `Posicion(v,n,x)`, que devuelva la posición de x en el vector v , de n componentes. Si x no está en el vector, la función tomará el valor 0.

CUESTIONES Y EJERCICIOS RESUELTOS

- c) Escribir un procedimiento llamado por la sentencia `Borrar(v,n,x)`, que elimine el elemento `x` del vector `v`, actualizando el número de elementos del vector, `n`. Deberá utilizar los dos subprogramas anteriores (*Ordena* y *Posicion*). Si el elemento `x` no existe, se indicará por pantalla.
- d) Escribir un procedimiento llamado por la sentencia `Insertar(v,n,x)`, que inserte el elemento `x` en el vector `v`, actualizando el valor de `n`, número de elementos de `v`. También deberá utilizar los subprogramas *Ordena* y *Posicion*. Si el elemento ya existe, se indicará por pantalla y no se realizará la inserción.
- e) Escribir un programa con un menú de opciones, utilizando los subprogramas anteriores.

Solución

```
PROGRAM vectores(input, output);
uses CRT;
TYPE
  vector = ARRAY [1..100] OF integer;
VAR
  v: vector;
  n, x: integer;
  fin:boolean;
  opcion:char;
(*****)
PROCEDURE Ordena ( VAR w: vector; m:integer);
VAR
  i, j: integer;
  aux: integer;
BEGIN
  Write ('Ordenando');
  FOR j:=2 TO m DO
    FOR i:= m DOWNTO j DO
      IF w[i-1] > w[i] THEN
        BEGIN
          Write('.');
          aux := w[i-1];
          w[i-1] := w[i];
          w[i] := aux;
        END;
    Writeln;
  END;
(*****)
FUNCTION Posicion(w:vector; m,xBus:integer):integer;
VAR
  i: integer;
  encontrado:boolean;
BEGIN
  Writeln('Buscando...');
  encontrado:=false;i:=0;
  WHILE (i<m) AND NOT(encontrado) DO
    BEGIN
      i:=i+1;
      IF w[i]=xBus THEN encontrado:=true;
    END;
  IF encontrado THEN posicion:=i
  ELSE posicion:=0;
END;
(*****)
```

ESTRUCTURA DE DATOS ARRAY

```

PROCEDURE Borrar (VAR w:vector; VAR m:integer; xBor:integer);
VAR
    i, pos: integer;
BEGIN
    pos := Posicion (w, m, xBor);
    IF Pos = 0 THEN Writeln('El elemento ',xBor,' no existe')
    ELSE
        BEGIN
            m:=m-1;
            FOR i:=pos TO m DO w[i] := w[i+1];
        END;
END;
(*****
PROCEDURE Insertar(VAR w:vector; VAR m:integer; xIns:integer);
VAR
    i,j,pos: integer;
BEGIN
    IF m = 100
    THEN Writeln(' El vector ya está completo')
    ELSE
        BEGIN
            pos := Posicion (w, m, xIns);
            IF pos <> 0
            THEN Writeln('El elemento ',xIns,' ya existe')
            ELSE
                BEGIN
                    m:=m+1;
                    w[m] := xIns;
                    Ordena(w,m);
                END
            END;
END;
(*****
BEGIN
    (* PROGRAMA PRINCIPAL *)
    fin := false;
    REPEAT
        ClrScr;
        Writeln('          MENU:');
        writeln;
        writeln('    1.- Leer elementos del vector');
        writeln('    2.- Escribir el vector');
        writeln('    3.- Ordenar el vector');
        writeln('    4.- Insertar un elemento');
        writeln('    5.- Suprimir un elemento');
        writeln('    6.- Fin');
        writeln;
        REPEAT
            Write('Elige una opción...');
            Readln(opcion);
        UNTIL opcion IN ['1','2','3','4','5','6'];
        CASE opcion OF
            '1': Leervector(v,n);
            '2': EscribeVector(v,n);
            '3': BEGIN
                    EscribeVector(v,n);
                    Ordena(v,n);
                    EscribeVector(v,n);
                END;
            '4': BEGIN
                    Ordena(v,n);
                    write('¿Valor a insertar? ');
                    readln(x);
                    Insertar(v,n,x);
                    EscribeVector(v,n);
                END;
            '5':BEGIN
                    Ordena(v,n);

```


CUESTIONES Y EJERCICIOS RESUELTOS

```
        EscribeVector(v,n);
        Write('¿Valor a suprimir? ');
        Readln(x);
        Borrar(v,n,x);
        EscribeVector(v,n);
        END;
    '6': fin:=true;
END;
UNTIL fin;
END.
```

- 8.18** Escribir un programa para contar el número de veces que se repite cada letra en una frase acabada en punto.

La finalidad de este ejercicio es ilustrar el uso de índices de *arrays* con significado semántico. Esto quiere decir que el subíndice no indica simplemente la posición o número de orden del elemento dentro del array. En este caso, el elemento $x[i]$ contiene el número de veces que se repite en la frase la letra contenida en la variable i , de tipo *char*, que varía de 'a' a 'z'.

Solución

```
PROGRAM Cuentalettras (input,output);
Uses crt;
CONST
    fin='.';
VAR
    i: char;
    x: ARRAY ['a'..'z'] OF integer;
    letra: char;
BEGIN
    (* Inicialización del array de contadores a cero *)
    FOR i:='a' TO 'z' DO x[i]:=0;
    (* Lectura de la frase *)
    Writeln('Escribe una frase acabada en punto (en minúsculas)');
    letra := Readkey;
    Write(letra);
    WHILE letra<>fin DO
        BEGIN
            IF letra<>' '
            THEN x[letra]:=x[letra]+1;
            letra := Readkey;
            Write(letra);
        END;
    (* Escritura del nº de apariciones *)
    Writeln;
    Writeln;
    Writeln ( 'LETRA                Nº REPETICIONES');
    Writeln ( '-----                -----');
    FOR i:='a' TO 'z' DO
        IF x[i]>0
        THEN
            Writeln ( ' ',i,' ',x[i],' ');
    Write('Pulse <Intro> para volver al editor...');
    Readln;
END.
```

- 8.19** Programar en Pascal un juego de preguntas (estilo Trivial Pursuit) con las siguientes reglas:

ESTRUCTURA DE DATOS ARRAY

- El número de jugadores (10 como máximo) será una variable preguntada al usuario al comienzo del juego. Los jugadores se identifican por su número.
- La pregunta será elegida al azar entre un total de 100, y será identificada por su número (no inventar los contenidos de las preguntas).
- Para cada pregunta habrá tres respuestas posibles: *A*, *B* ó *C*. La respuesta de cada jugador a cada pregunta se comparará con un patrón de respuestas correctas. Dicho patrón será generado de manera aleatoria por un subprograma al comienzo del juego. En este subprograma se pueden incluir el resto de las sentencias de inicialización necesarias, como la lectura del número de jugadores.
- Se realizará el mismo número de preguntas a cada jugador. Es decir, si un jugador acierta una respuesta no continúa jugando, sino que la siguiente pregunta será para el siguiente jugador. Al comenzar cada ronda de preguntas se indicará por pantalla su número de orden. También se indicará por pantalla el número del jugador al que corresponde jugar. Cada pregunta aparecerá en pantalla al pulsar **Intro** el jugador correspondiente. El juego acaba cuando el usuario responda *n* ó *N* a un mensaje en pantalla que aparecerá después de cada ronda de preguntas.
- Los resultados aparecerán en pantalla al final del juego en forma de tabla, indicando para cada jugador el número de aciertos y de fallos, así como el ganador del juego (El jugador que acierte mayor número de preguntas).

Nota: Para la generación aleatoria se permite el uso de los subprogramas de Turbo Pascal *Random* y *Randomize*, cuyo funcionamiento es el siguiente:

```
...
Randomize;
...
x := Random;
...
```

La variable *x*, de tipo *real*, tomará un valor aleatorio entre 0 y 1, tal que $0 \leq x < 1$. *Randomize* inicializa la semilla del generador.

Se valorará el uso de subprogramas, así como la claridad, estructuración y brevedad del programa.

Solución

```
PROGRAM JuegoPreguntas(input,output);
CONST
  npreg = 100;           {número maximo de preguntas}
  njmax = 10;           {número maximo de jugadores}
  nletras = 3;         {número de respuestas por pregunta}
TYPE
  patron = ARRAY [1..npreg] OF 'A'..'C';
  aciertos = ARRAY [1..njmax] OF integer;
VAR
  correcta: patron;     {respuestas correctas}
```

CUESTIONES Y EJERCICIOS RESUELTOS

```

    contador: aciertos;    {número de respuestas correctas de
                           cada jugador}
    j, nj, nronda: integer
                           {nj = número de jugadores}
    respu: char;           {nronda = número de preguntas por jugador}
    fin: boolean;
    (*****
PROCEDURE Inicializar(VAR contador:aciertos; VAR correcta:patron;
                      VAR nj:integer);
VAR
    j: integer;
BEGIN
    Writeln ('BIENVENIDOS A NUESTRO JUEGO DE PREGUNTAS');
    Write ('¿Número de jugadores (maximo 10)?');
    Readln (nj);
    FOR j := 1 TO nj DO Contador[j] := 0;
    nronda := 0;
    Randomize;             {Llenado del patron de respuestas correctas}
    FOR j := 1 TO npreg DO
        Correcta[j] := Chr(Ord('A')+Trunc(Random*nletras));
    fin := false;
END;
    (*****
PROCEDURE Pregunta(correcta:patron; VAR cuenta:integer);
VAR
    letra: char;
    preg: 1..100;
BEGIN
    Writeln('; PREPARADO, JUGADOR Número ',j);
    Write('Pulsa <Intro> para jugar...');
    Readln;
    preg := Trunc(Random * npreg) + 1;
    Write ('¿Pregunta ',preg,'? ');
    Readln (letra);
    letra := Ucase(letra);
        (* Se permite aunque es de Turbo Pascal *)
    IF correcta[preg] = letra
    THEN
        BEGIN
            cuenta := cuenta + 1;
            Write ('¡ENHORABUENA, HAS ACERTADO!');
            Writeln (' Preguntas acertadas: ', cuenta);
        END
    ELSE
        BEGIN
            Write ('Lo siento. La respuesta acertadaera',Correcta[preg]);
            Writeln (' Preguntas acertadas: ', cuenta);
        END;
END;
    (*****
PROCEDURE Resultados(contador: aciertos; nj:integer);
FUNCTION Ganador(contador: aciertos; nj: integer):integer;
VAR
    max, j: integer;
BEGIN
    max := contador[1];
    Ganador := 1;
    FOR j:=1 TO nj DO
        IF contador[j] > max
        THEN
            BEGIN
                max := contador[j];
                Ganador := j; (* En caso de empate gana el primero *)
            END;
END;

```

ESTRUCTURA DE DATOS ARRAY

```

BEGIN
  Writeln ('RESULTADOS DEL JUEGO:');
  Writeln ('Número de preguntas por jugador: ',nronda);
  Writeln ('Jugador':12,'Preguntas acertadas':25,'Preguntas
falladas':25);
  FOR j := 1 TO nj DO Writeln (j:8, Contador[j]:18,nronda-
Contador[j]:25);
  Writeln('EL GANADOR ES EL JUGADOR Número ',Ganador(contador,nj));
  Writeln('Pulse <Intro> para continuar');
  Readln;
END;
(*****
BEGIN      (* PROGRAMA PRINCIPAL *)
  Inicializar(contador, correcta, nj);
  WHILE NOT fin DO
    BEGIN
      nronda := nronda+1;
      Writeln ('Ronda de preguntas número ',nronda);
      FOR j:=1 TO nj DO Pregunta(correcta, contador[j]);
      Writeln ('¿Otra pregunta (s/n)?');
      Readln (respu);
      IF respu = 'N' OR respu = 'n' THEN fin := True;
    END;
  Resultados(contador, nj);
END.

```

- 8.20** Escribir un programa que lea una matriz cuadrada e intercambie su primera fila con su diagonal principal.

Solución

```

PROGRAM Diagonal(input,output);
CONST
  m = 100;
TYPE
  indice = 1..m;
  matriz = ARRAY [indice,indice] OF integer;
VAR
  i,j,n:indice;
  a:matriz;
  aux:integer;
(*****
PROCEDURE Leer (num: indice; VAR d: matriz);
VAR
  i,j: indice;
BEGIN
  Writeln;
  Writeln('***** LECTURA DE LOS ELEMENTOS DE LA MATRIZ *****');
  Writeln;
  Writeln;
  FOR i:=1 TO num DO
    FOR j:=1 TO num DO
      BEGIN
        Write ('Dame el elemento ', i:3, ', ', j:3, ': ');
        Readln (d[i,j]);
      END;
    END;
  END; (* Leer *)
(*****

```

CUESTIONES Y EJERCICIOS RESUELTOS

```
PROCEDURE Escribir (num:indice; VAR d: matriz);
VAR
  i,j: indice;
BEGIN
  FOR i:=1 TO num DO
    BEGIN
      Write ('|');
      FOR j:=1 TO num DO
        BEGIN
          Write (d [i,j]:5);
          Write(' ');
        END;
      Writeln ('| ');
    END;
  END; (* Escribir *)
  (*****
  BEGIN
  Write('Dame la dimensión de la matriz ');
  Readln(n);
  Leer(n,a);
  Escribir(n,a);
  (* Intercambio de la primera fila con la diagonal principal *)
  FOR i:=1 TO n DO
    BEGIN
      aux:=a[i,i];
      a[i,i]:=a[1,i];
      a[1,i]:=aux
    END;
  (* Escritura de la matriz modificada *)
  Writeln;
  Writeln('La matriz modificada es:');
  Escribir(n,a);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
  END.
```

- 8.21** Escribir un programa para contar el número de elementos positivos, negativos y nulos de una matriz de números reales.

Solución

```
PROGRAM CuentaElementos (input,output);
CONST
  m = 10;
  l = 10;
  precision = 1E-6; (* para comparar reales *)
TYPE
  filas = 1..m;
  columnas = 1..l;
  matriz = ARRAY [filas, columnas] OF real;
VAR
  i,j,n,k,contapos,contaneg,contanul: integer;
  respu: char;
  a: matriz;
  (*****
  PROCEDURE Leer (f: filas; c:columnas; VAR d:matriz);
  VAR
    i: filas;
    j: columnas;
  BEGIN
    Writeln;
    Writeln('***** LECTURA DE LOS ELEMENTOS DE LA MATRIZ *****');
    Writeln;
    Writeln;
```

ESTRUCTURA DE DATOS ARRAY

```

FOR i:=1 TO f DO
  FOR j:=1 TO c DO
    BEGIN
      Write ('Dame el elemento ', i:3, ', ', j:3, ': ');
      Readln (d[i,j]);
    END;
  END; (* Leer *)
  (*****
PROCEDURE Escribir (f: filas; c:columnas; VAR d: matriz);
VAR
  i: filas;
  j: columnas;
BEGIN
  FOR i:=1 TO f DO
    BEGIN
      Write ('|');
      FOR j:=1 TO c DO
        BEGIN
          Write (d [i,j]:5:1);
          Write(' ');
        END;
        Writeln ('| ');
      END;
    END; (* Escribir *)
  (*****
PROCEDURE Verificar(f: filas; c:columnas; VAR d: matriz);
(* Verificación de la matriz leída *)
VAR
  i: filas;
  j: columnas;
BEGIN
  REPEAT
    Escribir(f, c, d);
    REPEAT
      Writeln;
      Write('¿Existe algún error? (S..si/N..no) ');
      Readln(respu)
    UNTIL (respu='S') OR (respu='s') OR (respu='N') OR (respu='n');
    IF (respu='S') OR (respu='s')
    THEN
      BEGIN
        Write('Dame la fila del elemento erróneo ');
        Readln(i);
        Write('Dame la columna del elemento erróneo ');
        Readln(j);
        Writeln('El valor actual del elemento de la fila',i:3,
          ' y la columna',j:3,' es ',a[i,j]:6:1);
        Writeln;
        Write('Escribe el nuevo valor ');
        Readln(d[i,j])
      END;
    UNTIL (respu='N') OR (respu='n');
  END; (* Verificar *)
  (*****
BEGIN (* Programa principal *)
  Write('Dame el número de filas de la matriz ');
  Readln(n);
  Writeln;
  Write('Dame el número de columnas de la matriz ');
  Readln(k);
  Leer(n, k, a);
  Verificar(n, k, a);
  (* Cálculo del número de elementos positivos,negativos y nulos *)
  contapos:=0;
  contaneg:=0;
  contanul:=0;
  FOR i:=1 TO n DO

```

CUESTIONES Y EJERCICIOS RESUELTOS

```
FOR j:=1 TO k DO
  BEGIN
    IF a[i,j]>0.0
      THEN contapos:=contapos+1;
    IF a[i,j]<0.0
      THEN contaneg:=contaneg+1;
    (* No se deben comparar reales directamente con = *)
    IF Abs(a[i,j]) < precision
      THEN contanul:=contanul+1
    END;
  Writeln;
  Writeln('El número de elementos positivos es ',contapos:3);
  Writeln('El número de elementos negativos es ',contaneg:3);
  Writeln('El número de elementos nulos es ',contanul:3);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

8.22 Realizar un programa que multiplique matrices cuadradas de orden n, usando sub-programas.

Solución

```
PROGRAM ProductoMatrices (input, output);
CONST
  n = 10;
TYPE
  matriz = ARRAY[1..n, 1..n] OF real;
VAR
  m: 1..n;
  a, b, c: matriz;
  (*****)
PROCEDURE Leer (p:integer; VAR d:matriz);
VAR
  i, j: 1..n;
BEGIN
  FOR i:=1 TO p DO
    FOR j:=1 TO p DO
      BEGIN
        Write ('Dame el elemento ', i:3, ', ', j:3, ' : ');
        Readln (d[i,j]);
      END;
    END;
  END; (* Leer *)
  (*****)
PROCEDURE Escribir (p: integer; VAR d: matriz);
VAR
  i, j: 1..n;
BEGIN
  FOR i:=1 TO p DO
    BEGIN
      Write ('|');
      FOR j:=1 TO p DO
        BEGIN
          Write (d [i,j]:5:1);
          Write(' ');
        END;
      Writeln ('| ');
    END;
  END; (* Escribir *)
  (*****)
PROCEDURE Multiplica(p: integer; d1, d2: matriz; VAR prod: matriz);
VAR
  i, j, k: 1..n;
  aux: real;
```

ESTRUCTURA DE DATOS ARRAY

```

BEGIN
  FOR i:=1 TO p DO
    FOR j:=1 TO p DO
      BEGIN
        aux:=0;
        FOR K:=1 TO p DO
          aux := aux + d1[i,k] * d2[k,j];
          prod[i,j]:= aux;
        END;
      END; (* Multiplica *)
    (* ***** *)
  BEGIN (* Programa principal *)
    Writeln ('          PROGRAMA PARA CALCULAR EL PRODUCTO DE');
    Writeln ('          MATRICES CUADRADAS');
    Write ('Introduzca la dimensión de las matrices');
    Readln (m);
    Writeln('Introduzca los datos de la primera matriz:');
    Leer (m,a);
    Writeln('Introduzca los datos de la segunda matriz:');
    Leer (m,b);
    Multiplica (m, a, b, c);
    Writeln('La matriz producto es:');
    Escribir (m,c);
    Write('Pulse <Intro> para volver al editor...');
    Readln;
  END.

```

8.23 Dada una matriz cuadrada de orden n múltiplo de 3, cuyos elementos son números reales, obtener otra de orden $n/3$, donde cada elemento de la segunda se obtiene como valor medio de 9 celdas, tal y como se muestra en la figura 8.6.

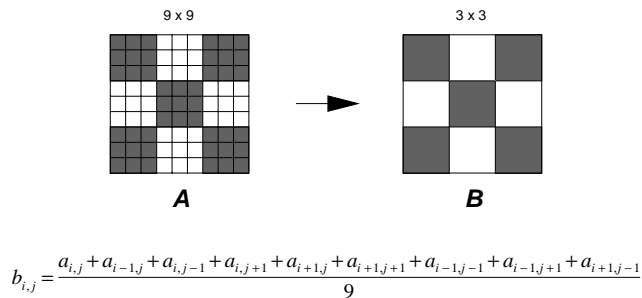


Figura 8.6 Reducción de una matriz

Solución

```

PROGRAM Reduccion (input,output);
CONST
  n = 30;
TYPE
  matriz = ARRAY [1..n,1..n] OF real;
VAR
  s,i,j,k,l: integer;
  a,b: matriz;
(* ***** *)

```


CUESTIONES Y EJERCICIOS RESUELTOS

```

PROCEDURE Leer (num: integer; VAR d: matriz);
VAR
  i,j: integer;
BEGIN
  Writeln;
  Writeln('***** LECTURA DE LOS ELEMENTOS DE LA MATRIZ *****');
  Writeln;
  Writeln;
  FOR i:=1 TO num DO
    FOR j:=1 TO num DO
      BEGIN
        Write ('Dame el elemento ', i:3, ', ', j:3, ': ');
        Readln (d[i,j]);
      END;
    END; (* Leer *)
  (*****)
PROCEDURE Escribir (num: integer; VAR d: matriz);
VAR
  i,j: integer;
BEGIN
  FOR i:=1 TO num DO
    BEGIN
      Write ('|');
      FOR j:=1 TO num DO
        BEGIN
          Write (d [i,j]:5:1);
          Write(' ');
        END;
        Writeln ('| ');
      END;
    END; (* Escribir *)
  (*****)
BEGIN
  REPEAT
    Write ('la dimension de la matriz, multiplo de 3, es : ');
    Readln (s);
    Writeln;
  UNTIL s MOD 3=0;
  Leer(s, a);
  Writeln ('La matriz queda así :');
  Writeln;
  Escribir(s,a);
  FOR k:=1 TO (s DIV 3) DO
    FOR l:=1 TO (s DIV 3) DO
      BEGIN
        b[k,l]:=0;
        FOR i:=(k*3)-2 TO k*3 DO
          FOR j:=(l*3)-2 TO l*3 DO
            b[k,l]:=b[k,l]+a[i,j];
          b[k,l] := b[k,l]/9;
        END;
        Writeln;
        Writeln ('La reducción de la matriz es :');
        Writeln;
        Escribir(s DIV 3, b);
        Write('Pulse <Intro> para volver al editor...');
        Readln;
      END.

```

- 8.24** Diseñar un programa para reproducir las fotos del cometa Halley que son enviadas a la Tierra por la sonda espacial Giotto. Las fotos son enviadas como matrices de números que representan niveles de luminosidad. La resolución será de 80×20 . Los valores de luminosidad se procesarán según la siguiente fórmula:

ESTRUCTURA DE DATOS ARRAY

$$luz_{i,j} = \frac{a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1} + a_{i-1,j-1} + a_{i-1,j+1} + a_{i+1,j-1} + a_{i+1,j+1}}{9}$$

Los valores de las esquinas o límites de la matriz no se procesan.

Solución

```

PROGRAM FotoReproduccion(input, output);
CONST
  m = 20;
  l = 80;
TYPE
  filas = 1..m;
  columnas = 1..l;
  matriz = ARRAY [filas,columnas] OF real;
VAR
  i: filas;
  j: columnas;
  luz :real;
  respu :char;
  a:matriz;
  (*****
PROCEDURE GeneraDatos (f: filas; c:columnas; VAR d:matriz);
CONST
  inf = 0;
  sup = 200;
VAR
  i: filas;
  j: columnas;
BEGIN
  Randomize;
  Writeln;
  Writeln('**** GENERACION ALEATORIA DE LOS DATOS DE LA MATRIZ ****');
  Writeln;
  Writeln;
  FOR i:=1 TO f DO
    FOR j:=1 TO c DO
      d[i,j] := inf + (sup-inf) * Random;
  END; (* GeneraDatos *)
  (*****
PROCEDURE Escribir (f: filas; c:columnas; VAR d: matriz);
VAR
  i: filas;
  j: columnas;
BEGIN
  FOR i:=1 TO f DO
    BEGIN
      Write ('|');
      FOR j:=1 TO c DO
        BEGIN
          Write (d [i,j]:5:1);
          Write(' ');
        END;
      Writeln ('| ');
    END;
  END; (* Escribir *)
  (*****
BEGIN (* Programa principal *)
  (* Para probar el programa generamos aleatoriamente los datos *)
  GeneraDatos(m, l, a);
  Escribir(m, l, a);
  Write('Pulsa <Intro> para continuar...');
  Readln;
  (* Representación de la matriz leida *)

```

CUESTIONES Y EJERCICIOS RESUELTOS

```

Writeln;
FOR i:=2 TO m-1 DO
  BEGIN
  Write('|');
  FOR j:=2 TO l-1 DO
    BEGIN
      luz := (a[i,j] + a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1]
              + a[i-1,j-1] + a[i-1,j+1] + a[i+1,j-1] + a[i+1,j+1])
              / 9;
      IF luz>=150.0 THEN Write(' ');
      IF (luz>=125.0) AND (luz<150.0) THEN Write('.');
      IF (luz>=100.0) AND (luz<125.0) THEN Write('*');
      IF (luz>=75.0) AND (luz<100.0) THEN Write('-');
      IF (luz>=50.0) AND (luz<75.0) THEN Write('#');
      IF (luz>=25.0) AND (luz<50.0) THEN Write('@');
      IF luz<25 THEN Write('%')
    END;
  Write('|');
  END;
END.

```

- 8.25** Escribir un subprograma en Pascal que reciba como argumentos una matriz cuadrada y su dimensión (número impar), y que recorra la matriz, empezando por la casilla central, en forma de espiral, y asigne a los elementos de la matriz el número correspondiente al orden de recorrido. En la figura tenemos un ejemplo de la matriz resultante si la dimensión es 5.

Sugerencias. Colocar primero el elemento central, y luego ir rodeándolo con sucesivas vueltas hasta completar la matriz. Para empezar cada vuelta hay que colocarse primero en su esquina superior derecha, y luego recorrer cuatro tramos de línea recta. En cada vuelta los tramos tienen dos casillas más.

21	22	23	24	25
20	7	8	9	10
19	6	1	2	11
18	5	4	3	12
17	16	15	14	13

Figura 8.7 Recorrido de una matriz en espiral

Solución

```

PROGRAM RecorreEspiral(input,output);
Uses Crt;
TYPE
  matriz = ARRAY[1..15,1..15] OF 1..225;
VAR
  i,j,n: integer;
  A: matriz;
  (*****

```

ESTRUCTURA DE DATOS ARRAY

```

PROCEDURE Espiral(VAR A:matriz; n:integer);
VAR
  i,j,k,orden,lado: integer;
BEGIN
  i := (n DIV 2) + 1;
  j:=i;
  A[i,j]:=1;
  orden:=2;
  lado:=2;
  WHILE lado<=n DO
  BEGIN
    j:=j+1;          (* Avanzando una columna nos situamos al
                    principio de la siguiente vuelta *)
    A[i,j]:=orden;
    orden:=orden+1;
    (* Tramo descendente avanzando filas *)
    FOR k:=1 TO (lado-1) DO
    BEGIN
      i:=i+1;
      A[i,j]:=orden;
      orden:=orden+1;
    END;
    (* Tramo de dcha a izda descendiendo columnas *)
    FOR k:=1 TO lado DO
    BEGIN
      j:=j-1;
      A[i,j]:=orden;
      orden:=orden+1;
    END;
    (* Tramo de abajo a arriba disminuyendo filas *)
    FOR k:=1 TO lado DO
    BEGIN
      i:=i-1;
      A[i,j]:=orden;
      orden:=orden+1;
    END;
    (* Tramo de izda a dcha avanzando columnas ---->*)
    FOR k:=1 TO lado DO
    BEGIN
      j:=j+1;
      A[i,j]:=orden;
      orden:=orden+1;
    END;
    lado :=lado+2;
  END;  (* WHILE *)
END;  (* Espiral *)
(*****)
BEGIN
Write('¿dimensión (número impar)?');Readln(n);
Espiral(A,n);
FOR i:=1 TO n DO
  BEGIN
    FOR j:=1 TO n DO
      Write(A[i,j]:4);
    Writeln;
  END;
Readln;
END.

```

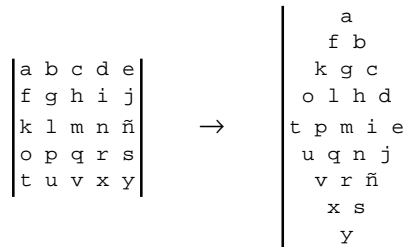
8.26 Realizar un programa que lea una matriz cuadrada y la gire 45° de la siguiente forma:

CUESTIONES Y EJERCICIOS RESUELTOS

- lee la matriz cuadrada de orden impar, n .
- los elementos de la matriz son caracteres.
- escribe una matriz de orden $2n-1$, que contiene a la matriz anterior girada, tal y como se muestra a continuación:



Análisis: Si la matriz inicial tiene dimensión n , la dimensión de la matriz girada será $2n-1$. Por ejemplo, con una matriz de 5 filas y 5 columnas tenemos:



Llamaremos i, j a la posición de un elemento original, y f, c a la posición del mismo elemento en la matriz girada. Puede comprobarse que:

$$\begin{aligned} f &= j + i - 1 \\ c &= j - i + n \end{aligned}$$

Solución

```
PROGRAM Rombo (input , output);
CONST
  maxA=19;      (* Máximo orden de la matriz A *)
  maxB=37;      (* Máximo orden de la matriz B = 2 * maxA - 1 *)
TYPE
  indice = 1..maxB;
  matriz = ARRAY [indice,indice] OF char;
VAR
  fila, columna, n, n2: indice;
  q, r, h: integer;
  a, b: matriz;
  (*****)
PROCEDURE Leer (num: indice; VAR d: matriz);
VAR
  i, j: indice;
BEGIN
  Writeln;
  Writeln('*** LECTURA DE LOS ELEMENTOS DE LA MATRIZ ***');
  Writeln;
  Writeln;
  FOR i:=1 TO num DO
```

ESTRUCTURA DE DATOS ARRAY

```

FOR j:=1 TO num DO
  BEGIN
    Write ('Dame el elemento ', i:3, ', ', j:3, ': ');
    Readln (d[i,j]);
  END;
END; (* Leer *)
(*****
PROCEDURE  Escribir (num:indice; VAR d: matriz);
VAR
  i,j: indice;
BEGIN
  FOR i:=1 TO num DO
    BEGIN
      Write ('|');
      FOR j:=1 TO num DO
        BEGIN
          Write (d [i,j]:5);
          Write(' ');
        END;
      Writeln ('| ');
    END;
  END; (* Escribir *)
(*****
  BEGIN
    REPEAT      (*obtiene el orden de la matriz*)
      Writeln ('Dame el orden de la matriz');
      Writeln ('Recuerda que ha de ser impar y menor que ', maxA);
      Write ('Por pantalla salen bien si orden< 7:');
      Readln(n);
      UNTIL (n MOD 2 <> 0) AND (n < maxA);
      Leer(n,a);
    (* Inicializamos la nueva matriz a blancos *)
      n2 := 2 * n - 1 ;
      FOR fila := 1 TO n2 DO
        FOR columna := 1 TO n2 DO
          b[fila , columna] := ' ';
    (* Giramos la matriz inicial 45 grados *)
      q:=n;
      r:=0;
      h:=0;
      FOR fila := 1 TO n DO
        BEGIN
          h:=q;
          FOR columna := 1 TO n DO
            BEGIN
              b[columna + r, h]:= a[fila , columna];
              h:=h+1;
            END;
            q:=q - 1;
            r:=r + 1;
          END;
          Writeln;
          Writeln;
          Writeln('Matriz inicial:');
          Escribir(n, a);
          Writeln('Matriz girada 45 grados:');
          Escribir(n2,b);
          Write('Pulse <Intro> para volver al editor...');
          Readln;
        END.

```

8.27 Escribir un programa que pase a letras mayúsculas una cadena de caracteres (codigo ASCII o EBCDIC)

CUESTIONES Y EJERCICIOS RESUELTOS

Solución

```
PROGRAM Escribe_Mayusculas (input,output);
(* Este programa pasa a mayúsculas una cadena de caracteres,
   aunque deja sin pasar las vocales acentuadas y la ñ. *)
(* Ejemplo del manejo de string *)
TYPE
  string80 = STRING [80];
VAR
  cadena:string80;
  (*****)
PROCEDURE pasa_a_mayusculas(VAR a:string80);
VAR
  i:integer;
BEGIN
  FOR i:=1 TO Ord(a[0]) DO
    a[i]:= Uppcase(a[i])
  END;
  (*****)
BEGIN
  Write ('Introduzca una cadena de caracteres : ');
  Readln(cadena);
  pasa_a_mayusculas(cadena);
  Writeln;
  Writeln('En mayúsculas : ',cadena);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

- 8.28 Ampliar el programa anterior para que pase los caracteres especiales específicos del Castellano (vocales acentuadas y la ñ).

Solución

```
PROGRAM Escribe_Todo_a_Mayusculas (input,output);
(* Este programa pasa a mayúsculas una cadena de caracteres,
   incluyendo las vocales acentuadas y la ñ. *)
(* Ejemplo del manejo de string *)
TYPE
  string80 = STRING [80];
VAR
  cadena:string80;
  (*****)
PROCEDURE pasa_a_mayusculas(VAR a:string80);
VAR
  i:integer;
BEGIN
  FOR i:=1 TO Ord(a[0]) DO
    IF (a[i]='á') OR (a[i]='é') OR (a[i]='í')
      OR (a[i]='ó') OR (a[i]='ú') OR (a[i]='ñ')
      (* Esta expresión puede simplificarse utilizando conjuntos *)
    THEN
      CASE a[i] OF
        'á': a[i]:='A';
        'é': a[i]:='E';
        'í': a[i]:='I';
        'ó': a[i]:='O';
        'ú': a[i]:='U';
        'ñ': a[i]:='Ñ'
      END
    ELSE
      a[i]:= Uppcase(a[i]);
    END;
  END;
END;
```

ESTRUCTURA DE DATOS ARRAY

```
(*****)  
BEGIN  
  Write ('Introduzca una cadena de caracteres : ');  
  Readln(cadena);  
  pasa_a_mayusculas(cadena);  
  Writeln;  
  Writeln('En mayúsculas : ',cadena);  
  Write('Pulse <Intro> para volver al editor...');  
  Readln;  
END.
```

8.29 Realizar un programa que ordene por orden alfabético un *array* de *strings*.

Solución

```
PROGRAM Ordenacion_por_orden_alfabetico (input,output);  
(* Este programa ordena por orden alfabético un array de string,  
para evitar problemas con las vocales acentuadas, se almacenan  
las cadenas de caracteres en mayúsculas. *)  
(* La Ñ se considera posterior a la z *)  
(* Ejemplo del manejo de string *)  
CONST  
  n = 100;      (* número máximo de cadenas a ordenar *)  
  punto='.';   (* cadena de parada*)  
TYPE  
  string80 = STRING [80];  
  vector = ARRAY [1..n] OF string80;  
VAR  
  c:vector;  
  k:integer;  
(*****)  
PROCEDURE pasa_a_mayusculas(VAR a:string80);  
VAR  
  i:integer;  
BEGIN  
  FOR i:=1 TO Ord(a[0]) DO  
    IF (a[i]='á') OR (a[i]='é') OR (a[i]='í')  
      OR (a[i]='ó') OR (a[i]='ú') OR (a[i]='ñ')  
    THEN  
      CASE a[i] OF  
        'á': a[i]:='A';  
        'é': a[i]:='E';  
        'í': a[i]:='I';  
        'ó': a[i]:='O';  
        'ú': a[i]:='U';  
        'ñ': a[i]:='Ñ';  
      END  
    ELSE  
      a[i]:= Ucase(a[i]);  
    END;  
END;  
(*****)  
PROCEDURE lee_cadena (VAR cad:string80);  
BEGIN  
  Writeln ('Introduzca una cadena de caracteres');  
  Write ('(o punto para terminar) --> ');  
  Readln(cad);  
  pasa_a_mayusculas(cad)  
END;  
(*****)
```


EJERCICIOS PROPUESTOS

```
PROCEDURE lee_vector_de_cadenas(VAR i:integer;VAR b:vector);
(* Lectura de las cadenas de caracteres *)
(* Devuelve el vector, y el número de elementos introducidos *)
VAR
  cad:string80;
BEGIN
  i:=1;
  REPEAT
    lee_cadena(cad);
    b[i]:=cad;
    i:=i+1;
  UNTIL (i=n+1) OR (cad=punto);
  IF i=n+1 THEN i:=n ELSE i:=i-2
END;
(*****)
PROCEDURE ordena_vector_cadenas (m:integer;VAR b:vector);
(* Ordenación por orden alfabetico, método de la burbuja *)
(* Se introduce el vector desordenado y el número de elementos ocupa-
dos *)
(* Sale el vector ordenado alfabeticamente *)
(* La ñ se considera posterior a la z *)
VAR
  i,j:integer;
  aux:string80;
BEGIN
  FOR j:=2 TO m DO
    FOR i:=m DOWNTO j DO
      IF b[i-1]>b[i] THEN
        BEGIN
          aux:=b[i];
          b[i]:=b[i-1];
          b[i-1]:=aux
        END;
      END;
    END;
  END;
(*****)
PROCEDURE escribe_vector(m:integer;b:vector);
VAR
  i:integer;
BEGIN
  Writeln;
  FOR i:=1 TO m DO Writeln(b[i])
END;
(*****)
(* Programa Principal *)
BEGIN
  lee_vector_de_cadenas(k,c);
  ordena_vector_cadenas(k,c);
  escribe_vector(k,c);
  Write('Pulse <Intro> para volver al editor...');
  Readln;
END.
```

8.13 EJERCICIOS PROPUESTOS

8.30 El producto escalar de dos vectores A y B de dimensión n , viene dado por la fórmula:

$$A \cdot B = \sum_{i=1}^n a_i \cdot b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n$$

a) Diseñar una función con tres parámetros: A , B y n , que calcule el producto escalar según esta fórmula.

ESTRUCTURA DE DATOS ARRAY

b) Si el producto escalar es cero, se dice que los vectores son *ortogonales*. Escribir un programa que, utilizando la función anterior, nos indique si dos vectores son ortogonales.

8.31 Escribir un programa que tome la temperatura de las 24 horas de un día y nos de la media, máxima y mínima, utilizando subprogramas.

8.32 Realizar un programa que dada una cantidad comprendida entre 1 y 99.999.999 la exprese en letra. Se puede utilizar la extensión *string*.

8.33 Diseñar un programa que realice las siguientes operaciones con una matriz cuadrada, seleccionadas desde un menú:

- a) Cálculo de la matriz transpuesta.
- b) Producto de matrices.
- c) Suma de las filas de la matriz.
- d) Suma de las columnas de la matriz.
- e) Cálculo de la matriz inversa.

8.34 Sea la declaración:

```
TYPE
  t = ARRAY[-1000..1000] OF 'a'..'z';
VAR
  x: t;
```

Escribir un procedimiento que, tras la llamada `control(x)`, nos escriba el número de elementos del *array* `x` igual a cada letra minúscula. Efectúe las declaraciones pensando que el procedimiento no tiene que devolver ningún resultado, solo imprimir lo antes pedido.

8.35 Sea la declaración:

```
CONST
  n = 100;
TYPE
  t = ARRAY [1..n] OF char;
VAR
  x: t;
```

Escribir un fragmento de programa para averiguar si todos los elementos de la matriz `x` son dígitos del '0' al '9'.

EJERCICIOS PROPUESTOS

8.36 Escribir una función booleana `chequeo` que nos indique si una sucesión de m caracteres, almacenada en el vector `b` se encuentra en el vector `a` de n caracteres.

8.37 Escribir un programa para imprimir los *puntos de silla* de una matriz de números reales. Llamamos *puntos de silla* a los elementos que son a la vez máximo de su fila y mínimo de su columna.

8.38 Partiendo de las declaraciones:

```

CONST
    max = 10;
    necmax = 10;
TYPE
    vector = ARRAY [0..max] OF real;
    matriz = ARRAY[1..necmax] OF vector;
    
```

Vamos a representar un polinomio mediante una variable de tipo `vector` (el elemento que ocupa la posición i será el coeficiente de grado i del polinomio), y un sistema de ecuaciones polinómicas mediante una variable de tipo `matriz`. Se pide:

a) Escribir un subprograma que reciba un polinomio como argumento, y devuelva al punto de llamada la derivada de dicho polinomio.

b) Escribir un subprograma que reciba un sistema de ecuaciones como argumento e imprima por *output* el sistema formado por las derivadas de las ecuaciones del sistema de partida, utilizando el subprograma anterior.

8.39 Una *pila* es una estructura de almacenamiento de datos tal que, una vez llena, cada dato entrante produce la salida del primero que entró entre los restantes.

Como ejemplo, si tenemos una pila con capacidad de tres datos, representando el orden de entrada en lugar del dato entrante, su contenido será:

posición	1	2	3	fuera
paso 1	1	x	x	x
paso 2	2	1	x	x
paso 3	3	2	1	x
paso 4	4	3	2	1
...
paso n	n	n-1	n-2	n-3

x = Elemento sin dato (indeterminado)

ESTRUCTURA DE DATOS ARRAY

Construir una pila de caracteres de 20 datos de capacidad, que almacene datos de entrada, leídos por teclado. Imprimir cada dato que sale fuera y la pila completa al cabo de 40 lecturas de datos.

8.40 Dadas las declaraciones:

```
CONST
    m = 100;
TYPE
    fechas = string[6];
    vectorFechas = ARRAY[1..m] OF fechas;
```

Escribir un subprograma que ordene de *anterior* a *posterior* una variable de tipo `vectorFechas`. Se suponen las fechas en formato *ddmmaa*, donde *dd* representa el día, *mm* el mes y *aa* el año.

Nota: Si se transforman las fechas a formato *aammdd*, el orden alfabético de los *strings* coincide con el orden temporal de las fechas.

8.41 El análisis léxico, que forma parte del proceso de compilación permite reconocer las unidades sintácticas o *tokens* de un programa (identificadores, palabras reservadas, etc). Un analizador léxico se puede programar mediante un autómata finito, que es un dispositivo formal que decide si un *token* pertenece o no al lenguaje.

El autómata finito se define por:

- Un conjunto de estados $Q = \{q_0, q_1, \dots, q_N\}$, donde q_0 es el estado inicial.
- Un conjunto de estados finales que pertenecen a Q .
- Un alfabeto E , que describe al lenguaje de programación
- Una función de transición que permite pasar de un estado a otro, y se define como:

$$f: Q \times E \rightarrow Q$$

Si se considera un lenguaje formado solo por los caracteres *a* y *b*, es decir $E = \{a, b\}$, un *token* formado por caracteres *a* o caracteres *b* o caracteres *a* y *b* pertenece al lenguaje si, partiendo del estado inicial q_0 y leyendo el *token* caracter a caracter se llega a un estado final según la tabla de transición siguiente:

	a	b
q_0	q_1	q_2
q_1	q_2	q_1
q_2	q_2	q_2

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

donde q_0 es estado inicial y q_2 es estado final.

Se debe tener en cuenta que el alfabeto de entrada son las columnas y los estados las filas de un array bidimensional.

Ejemplos:

- El *token* aa pertenece al lenguaje, ya que:

$$f(q_0, a) = q_1; f(q_1, a) = q_2 \rightarrow \text{estado final}$$

Explicación: Se lee el primer caracter del *token* desde el estado inicial; como la posición $[q_0, a]$ del array vale q_1 , se pasa al estado q_1 ; se lee el siguiente caracter desde el estado q_1 ; como la posición $[q_1, a]$ del array es q_2 , se pasa al estado q_2 . Como es estado final y no hay más caracteres en el *token*, el proceso finaliza perteneciendo el *token* al lenguaje.

- El *token* abb no pertenece al lenguaje ya que:

$$f(q_0, a) = q_1; f(q_1, b) = q_1; f(q_1, b) = q_1 \rightarrow \text{estado no final}$$

Explicación: Es análoga a la anterior, pero ahora cuando se acaban de leer los caracteres del *token* se llega a un estado q_1 que no es estado final, por tanto el *token* no pertenece al lenguaje.

Se pide realizar un programa que compruebe, mediante procedimientos y funciones, si un *token* (cadena de caracteres a y b) leído por teclado pertenece al lenguaje definido o no.

Nota: Se debe inicializar el array bidimensional con el contenido de la tabla de transiciones.

8.14 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Para profundizar en el tema de los tipos abstractos de datos, recomendamos consultar la obra *Introducción a la computación con Turbo Pascal*, de W. I. Salmon, editada por Addison Wesley Iberoamericana en 1993. Esta obra dedica un capítulo completo a los arrays (traducido por arreglos) como tipos abstractos de datos, ilustrado con varios ejemplos prácticos con cadenas y gráficos de tortuga. También tiene un capítulo dedicado al tipo abstracto de datos *vector* el libro titulado *Estructuras de datos, realización en Pascal* de M. Collado Machuca, R. Morales Fernández, y J.J. Moreno Navarro, publicado por Ediciones Díaz de Santos (1987).

Los algoritmos de ordenación pueden estudiarse de forma exhaustiva en el capítulo titulado *Ordenación* del libro *Algoritmos + estructuras de datos = programas* de N. Wirth (Ed. del Castillo, 1980). También se pueden estudiar en el capítulo titulado *Clasificación* del libro *Estructuras de datos y algoritmos* de A.V. Aho, J.E. Hopcroft y J. D. Ullman (Ed. Addison-Wesley, 1988). Si los libros anteriores son demasiado escuetos en sus explicaciones para el nivel de comprensión del

lector, puede consultarse la obra *Estructuras de datos* de *O. Cairó* y *S. Guardati* (Ed. McGraw-Hill, 1993). Como obra totalmente dedicada al tema de ordenación, y pionera históricamente puede consultarse *El arte de programar ordenadores, volumen III: clasificación y búsqueda* de *D. Knuth*, en la editorial Reverté (1987).

Respecto a las aplicaciones al Cálculo Numérico, tanto para el método de Gauss-Jordan como para el método de cuadratura de *Gauss* recomendamos las obras ya citadas en el capítulo 6 en el apartado de ampliaciones y notas bibliográficas. Si se desearan más constantes para el método de cuadratura de *Gauss*, así como otras fórmulas o tablas matemáticas, puede consultarse la obra *Handbook of mathematical functions with formulas, graphs, and mathematical tables* por *M. Abramowitz* e *I. Stegun* (Ed. Dover, 1965).

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS



CAPITULO 9

CONJUNTOS

CONTENIDOS

- 9.1 Conjuntos en Pascal
- 9.2 Construcción de conjuntos
- 9.3 Operaciones con conjuntos
- 9.4 Expresiones lógicas con conjuntos
- 9.5 Representación interna de los conjuntos
- 9.6 Ejercicios resueltos
- 9.7 Cuestiones y ejercicios propuestos
- 9.8 Ampliaciones y notas bibliográficas

9.1 CONJUNTOS EN PASCAL

Un conjunto es un tipo de datos estructurado incorporado por el lenguaje Pascal del que carecen la mayoría de los lenguajes de programación. Un conjunto (*set*) es una colección de elementos de un mismo tipo, denominado *tipo base*, sin que exista ninguna relación de orden entre ellos.

CONSTRUCCION DE CONJUNTOS

Aunque el Pascal, en sí, no imponga ninguna limitación al tamaño máximo de un conjunto, las diferentes implementaciones, limitan el número de elementos permitidos en el tipo base, con el fin de que los programas sean más eficientes. Este número oscila habitualmente entre 64 y 256. El compilador Turbo Pascal permite hasta 256 elementos.

Para declarar un tipo conjunto en Pascal se procede de la siguiente manera:

```
TYPE
    conjunto = SET OF tipo-base;
```

La sintaxis en notación EBNF es de la forma:

```
<tipo conjunto> ::= SET OF <tipo simple>
```

Algunos ejemplos de declaraciones de tipos y variables conjunto se muestran a continuación:

```
TYPE
    conjuntoNumeros = SET OF 1..50 ;
    conjuntoLetras  = SET OF 'A'..'Z' ;

VAR
    pares, primos      : conjuntoNumeros ;
    vocales, consonantes : conjuntoLetras ;
```

Las variables de tipo conjunto que acabamos de declarar, al igual que todo tipo de variables, están indefinidas hasta que se les asigne algún valor en el programa. Es pues un error pensar que en principio se encuentran inicializadas a conjunto vacío.

Antes de estudiar las operaciones que se pueden realizar con conjuntos, vamos a recordar algunas definiciones:

- Conjunto universal** es el conjunto que contiene todos los valores del tipo base.
- Conjunto vacío** es un conjunto que no contiene ningún elemento.
- Subconjunto** sean A y B dos conjuntos, decimos que A es un subconjunto de B si todos los elementos de A pertenecen también a B.

9.2 CONSTRUCCION DE CONJUNTOS

Un conjunto se construye a partir de constantes de su tipo base. Para ello se escriben los elementos componentes uno a continuación de otro, encerrados entre corchetes y separados entre sí por comas.

A continuación se muestran dos ejemplos de conjuntos; uno del tipo `conjuntoNumeros` y otro del tipo `conjuntoLetras` definidos antes:

```
[1,5,6,8]
['A','E','M']
```

El orden en que se enumeran los elementos de un conjunto es indiferente.

Un conjunto puede contener un solo elemento como se muestra a continuación.

```
[5]           o           ['W']
```

CONJUNTOS

También se pueden construir conjuntos a partir de variables. Así teniendo en cuenta las declaraciones anteriores se puede poner el siguiente fragmento de código

```
...
VAR
  a, b: 'A' .. 'Z';
...
  a := 'F';
  b := 'G';
  consonantes := [a,b];
  vocales := ['A','E','I','O','U'];
...
```

El conjunto [a,b] contendría dos elementos cuyos valores serían los que tuvieran asignados las variables a y b en ese momento.

Un caso especial lo constituye el *conjunto vacío*, el cual es compatible con cualquier tipo conjunto. Su sintaxis es:

```
[] (* conjunto vacío *)
```

Estos conjuntos que acabamos de ver, pueden ser asignados a variables conjunto de sus respectivos tipos, o participar en expresiones lógicas como veremos posteriormente.

Algunas sentencias de asignación pueden ser:

```
pares := [2,4,6,8] ;
vocales := ['A','E','I','O','U'] ;
```

A la hora de especificar los elementos de un conjunto, podemos abreviar representándolos como un subrango si varios de ellos son consecutivos. Así, los conjuntos:

```
['A','B','C','M','X','Y','Z']
['A'..'C','M','X'..'Z']
```

son equivalentes.

Los siguientes conjuntos

```
['C'..'A'] y [8..1]
```

se consideran vacíos, pues el primer elemento del subrango es mayor que el segundo.

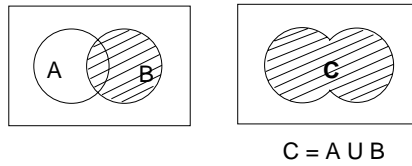
9.3 OPERACIONES CON CONJUNTOS

Las tres operaciones que podemos realizar con conjuntos son: unión, intersección y diferencia de conjuntos.

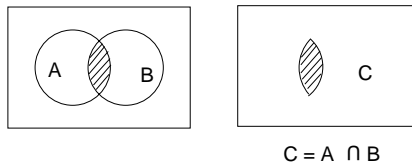
En las figuras siguientes se muestra gráficamente el resultado de estas operaciones aplicadas a dos conjuntos A y B.

OPERACIONES CON CONJUNTOS

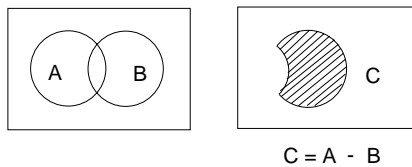
UNION



INTERSECCION



DIFERENCIA



Los operadores usados en Pascal para denotar dichas operaciones se muestran a continuación:

Operador	Significado
+	unión de conjuntos
*	intersección de conjuntos
-	diferencia de dos conjuntos.

Tabla 9.1 Operaciones con conjuntos

Veamos algunos ejemplos:

```
vocales := ['A','E','I','O'];
vocales := vocales + ['U']; (* unión *)
```

la variable `vocales` contiene ahora las cinco letras vocales.

```
primos := [1,2,4,5] * [1,2,5,8];
```

hace que `primos` tome el valor: `[1,2,5]`

Análogamente:

```
primos := [1,2,5,8] - [4,6,8];
```

asigna a `primos` el conjunto `[1,2,5]` pues esos son los elementos que pertenecen al primer conjunto pero no pertenecen al segundo.

9.4 EXPRESIONES LOGICAS CON CONJUNTOS

Los operadores de relación pueden usarse con conjuntos para formar expresiones booleanas. Estos operadores adquieren aquí un significado especial.

A continuación se muestran los cuatro operadores relacionales que son aplicables a conjuntos, indicando la interpretación que se le da a cada uno de ellos.

Operador	Significado
=	Igualdad de conjuntos.
<>	Desigualdad de conjuntos.
<=	Inclusión el primer conjunto está incluido en el segundo.
>=	Inclusión el segundo conjunto está incluido en el primero.

Tabla 9.2 Operaciones lógicas con conjuntos

Los conjuntos a comparar deberán ser del mismo tipo.

Se observa que, debido a la interpretación particular que se les da a los operadores relacionales entre conjuntos, puede suceder que las expresiones: $A \geq B$ y $A \leq B$; sean ambas falsas!. Basta con que A y B sean conjuntos disjuntos.

Esto no puede ocurrir con otro tipo de variables.

Las comparaciones que hemos visto trabajan con conjuntos completos. Pero el Pascal incluye un nuevo operador relacional el **operador IN** que sirve para comprobar la pertenencia de un determinado elemento a un conjunto.

La sintaxis de una expresión booleana utilizando el **operador IN** se muestra a continuación:

```
elemento IN conjunto
```

y devolverá un valor *true* si el *elemento* pertenece al *conjunto*. El elemento especificado deberá ser una constante, variable o expresión del mismo tipo que el tipo base del conjunto.

A continuación se ilustra el uso de este operador:

```
5 IN [4,5,8]
'A' IN vocales
(2*x+1) IN [4..10]
x IN primos+pares
```

La primera devuelve *true*. La segunda devuelve *true* suponiendo que `vocales` tiene el valor: `['A','E','I','O','U']`. La tercera devuelve *true* si el resultado de la expresión $2*x+1$ está comprendido entre 4 y 10 ; (suponemos x de tipo *integer*). Finalmente, la última expresión será *true* si la *unión* de los conjuntos `primos` y `pares` contiene el valor de la variable x de tipo *integer*.

REPRESENTACION INTERNA DE LOS CONJUNTOS

Este tipo de expresiones con el operador *IN* se usan frecuentemente en conjunción con estructuras de control del tipo *IF*, *WHILE* y *REPEAT*.

A continuación se muestran dos casos típicos:

```
...
VAR
ch : char ;
...
IF ch IN ['A','E','I','O','U']
  THEN Writeln ('Es una vocal mayúscula');
...
```

que es mucho más corto que escribir:

```
IF (ch='A') OR (ch='E') OR (ch='I') OR
   (ch='O') OR (ch='U')
  THEN Writeln ('Es una vocal mayúscula');
```

En ocasiones, en uso interactivo, queremos que se nos responda *sí* o *no* a una determinada pregunta en un determinado instante de un programa, tecleando una *S* o una *N* mayúscula o minúscula indistintamente. Para asegurarnos de que la tecla que se pulse sea exactamente una de esas y no admitir ninguna otra como válida podemos proceder así:

```
REPEAT
  Write ('¿Le está gustando el lenguaje Pascal? (S/N)');
  Readln (ch) ;
UNTIL ch IN ['S','s','N','n'] ;
IF ch IN ['S','s']
  THEN Writeln ('¡Me alegro!')
  ELSE Writeln ('Espero que cambie');
```

mientras no se pulse una *s* o una *n* (mayúscula o minúscula, indistintamente), no se saldrá del bucle *REPEAT*.

9.5 REPRESENTACION INTERNA DE LOS CONJUNTOS

Los conjuntos se representan internamente en el ordenador como un array de *bits*, con tantos *bits* como el número máximo de elementos que se puede definir en un conjunto (habitualmente un múltiplo de 2 entre 64 y 256). Si el conjunto está vacío todos los *bits* están a cero. Si el *i-ésimo* es uno, entonces el elemento cuyo ordinal es *i* está incluido en el conjunto. La ventaja principal de esta representación radica en que las operaciones de pertenencia, inserción de un elemento o supresión de un elemento se pueden realizar en un tiempo constante, mediante una referencia directa al *bit* apropiado. Mientras que las operaciones de unión, intersección y diferencia se realizan en un tiempo proporcional al número máximo posible de elementos del conjunto.

En el compilador Turbo Pascal el número máximo de elementos de un conjunto es 256, por tanto un conjunto nunca ocupa más de 32 bytes. El número de bytes que ocupa un conjunto determinado se calcula como:

$$\text{Tamaño (en bytes)} = (\text{Max DIV } 8) - (\text{Min DIV } 8) + 1$$

donde *Min* y *Max* son los límites inferior y superior del tipo base de ese conjunto. El número del byte donde está el elemento *i-ésimo* del conjunto es:

$$\text{Número de byte} = (i \text{ DIV } 8) - (\text{Min DIV } 8)$$

y el número de bit dentro del byte anterior es:

$$\text{Número de bit} = i \text{ MOD } 8$$

9.6 EJERCICIOS RESUELTOS

- 9.1** Realizar un programa que lea un texto por teclado, e indique si aparecen todas las vocales en mayúsculas o no aparecen.

Se presionará la tecla **Intro** al finalizar el texto de entrada.

Solución

Algoritmo

```

INICIO
  MIENTRAS existan caracteres
    Leer carácter
    SI es una letra del alfabeto
      ENTONCES
        Incorporar el carácter leído al conjunto
      FIN_SI
    SI el conjunto es >= al conjunto de vocales en
      mayúscula
      ENTONCES
        Escribir sí aparecen
      SI_NO
        Escribir no aparecen
      FIN_SI
    FIN_MIENTRAS
  FIN
  
```

Codificación en Pascal

```

PROGRAM Vocales (input,output);
VAR
  letra : char;          (* Carácter que se lee *)
  conjunto :SET OF 'A'..'Z';  (* Conjunto de letras del texto *)
BEGIN
  (* Inicialización a vacío del conjunto de letras del texto *)
  conjunto:=[];

  (* Bucle de lectura del texto y llenado del conjunto que contiene
  las letras del texto dado *)

  WHILE NOT Eoln DO  (* fin de línea del fichero input *)
  BEGIN
    Read(letra);

    (* Conversión de las letras minúsculas a mayúsculas, pues sólo
    hemos definido un conjunto de letras mayúsculas. *)
  
```

EJERCICIOS RESUELTOS

```
IF letra IN ['a'..'z']
  THEN letra:=Chr(Ord(letra)-Ord('a')+Ord('A'));

(* Incorporación del carácter leído al conjunto. *)

conjunto:=conjunto+[letra]
END;

(* Si el conjunto de letras del texto incluye al subconjunto de las
vocales, entonces contiene todas las vocales. *)

Writeln;
IF conjunto>=['A','E','I','O','U']
  THEN Writeln('Contiene todas las vocales')
  ELSE Writeln('Le falta alguna vocal');
END.
```

- 9.2** Realizar un programa que lea una línea de texto por teclado y escriba cada uno de los caracteres diferentes que se hayan utilizado.

También se pide obtener un listado ordenado de todas las letras minúsculas del alfabeto español que hayan sido empleadas, indicando además si el texto contiene alguna vocal acentuada.

Solución. El algoritmo es el siguiente:

```
INICIO
  MIENTRAS existan caracteres
    Leer carácter
    Incorporar el carácter leído al conjunto
  FIN_MIENTRAS
  PARA cada carácter del código ASCII extendido
    SI aparece en el conjunto
      ENTONCES
        Escribirlo
    FIN_SI
    SI la intersección del conjunto de vocales acen-
    tuadas con el conjunto <> del vacío
      ENTONCES
        Escribir las vocales acentuadas
    FIN_SI
  FIN_PARA
FIN
```

Codificación en Pascal

```
PROGRAM Caracteres(input,output);
TYPE
  conjunto=SET OF char;
VAR
  usadas:conjunto;
  ch:char;
```

CONJUNTOS

```
BEGIN
  usadas:=[];
  Writeln(' Introduzca una línea de texto:');
  WHILE NOT Eoln DO
    BEGIN
      Read(ch);
      usadas:=usadas+[ch];
    END;
  Readln;
  Writeln;
  Writeln(' Listado de caracteres diferentes:');
  FOR ch:=Chr(0) TO Chr(255) DO
    IF ch IN usadas
      THEN Write(ch, ' ');
  Writeln;
  Writeln(' Listado de minúsculas y caracteres propios del
    español:');
  FOR ch:='a' TO 'n' DO
    IF ch IN usadas
      THEN Write(ch, ' ');
  IF 'ñ' IN usadas
    THEN Write('ñ ');
  FOR ch:='o' TO 'z' DO
    IF ch IN usadas
      THEN Write(ch, ' ');
  Writeln;
  IF ['á','é','í','ó','ú']*usadas<>[]
    THEN BEGIN
      Writeln('El texto contiene las siguientes vocales
        acentuadas:');
      IF 'á' IN usadas
        THEN Write('á ');
      IF 'é' IN usadas
        THEN Write('é ');
      IF 'í' IN usadas
        THEN Write('í ');
      IF 'ó' IN usadas
        THEN Write('ó ');
      IF 'ú' IN usadas
        THEN Write('ú ');
      END
    ELSE Writeln('La línea no contiene ninguna vocal acentuada');
  END.
```

9.3 Realizar una función que devuelva como resultado el número de elementos que contiene una variable de tipo conjunto. Considere el tipo conjunto definido como:

```
TYPE
  tipoConjunto= SET OF 1..49;
```

Solución

Algoritmo

```
INICIO
  PARA i = 1 HASTA i = 49
    SI el valor de i está en el conjunto
      ENTONCES
        Incrementar contador de elementos
      FIN_SI
  FIN_PARA
FIN
```


EJERCICIOS RESUELTOS

Codificación en Pascal

```
FUNCTION Cardinal (conjunto: tipoConjunto): integer;
VAR
  i, cont: integer;
BEGIN
  cont:=0;
  FOR i:=1 TO 49 DO
    IF i IN conjunto
      THEN cont:=cont+1;
    Cardinal:=cont;
  END; {Cardinal}
```

9.4 Un boleto de Lotería Primitiva está formado por 8 bloques de apuestas de 49 números cada uno, de los cuales se deben tachar 6 en cada bloque.

Se pide:

- Diseñar la estructura de datos más apropiada para almacenar un boleto de ocho apuestas.
- Realizar un procedimiento que, recibiendo como parámetros el boleto, la combinación ganadora y el número complementario, determine las posibles apuestas premiadas, indicando el número de bloque en que se producen, así como el número de aciertos.
- Realizar otro procedimiento que escriba por *output* la relación de números que no hayan sido tachados en ninguna de las apuestas, y en base a ello determine si existe la seguridad de acertar al menos un número en alguna de las apuestas, sea cual sea la combinación ganadora.

Soluciones

- Podemos considerar una apuesta como un conjunto de 6 números de entre los 49 posibles, o bien como un conjunto vacío en el caso de que hayamos dejado algún bloque sin rellenar. Como un boleto contiene un máximo de 8 apuestas, la estructura de datos más adecuada será

```
TYPE
tipoConjunto= SET OF 1..49;
tipoBoleto= ARRAY [1..8] OF tipoConjunto;
```

- Se supondrá que la combinación ganadora viene almacenada en un conjunto del mismo tipo que los bloques de los boletos, mientras que el número complementario, que tiene un tratamiento aparte, será una variable integer. Haremos uso de la `FUNCTION Cardinal` del ejercicio resuelto 9.3.

Algoritmo

```

INICIO
  PARA bloque = 1 HASTA bloque = 8
    Hallar intersección de la combinación ganadora
    con el bloque
    Llamar a la función Cardinal con dicha intersec-
    ción
  FIN_PARA
  SI aciertos >= 3
    ENTONCES
      Escribir el número de bloque y de aciertos con
      el complementario
    FIN_SI
FIN

```

Codificación en Pascal

```

PROCEDURE VerAciertos (boleto,combGanadora:tipoBoleto;complementario
                    :integer);
VAR
  bloque, aciertos:integer;
BEGIN
  FOR bloque:=1 TO 8 DO
    BEGIN
      aciertos:= Cardinal(boleto[bloque]*combGanadora);
      IF aciertos>=3
        THEN Write('**BLOQUE ',bloque,' PREMIADO CON ',aciertos,'
                   ACIERTOS ');
      IF (aciertos=5) AND (complementario IN boleto[bloque])
        THEN Writeln(' MAS EL NUMERO COMPLEMENTARIO')
        ELSE Writeln;
    END;
  END;{VerAciertos}

```

- c) Para ver qué números no han sido tachados, formaremos el conjunto unión de todas las apuestas del boleto. La diferencia con el conjunto universal nos dará el resultado pedido.

Algoritmo

```

INICIO
  PARA bloque = 1 HASTA bloque = 8
    Hallar la unión de las apuestas de cada bloque
  FIN_PARA
  PARA i = 1 HASTA i = 49
    SI el valor de i no está en la unión
      ENTONCES
        Escribir el número no tachado
      FIN_SI
  FIN_PARA
  Llamar a la función Cardinal con la unión
  SI los números tachados >= 44
    ENTONCES
      Escribir un acierto al menos asegurado
    SI_NO

```

CUESTIONES Y EJERCICIOS PROPUESTOS

```
        Escribir el acierto no está asegurado
    FIN_SI
FIN
```

Codificación en Pascal

```
PROCEDURE NoTachados (boleto: tipoBoleto);
VAR
    bloque,i: integer;
    union: tipoConjunto;

BEGIN
    union:=[];
    FOR bloque:=1 TO 8 DO
        union:=union+boleto[bloque];

        Writeln(' ***** RELACION DE NUMEROS NO TACHADOS *****');
        FOR i:=1 TO 49 DO
            IF i IN ([1..49]-union) { También serviría IF NOT (i IN union) }
                THEN Writeln(i);

        {Para tener la seguridad de acertar al menos un número sea cual
        sea la combinación ganadora, es necesario que se hayan tachado
        al menos 44 números diferentes entre todas las apuestas.}

        IF Cardinal (union)>=44
            THEN Writeln ('AL MENOS TIENE UN ACIERTO ASEGURADO')
            ELSE Writeln ('PUEDE SER QUE NO ACIERTE NINGUN NUMERO');
END;{NoTachados}
```

Del ejercicio que se acaba de resolver, se desprende que una buena elección de la estructura de datos para resolver un problema simplifica notablemente la programación del mismo.

9.7 CUESTIONES Y EJERCICIOS PROPUESTOS

9.5 Sea el siguiente fragmento de programa:

```
VAR
    c : SET of CHAR;
    i : integer;
BEGIN
    c := [];
    FOR i := Ord ('A') TO Ord ('Z') DO
        c := c + [Chr (i)];
    ...
```

Indicar:

- ¿ Qué contiene c tras la ejecución ?
- Simplificar el bucle de forma que el resultado sea el mismo.

9.6 Construir un programa usando el tipo conjunto para simular el juego del bingo. El juego del bingo tiene un bombo con 90 bolas numeradas del 1 al 90 y una serie de

cartones con tres líneas de 5 números cada una. El primer jugador que rellene el cartón gana, es decir, que las bolas que salen del bombo coinciden con los números del cartón.

9.7 Indicar las sentencias incorrectas del siguiente programa y averiguar porqué son incorrectas.

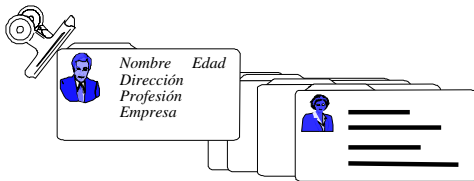
```
PROGRAM Conjuntos;
TYPE
  c1 = SET OF 'A'..'Z';
  c2 = SET OF 1..9;
VAR
  a,b : c1;
  x,y,z :c2;
BEGIN
  x := [1,3,7];
  y := [8,6];
  z := [];
  a := ['A','C'];
  b := ['A'];
  b := x+a;
  Writeln (3 IN x);
  Writeln ((x + y + y) = [1,3,7,8,6]);
  Writeln ('C' IN a);
  Writeln (((x + y) * z) = []);
  Readln;
END.
```

9.8 Diseñar un tipo abstracto de datos denominado conjunto, que sobre un array de booleanos implemente las operaciones con conjuntos. Realizar una implementación como una *unit* de Turbo Pascal, y usarla para resolver los ejercicios 9.1 y 9.2 de este capítulo. Esta implementación es útil, para cuando el lenguaje de programación utilizado no tiene conjuntos.

9.8 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Sobre la implementación del tipo abstracto de datos conjunto pueden consultarse los libros: *Estructuras de datos y algoritmos* de A.V. Aho, J.E. Hopcroft, y J.D. Ullman (Ed. Addison-Wesley, 1988), véase capítulos 4 y 5; y *Estructuras de datos, realización en Pascal* de M. Collado, R. Morales, J.J. Moreno (Ed. Díaz de Santos, 1987), véase capítulo 4.

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS



CAPITULO 10

ESTRUCTURA DE DATOS REGISTRO

CONTENIDOS

- 10.1 Concepto de registro
- 10.2 Procesamiento de registros
- 10.3 Registros jerárquicos
- 10.4 Sentencia *WITH*
- 10.5 Arrays de registros: Tablas
- 10.6 Registros variantes
- 10.7 Los registros como Tipos Abstractos de Datos
- 10.8 Representación interna de los registros
- 10.9 Extensiones del compilador Turbo Pascal
- 10.10 Registros del microprocesador. Interrupciones
- 10.11 Cuestiones y ejercicios resueltos
- 10.12 Cuestiones y ejercicios propuestos
- 10.13 Ampliaciones y notas bibliográficas

CONCEPTO DE REGISTRO

10.1 CONCEPTO DE REGISTRO

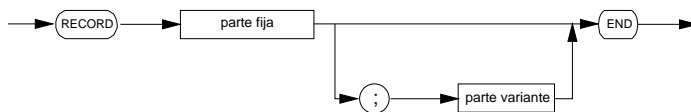
En el capítulo ocho se presentó la estructura de datos *array*, la cual se caracterizaba por tener todos sus componentes del mismo tipo. En este capítulo se estudiará una nueva estructura de datos denominada *registro* (*RECORD* en la nomenclatura anglosajona), que se puede definir como un tipo de datos estructurado con un número fijo de componentes, no necesariamente homogéneos, que son accedidos por nombre en vez de por un subíndice como ocurría en los *arrays*. A los componentes de un *registro* se les denomina *campos*.

El diagrama sintáctico del tipo *registro* es el representado en la figura 10.1. El diagrama sintáctico de *parte variante* se detalla más adelante en este capítulo, sección 10.6, *Registros variantes*. En notación *EBNF* el tipo *registro* se define así:

```
<tipo registro> ::= RECORD <listas_de_campos> END
<listas_de_campos> ::= <parte fija> | <parte fija> ; <parte variante>
| <parte variante>
<parte fija> ::= <sección de registro> { ; <sección de registro> }
<sección de registro> ::= <identificador de campo>
{ , <identificador de campo> } : <tipo> | <vacío>
```

El significado de <parte variante> se estudiará en la sección 10.6, *Registros variantes*, de este mismo capítulo.

Tipo Registro:



Parte fija:

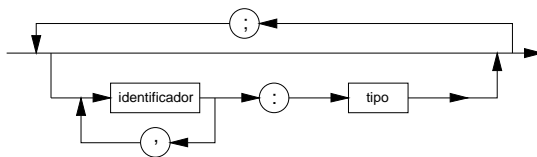


Figura 10.1 Diagrama sintáctico del tipo registro

Teniendo en cuenta el diagrama sintáctico y la gramática *EBNF*, la sintaxis general del tipo registro en Pascal es:

```
TYPE
registro = RECORD
    listaDeCampos1: tipo1;
    listaDeCampos2: tipo2;
    ...
    listaDeCamposN: tipoN;
```

ESTRUCTURA DE DATOS REGISTRO

```
                END;  
VAR  
  variable: registro;
```

Las palabras reservadas *RECORD* y *END* encierran las declaraciones de los campos. Las *listas de campos* pueden estar compuestas por un único identificador (representando un campo), o por varios identificadores de campo separados por comas, si los campos son del mismo tipo.

Ejemplo 10.1

Veamos con unos ejemplos como se declaran tipos *registro* en Pascal.

```
a)  TYPE  
      persona = RECORD  
          nombre   : PACKED ARRAY [1..25] OF char;  
          dni       : longInt; (* entero largo *)  
          sexo      : (mujer, varon) ;  
          estCivil  : (soltero, casado, otros) ;  
          sueldo    : real ;  
      END ;
```

```
b)  TYPE  
      fecha = RECORD  
          dia       : integer ;  
          mes       : integer ;  
          anio      : integer ;  
      END;
```

Este último *registro* también podía haberse declarado así:

```
TYPE  
  fecha = RECORD  
      dia, mes, anio : integer;  
  END ;
```

```
c)  TYPE  
      planeta = RECORD  
          nombre: PACKED ARRAY [1..10] OF char;  
          visible: boolean;  
          diametro, radioOrbita : real;  
      END;
```

Aquí se ha definido el tipo *planeta* que tiene una *estructura de registro*, con los *campos*: *nombre*, *visible*, *diametro* y *radioOrbita*. Puede observarse que los campos son de distintos tipos.

```
d)  TYPE  
      complejo = RECORD  
          re : real;  
          im : real;  
      END;
```


CONCEPTO DE REGISTRO

e) TYPE
 cuenta = RECORD
 numeroCliente: integer;
 nombreCliente: PACKED ARRAY [1..80] OF char;
 tipoCliente: char;
 saldo: real;
 END;

f) TYPE
 nom = PACKED ARRAY [1..80] OF char;
 socio= RECORD
 nombre : nom;
 apell1, apell2 : nom;
 sexo : (hombre, mujer)
 telefono: longInt; (* entero largo *)
 direccion: nom;
 ciudadCP : nom;
 END;

Como se puede apreciar en los ejemplos, los campos de un *registro* pueden ser de tipos simples o estructurados. Tal es el caso del campo *nombre* en los *registros* *persona*, *planeta* y *socio*, que es un *ARRAY empaquetado de caracteres* en los tres casos.

Gráficamente, un registro se puede representar de muy diversas maneras. La figura 10.2 simboliza la estructura definida en el ejemplo a). Emplea un tipo de representación bastante frecuente, en la cual los identificadores de los campos se sitúan en rectángulos adosados de tamaño aproximadamente proporcional al espacio que necesitan en memoria.

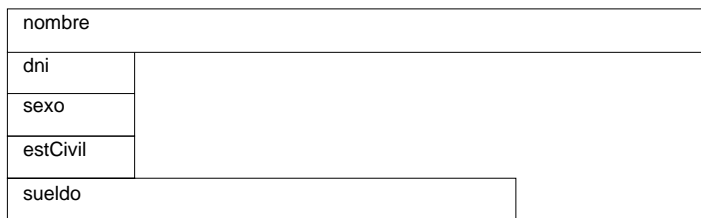


Figura 10.2 Representación de un registro

Ejemplo 10.2

También podemos declarar variables *registro* directamente con una sentencia *VAR*, si bien esto no es una buena práctica de programación.

```
VAR  
  r : RECORD  
    nombreTela : PACKED ARRAY [1..25] OF char ;  
    color      : (blanco, rojo, verde, azul) ;  
    precio, existencias : integer;  
  END ;
```

ESTRUCTURA DE DATOS REGISTRO

```
cliente : RECORD
        numeroCliente : integer;
        tipoCliente : char;
        saldo : real;
END;
```

Estas declaraciones equivalen a las siguientes:

```
TYPE
    banco = RECORD
        numeroCliente : integer;
        tipoCliente : char;
        saldo : real;
    END;
    tela = RECORD
        nombreTela : PACKED ARRAY [1..25] OF char ;
        color : (blanco, rojo, verde, azul) ;
        precio, existencias : integer;
    END ;
VAR
    cliente: banco;
    r : tela;
```

Los *registros* pueden ser empaquetados, al igual que los *arrays*, sin más que escribir *PACKED RECORD* en vez de *RECORD* al declarar el tipo *registro*. Ello implica un compromiso entre velocidad de proceso y ocupación de espacio, según se explicó en el capítulo 8.

Finalmente cabe señalar que el ámbito de los identificadores de campo se reduce al *registro* particular en el que están definidos; lo cual es equivalente a decir que campos de diferentes *registros* pueden tener el mismo nombre.

10.2 PROCESAMIENTO DE REGISTROS

Hasta ahora hemos visto el concepto de *registro* y la forma de definirlo en Pascal; pero ¿Cómo se usa dentro de un programa esta estructura de datos?

La operación más sencilla que podemos realizar con *registros* consiste en una **asignación** de un *registro* a otro del mismo tipo. Con esto realizamos una copia completa de toda la estructura, con todos sus campos.

Ejemplo 10.3

```
TYPE
    cliente = RECORD
        nombre : PACKED ARRAY [1..25] OF char;
        tfno : longInt;
        saldo : real;
    END ;
VAR
    a, b : cliente ;
```

REGISTROS JERARQUICOS

```
BEGIN
...
  a := b ; (* copia el registro b en el a *)
...
END.
```

Tras esta asignación, cada campo del *registro* *a* toma el valor del campo correspondiente del registro *b*. Se supone que se ha dado valor a los campos del *registro* *b* con anterioridad a la sentencia de asignación.

El procesamiento de los campos individuales de un *registro* es mucho más habitual que el de *registros* completos. La expresión usada para acceder a un campo de un *registro* se denomina **selector de campo**, y se construye poniendo el nombre de la variable *registro* en cuestión seguida de un punto '.' y a continuación el identificador de campo deseado.

Ejemplo 10.4

Con las declaraciones del ejemplo 10.3, podemos construir los selectores de campo:

<code>a.nombre</code>	Selecciona el campo <code>nombre</code> del <i>registro</i> <i>a</i> . Representa un <i>array empaquetado</i> de 25 caracteres.
<code>a.tfno</code>	Selecciona el campo <code>tfno</code> . Representa un entero largo (nótese que los números de teléfono se salen del rango del tipo <i>integer</i> , por eso se recurre al tipo <i>longInt</i>).
<code>a.saldo</code>	Selecciona el campo <code>saldo</code> , de tipo <i>real</i> .

Así pues, los elementos individuales (*campos*) de un *registro* pueden utilizarse de la misma manera que las variables ordinarias de su mismo tipo. La diferencia estriba en la forma de referenciarlas en el programa, según acabamos de ver.

Para referirnos a la primera letra del campo `nombre` del *registro* *a*, deberemos poner

```
a.nombre[1]
```

lo cual representa una variable de tipo *char*, (el primer elemento del *array* `a.nombre`).

10.3 REGISTROS JERARQUICOS

Los campos de un *registro* pueden ser a su vez un *registro* de otro tipo. Los *registros* así formados se denominan **registros jerárquicos**.

Ejemplo 10.5

Un ejemplo de *registro jerárquico* puede ser el siguiente:

```
TYPE
  cadena = PACKED ARRAY [1..10] OF char ;
```

ESTRUCTURA DE DATOS REGISTRO

```
fecha = RECORD
    dia : 1..31 ;
    mes : 1..12 ;
    anio : integer;
END;
persona= RECORD
    nombre : cadena;
    fechaNaci : fecha ;
    telefono : longInt; (* entero largo *)
END;
VAR
    alumno : persona ;
```

donde la variable `alumno` de tipo `persona` es un *registro jerárquico*. Suponiendo que ya se le han asignado los valores adecuados a sus diferentes campos, para escribir su contenido por pantalla pondríamos:

```
Writeln ('Nombre del alumno: ', alumno.nombre);
Writeln ('Fecha de nacimiento');
Writeln ('Día : ', alumno.fechaNaci.dia );
Writeln ('Mes : ', alumno.fechaNaci.mes );
Writeln ('Año : ', alumno.fechaNaci.anio);
Writeln ('Teléfono : ', alumno.telefono );
```

Para asignarle el valor 5 al día de nacimiento lo haremos mediante la sentencia:

```
alumno.fechaNaci.dia := 5 ;
```

10.4 SENTENCIA WITH

Cuando se tiene que acceder repetidamente a campos de un *registro* en un segmento de programa, los selectores de campo pueden resultar incómodos y reducir la legibilidad del programa. En estos casos, la sentencia **WITH** permite omitir el nombre de la variable *registro* a la hora de seleccionar cada uno de sus campos.

El diagrama sintáctico de la sentencia **WITH** es:

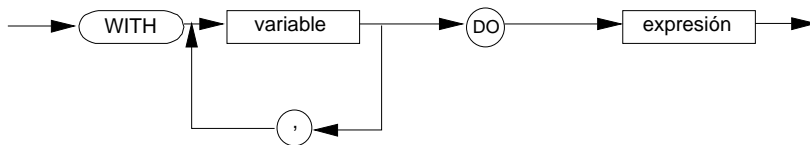


Figura 10.3 Diagrama sintáctico de WITH

Y en notación *EBNF*:

```
<sentencia with> ::= WITH <lista de variables registro> DO
                    <sentencia>
<lista de variables registro> ::= <variable registro> { , <variable registro> }
```

SENTENCIA WITH

De ambos se deduce la forma general de la sentencia *WITH* en lenguaje Pascal:

```
WITH variable_registro DO
    sentencia;
```

la *sentencia* puede ser (y en general así es) una sentencia compuesta.

Dentro del ámbito de la sentencia *WITH*, cualquier referencia a un campo de la *variable_registro* especificada, debe incluir el nombre del campo únicamente.

Ejemplo 10.6

Con la siguiente declaración:

```
TYPE
    fecha = RECORD
        dia   : 1..31 ;
        mes   : 1..12 ;
        anio  : integer;
    END;
VAR
    cumpleAños: fecha ;
```

una manera de actualizar la variable *cumpleAños* sería:

```
cumpleAños.dia := 8 ;
cumpleAños.mes := 11 ;
cumpleAños.anio := 1988 ;
```

sin embargo resulta más fácil y legible escribir:

```
WITH cumpleAños DO
BEGIN
    dia := 8 ;
    mes := 11 ;
    anio := 1988 ;
END; (* de WITH *)
```

Según se deduce del diagrama sintáctico y de la notación *EBNF*, La sentencia *WITH* puede ser *multiregistro*, es decir, incluir varios nombres de *registros* diferentes separados por ',' (coma).

La sintaxis general es:

```
WITH reg1, reg2, ... , regN DO
    sentencia;
```

lo cual es equivalente a :

```
WITH reg1 DO
WITH reg2 DO
.
.
WITH regN DO
    sentencia;
```

En este caso, si el nombre de un campo es común a varios *registros* especificados, su ámbito se reduce al *registro* más interno dentro de la sentencia *WITH* .

Ejemplo 10.7

Esta sentencia es muy útil para procesar *registros jerárquicos*. Así, para asignarle valores a la variable `alumno`, declarada en el ejemplo 10.5, pondríamos:

```
WITH alumno, fechaNaci DO
BEGIN
  nombre   := 'EUSTAQUIO ' ;
  dia      :=    5 ;
  mes      :=   12 ;
  anio     :=  1948 ;
  telefono := 253828 ;
END ;
```

10.5 ARRAYS DE REGISTROS: TABLAS

Una estructura de datos que en muchas aplicaciones puede ser útil es la *tabla*. Por ejemplo, con una *tabla* se puede representar una lista de una clase compuesta por 40 alumnos, en la que figura el nombre completo y la calificación final obtenida por cada alumno en la asignatura de Metodología de la Programación.

Nombre	Nota
José Ramón Alba Noriega	7.5
Enrique Barrio Aparicio	8.3
Juan López Pérez	9.5
...	...

Figura 10.4 Ejemplo de utilización de una *tabla*

Una de las formas de representar esta estructura de datos en Pascal es mediante un *array de registros*, como el usado en el ejemplo 10.8. El inconveniente de los arrays de registros es que consumen mucha memoria RAM, ésto se debe a que los *arrays* y los registros son estructuras de datos estáticas, debiendo de definir su tamaño en tiempo de compilación. En los capítulos siguientes se combinarán los registros con otras estructuras de datos supliendo los inconvenientes mencionados anteriormente.

Ejemplo 10.8

Para construir en Pascal una estructura de datos de tipo registro, que almacene los datos de la tabla de la figura 10.4, se pueden hacer las siguientes declaraciones:

```
CONST
  n = 40 ; (* número máximo de alumnos por clase *)
TYPE
  alumno = RECORD
    nombre      : PACKED ARRAY [1..50] OF char;
    notaFinal   : real;
  END;
  tabla = ARRAY [1..n] OF alumno;
VAR
  lista : tabla;
  i     : integer;
```

La variable `lista` es un *array de registros*, o *tabla*. Cada uno de sus componentes es un *registro*. Por ejemplo, la variable `lista[1]` es un *RECORD* que representa al primer alumno de la lista.

Para acceder a la nota final del alumno que ocupa la posición número 25 en la tabla utilizaremos el selector de campo:

```
lista[25].notaFinal
```

Ejemplo 10.9

Sobre el escenario del ejemplo 10.8, construyamos un segmento de programa que nos permita escribir por *output* la relación de alumnos aprobados junto con la nota obtenida por cada uno de ellos. Se supone que la tabla denominada `lista`, ya ha sido rellenada con anterioridad por el programa.

```
BEGIN (* programa *)
  ...
  Writeln(' RELACION DE ALUMNOS APROBADOS ');
  FOR i := 1 TO n DO
    WITH lista[i] DO
      IF notaFinal >= 5.0
        THEN Writeln ( nombre, notaFinal :4:1);

  Writeln;
  ...
  ...
END.
```

Observe, sin embargo, que *no está permitido* poner el bucle dentro de la sentencia *WITH*, es decir:

```
WITH lista[i] DO
  FOR i := 1 TO n DO
    ...
```

sería incorrecto. La causa es que *el selector de campo no puede variar dentro del ámbito de la sentencia WITH*.

10.6 REGISTROS VARIANTES

Los *registros* vistos hasta ahora mantienen invariante su composición a lo largo del programa. Sin embargo es posible definir *registros* cuya composición, o una parte de ella, varíe dependiendo del valor que tome un campo especial denominado *campo selector* o *tag-field*. Se llama *tipo selector* al tipo del *campo selector*.

Una situación donde convendría usar un *registro variante* sería por ejemplo en un *registro* de personal en el cual, si el individuo es casado habrá que considerar dos nuevos campos: nombre del cónyuge y número de hijos; mientras que si se trata de una persona soltera esos dos campos no deben existir. En el caso descrito el *campo selector* o *tag-field* es el estado civil, pues en función de éste, el *registro* contendrá esos campos o no.

El diagrama sintáctico de la parte variante de un *registro* es el representado en la figura 10.5.

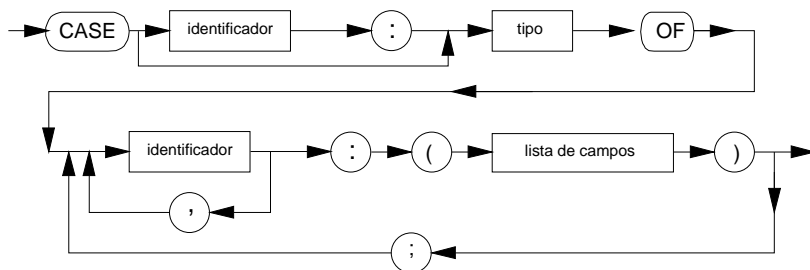


Figura 10.5 Diagrama sintáctico de la parte variante de un registro

La notación *EBNF* de un registro con parte variante es:

```

<parte variante> ::= CASE <campo selector>
                  <identificador de tipo> OF <variante>
                  { ; <variante> }
<campo selector> ::= <identificador de campo> : | <vacío>
<variante>       ::= <lista de etiquetas de case>
                  : ( <listas_de_campos> )
<lista de etiquetas de case> ::= { <etiqueta de case> { , <etiqueta de case> }
<etiqueta de case> ::= <constante>

```

Del diagrama sintáctico y la notación *EBNF* se deduce que la forma general de definición de un *registro variante* es:

```

TYPE
registroVariante = RECORD
    campoFijo1 : tipo1;
    campoFijo2 : tipo2 ;
    ...
    campoFijoN : tipoN ;
CASE campoSelector: tipoSelector OF
    valor1 : listaDeCampos1;
    valor2 : listaDeCampos2;

```


REGISTROS VARIANTES

```
        ...
        valorM : listaDeCamposM;
    END;
```

Observe que el *END* es único y no es necesario incluir otro para la cláusula *CASE*. Como la parte variante, si existe, es única y se sitúa al final de las *listas de campos*, el mismo *END* nos sirve para indicar el final de la parte variante y el final del registro. Las *listasDeCampos* se expresan así:

```
(campo1: tipo1; campo2: tipo2; ... campoN: tipoN )
```

si varios campos son del mismo tipo se pueden separar por comas, igual que ocurre con los campos de la parte fija.

Se puede observar en el diagrama sintáctico y en la notación *EBNF*, que *puede omitirse el identificador del campo selector*. Tal variante de *registro* se denomina *unión libre* o *unión no discriminada*. En algunos compiladores no se permiten las uniones libres. Por ejemplo en el compilador Turbo Pascal están permitidas las uniones libres. En programación estructurada se desaconseja la utilización de *uniones libres*, programando de forma que en ningún caso se pueda omitir el identificador del *tipo selector*.

Si existe, la parte variante es única, pero dentro de la lista de campos de la parte variante puede haber otros campos variantes (siempre después de los campos fijos). Es lo que se denomina *variantes anidadas*.

Ejemplo 10.10

Veamos ahora como se representaría el *registro variante* que comentábamos al principio:

```
TYPE
    linea = PACKED ARRAY [1..25] OF char;
    estados = (soltero, casado);
    persona = RECORD
        nombre : linea;
        dni    : integer;
        CASE estadoCivil: estados OF
            soltero: ( ) ;
            casado  : (conyuge: linea;
                    numHijos: integer);
        END;
```

Observe que si el valor del *tag-field* *estadoCivil*, es *soltero*, la lista de campos variantes está vacía, pues no se necesita ningún campo más. No obstante es necesario poner ambos paréntesis para indicar que la lista está vacía.

Si el campo *estadoCivil* toma el valor *casado*, aparecen dos nuevos campos cuyos nombres son *conyuge* y *numHijos*.

ESTRUCTURA DE DATOS REGISTRO

Debe tenerse presente al procesar *registros variantes* que, antes de acceder a uno de los campos de la parte variante, hay que asegurarse de que el valor del *campo selector (tag-field)* es el apropiado para que pueda existir ese campo.

Recíprocamente, al crear un *registro variante*, según el valor que asignemos al *campo selector (tag-field)* se crearán unos campos adicionales u otros, pero antes de acceder a ellos hay que asignarle el valor apropiado al *campo selector*.

Una vez asignado un valor al *campo selector*, se dice que la lista de campos correspondiente está *activa*. Si se intenta acceder a un campo que no pertenece a la lista de campos activa, se produce un *error de ejecución*. En cada instante sólo puede existir *una lista de campos activa*.

El *registro* del ejemplo 10.10 tenía una parte fija y otra variante. También puede ocurrir que un *registro* sea todo él variante; es decir, no tenga parte fija. El caso que se presenta a continuación es un buen ejemplo de ello:

Ejemplo 10.11

```
TYPE
  tipoCoord = (cartesianas, polares);
  coordenadas = RECORD
    CASE coord: tipoCoord OF
      cartesianas: (x, y: real);
      polares      : (ro, theta: real);
    END;
```

Utilización de memoria en registros variantes

Como se estudió en el capítulo 3, el *registro* es un tipo de datos *estático*. Es decir, se le reserva espacio en memoria en *tiempo de compilación*. El compilador tiene que reservar una cantidad fija de memoria, en la que quepa el registro completo. Como en *tiempo de compilación* no está definida la *lista activa* se reserva la memoria necesaria, para almacenar el registro cuando la lista activa sea la mayor declarada (mayor en ocupación de memoria).

En *tiempo de ejecución* estará siempre ocupada la memoria reservada a la parte fija, y según el valor del *campo selector*, se ocupará la parte necesaria de la memoria reservada a la parte variante. Esto equivale a superponer en el mismo área de memoria todas las posibles estructuras de registro que se pueden presentar, según se indica en la figura 10.6.

La figura 10.6 es una representación gráfica de la estructura diseñada en el ejemplo 10.10. El espacio reservado para la parte fija estará ocupado por los campos fijos, nombre, dni, estadoCivil. El espacio reservado para la parte variante es el suficiente para que quepa la mayor *lista de campos* de la parte variante. En este caso, si estadoCivil toma el valor soltero la lista de campos está *vacía*, luego la memoria reservada será la necesaria para almacenar la *lista activa* cuando estadoCivil toma el valor casado.

REGISTROS VARIANTES

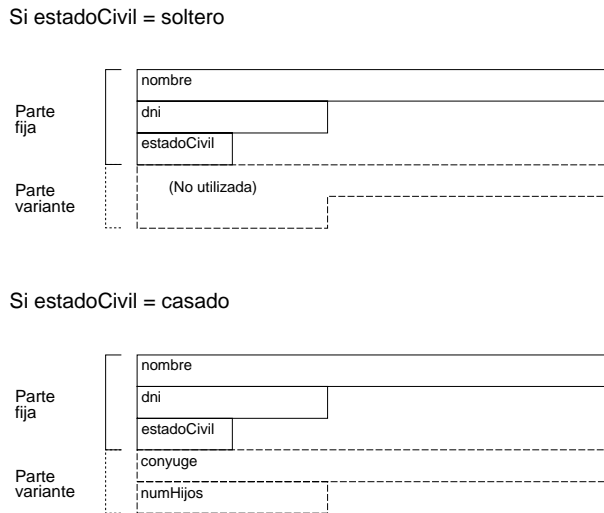


Figura 10.6 Utilización de memoria en un *registro variante*

Normas sobre la definición y uso de los registros variantes

A continuación se resumen brevemente todas las advertencias hechas sobre la definición y utilización de este tipo especial de registros.

- Una definición de *registro* puede contener sólo una *parte variante*, aunque en la lista de campos de la parte variante pueda contener otra parte variante (*variantes anidadas*). En el ejemplo 10.14 se ilustra el uso de variantes anidadas.
- Todos los identificadores de campos dentro de una definición de *registro* deben ser *únicos*.
- El *campo selector* se utiliza para indicar la *lista de campos activa* en una variable *registro*.
- El *campo selector* (o discriminador, en inglés *tag-field*) es un campo separado del registro (si está presente). Aunque se utiliza generalmente para activar una lista variante de campos, también contiene una información que puede usarse cuando convenga. En el ejemplo 10.15, se puede ver cómo se asigna valor a los campos selectores, y cómo su información se imprime en pantalla, al igual que los datos contenido en el resto de los campos.
- No confundir la cláusula *CASE* de la parte variante con una sentencia *CASE*. Ambas se parecen en que realizan una función selectora, pero son muy distintas. Las características distintivas de la cláusula *CASE* son:
 - ⌘ No se coloca un *END* para el *CASE*, se usa *END* para la definición de *registro* (consultar ejemplos 10.10 a 10.15).

ESTRUCTURA DE DATOS REGISTRO

- ✎ En realidad, el selector de *CASE* no es el *campo selector*, sino el *tipo selector*. El *campo selector* toma los valores del *tipo selector*, y para cada valor del *tipo selector*, tenemos una *lista de campos* (consultar los ejemplos que acabamos de citar). Esta es la causa de que se pueda omitir el identificador del *campo selector*, en la *unión libre* o *no discriminada*. Para ver como utilizar la *unión libre*, consultar el ejemplo 10.12.
- ✎ Una *lista de campos* puede estar *vacía*, lo cual se denota por () (consultar ejemplos 10.10, 10.12 y 10.14).
- ✎ Pueden usarse varias etiquetas para la misma *lista de campos*, como se observa en el ejemplo 10.12. A cada *lista de etiquetas de CASE* corresponde una *lista de campos*. Las etiquetas de *CASE* tienen que ser constantes del *tipo selector*.
- ✎ Las *listas de campos* van entre paréntesis. Definen los nombres y tipos de los campos.
- El *tipo selector* puede ser cualquier tipo ordinal, pero *debe utilizarse un identificador de tipo* para el mismo. No puede construirse el tipo al declarar el campo, aunque esté permitido para el resto de los campos del registro.
- Dicho tipo ordinal debe ser de cardinalidad finita, debiendo aparecer todos los valores posibles del mismo en las *listas de etiquetas* de la parte variante, aun cuando la lista de campos esté vacía para algunas de ellas. Esto elimina la utilización del tipo *integer*, y hace tediosa la utilización del tipo *char*. Generalmente el *tipo selector* suele ser un tipo *enumerado* (ejemplos 10.10 a 10.15). Se puede usar también un tipo *boolean* (ejemplo 10.14).
- El identificador del campo selector puede omitirse, si el campo selector y su tipo (*selector*) aparecen en la parte fija (o en otro tipo). No suele ser aconsejable, y algunos compiladores no lo permiten. Este tipo de variante se llama *unión libre*, y se utiliza en el ejemplo 10.12.
- La *lista de campos activa* se asigna en *tiempo de ejecución*. Puede cambiarse mediante una asignación al *campo selector*. Cuando se activa una lista de campos, se pierden los datos (si los hay) de la lista activa anterior. Ver ejemplos 10.13 y 10.15.
- El nombre del *campo selector* no aparece en los selectores de campo de los campos variantes. Es un error frecuente el incluirlo cuando se dan los primeros pasos en el mundo de la programación. En el ejemplo 10.13 se indica cómo formar selectores de campos para los campos variantes, y cuándo se pueden utilizar. También se puede consultar el ejemplo 10.15.
- Es un *error* acceder a un campo que no pertenece a la lista de campos activa.

Los ejemplos siguientes tratan de aclarar la aplicación de todas estas normas.

REGISTROS VARIANTES

Ejemplo 10.12

La siguiente declaración corresponde a una estructura de registro variante, con una lista de campos vacía para varias etiquetas (valores del *tipo selector*). Con fines didácticos, se utiliza la *unión libre*. Esta estructura es usada posteriormente en el ejercicio resuelto 10.3.

```
TYPE
  estado_civil = (s,c,v,d);
  registro = RECORD
    dni:ARRAY [1..8] OF char;
    nombre:ARRAY [1..60] OF char;
    sexo:(hombre,mujer);
    estado:estado_civil;
    CASE estado_civil OF      (* unión libre *)
      c:(Conyuge:RECORD
        dni:ARRAY [1..8] OF char;
        nombre:ARRAY [1..60] OF char
      END);
      s,v,d:();      (* lista de campos vacía *)
    END;
```

Ejemplo 10.13

Veamos cómo construir los selectores para los campos variantes, y cuándo podemos utilizarlos. Con la definición de tipo registro del ejemplo 10.10, si hacemos la siguiente declaración de variable:

```
VAR
  p: persona;
```

y suponiendo que el *campo selector*, `p.estadoCivil` tiene el valor `casado`, los selectores de los campos de la *lista activa* serían `p.conyuge` y `p.numhijos`. Si el *campo selector* vale `soltero`, la *lista de campos activa* está vacía.

Con la estructura definida en el ejemplo 10.11, si hacemos la declaración:

```
VAR
  punto: coordenadas;
```

cuando el *campo selector*, `punto.coord`, toma el valor *cartesianas*, los *selectores de campo* de la lista activa serán `punto.x` y `punto.y`. Si el *campo selector*, `punto.coord` vale *polares*, los selectores que podemos utilizar serán `punto.ro` y `punto.theta`.

Ejemplo 10.14

Diseñar una estructura de datos para almacenar los siguientes datos de una persona:

- Nombre completo
- DNI
- Edad
- Sexo (m o f)

ESTRUCTURA DE DATOS REGISTRO

- Cuando sexo = f:
 - ⌘ Número de hijos
- Cuando sexo = m
 - ⌘ Si ha hecho la mili (T o F)
 - § Cuando no ha hecho la mili
 - . Por qué causa (Menor de edad, objetor,...).

Solución

Se puede resolver utilizando una estructura de tipo registro con variantes anidadas. El selector de la primera variante será el campo sexo. Para sexo femenino habrá un campo con el número de hijos, y para sexo masculino un campo de tipo lógico indicando si se ha cumplido el servicio militar. Este campo será el selector de la variante anidada. Si toma el valor *true*, no hay más campos; pero si vale *false*, habrá otro campo enumerado, indicando la causa de que no se haya hecho la mili.

```
TYPE
Tsexo = (m, f);
TnoMili = (menor18, objetor, insumiso, excedente, exento);
Tpersona = RECORD
    nombre: string[50];
    dni: string[10];
    edad: byte;
    CASE sexo: Tsexo OF
        f: (numHijos: byte);
        m: (
            CASE mili: boolean OF
                true: ();
                false: (causa: TnoMili);
            ); (* fin lista de campos de m *)
    END; (* sirve para: las dos CASE y el RECORD *)
```

Ejemplo 10.15

Partiendo de la estructura diseñada en el ejemplo 10.12, escribir un procedimiento para leer por teclado los datos de un registro, y otro para escribirlos por pantalla.

Solución. Escribiremos un procedimiento para leer los datos, con un parámetro de tipo *Tpersona*, que es necesario declarar por *dirección*. Utilizaremos una sentencia *WITH*, para abreviar la escritura de los selectores de campo. Hay que asegurarse de que al utilizar un campo de una lista variante, el valor del *campo selector* es el apropiado para que dicha variante esté *activa*. El procedimiento para salida por pantalla es muy similar. En este último caso no necesitamos declarar el parámetro por dirección, ya que es parámetro de entrada. Prescindimos de elaborar un algoritmo, ya que únicamente realizaremos operaciones de lectura y escritura, con las siguientes precauciones:

- Activar la lista correspondiente antes de utilizar sus campos.

REGISTROS VARIANTES

- No se pueden leer ni escribir directamente valores de tipo enumerado.
- No se pueden leer valores de tipo lógico.
- Al escribir un dato lógico la salida es una **T** o una **F**.

Codificación en Pascal

```
PROCEDURE LeePersona(VAR p: Tpersona);
VAR
  ch, ch2, ch3: char;
BEGIN
  WITH p DO
  BEGIN
    Write('¿Nombre? ');
    Readln(nombre);
    Write('¿dni? ');
    Readln(dni);
    Write('¿edad? ');
    Readln(edad);
    REPEAT
      Write('¿sexo (m/f)? ');
      Readln(ch);
    UNTIL ch IN ['m', 'M', 'f', 'F'];
    CASE ch OF
      'f', 'F': BEGIN
        sexo := f;
        Write('¿Número de hijos?');
        Readln(numHijos);
      END;
      'm', 'M': BEGIN
        sexo := m;
        REPEAT
          Write('¿Servicio militar cumplido (s/n)? ');
          Readln(ch2);
        UNTIL ch2 IN ['s', 'S', 'n', 'N'];
        CASE ch2 OF
          's', 'S': mili := true;
          'n', 'N': BEGIN
            mili := false;
            Writeln('M- Menor de 18 años');
            Writeln('O- Objeter');
            Writeln('I- Insumiso');
            Writeln('E- Excedente');
            Writeln('X- Exento');
            REPEAT
              Write('Teclee la inicial de la causa: ');
              Readln(ch3);
            UNTIL ch3 IN ['m', 'M', 'o', 'O', 'i', 'I',
              'e', 'E', 'x', 'X'];
            CASE ch3 OF
              'm', 'M': causa := menor18;
              'o', 'O': causa := objeter;
              'i', 'I': causa := insumiso;
              'e', 'E': causa := excedente;
              'x', 'X': causa := exento;
            END; (* CASE ch3 *)
          END; (* 'n', 'N' *)
        END; (* CASE ch2 *)
      END; (* 'm', 'M' *)
    END;
  END;
END;
```

ESTRUCTURA DE DATOS REGISTRO

```
    END; (* CASE ch *)
  END; (* WITH *)
END; (* LeePersona *)

(*****

PROCEDURE EscribePersona(p: Tpersona);
VAR
  ch, ch2, ch3: char;
BEGIN
  WITH p DO
  BEGIN
    Writeln('Nombre: ', nombre);
    Writeln('DNI: ', dni);
    Writeln('De ', edad, ' años');
    CASE sexo OF
      f: BEGIN
        Writeln('Sexo femenino');
        Writeln('Con ', NumHijos, ' hijos');
      END;
      m: BEGIN
        Writeln('Sexo masculino');
        CASE mili OF
          true: Writeln('Servicio militar cumplido');
          false: BEGIN
            Write('Servicio militar no cumplido por ');
            CASE causa OF
              menor18: Writeln('Menor de 18 años');
              objetor: Writeln('Objetor');
              insumiso: Writeln('Insumiso');
              excedente: Writeln('Excedente');
              exento: Writeln('Exento');
            END; (* CASE causa *)
          END; (* false *)
        END; (* CASE mili *)
      END; (* sexo = m *)
    END; (* CASE sexo *)
    Write('Pulse <Intro> para continuar...');
    Readln;
  END; (* WITH *)
END; (* EscribePersona *)
```

Suponiendo la siguiente declaración:

```
VAR
  persona: Tpersona;
```

Podríamos utilizar los procedimientos anteriores como se indica a continuación:

```
Writeln('Ejemplo de utilización de registros con variantes anidadas');
Writeln('Introduzca sus datos personales:');
LeePersona(persona);
Writeln('Los datos introducidos son:');
EscribePersona(persona);
```

10.7 LOS REGISTROS COMO TIPOS ABSTRACTOS DE DATOS

Recordemos ahora lo estudiado en el capítulo 8, secciones 8.8, *Concepto de Tipo Abstracto de Datos (TAD)* y 8.9, *Los arrays como Tipos Abstractos de Datos*. Al igual que hicimos con la estructura de datos *array*, podemos aplicar aquí los conceptos de *encapsulación* y *ocultación de*

LOS REGISTROS COMO TIPOS ABSTRACTOS DE DATOS

información, y construimos TAD's con registros, a la medida de nuestras necesidades. Una manera de construir un TAD registro es mediante una *unit*, que englobe la declaración de la estructura y subprogramas (procedimientos y funciones) que realicen todas las operaciones necesarias para su utilización práctica.

Ejemplo 10.16

Crear mediante una unit de Turbo Pascal un Tipo Abstracto de Datos para manejar fechas.

Solución. Construiremos una *unit* denominada `TADfechas`. En la parte de *INTERFACE* declararemos la estructura de tipo registro para almacenar una fecha, y la cabecera de los subprogramas necesarios. En la parte de *IMPLEMENTACION* incluiremos el código de los subprogramas declarados anteriormente, en la parte de *INTERFACE*. Ya se realizó una *unit* similar en el ejercicio resuelto 7.5 (del capítulo 7). En dicho ejercicio resuelto se incluyen los algoritmos de los subprogramas construidos, pero no se utilizaban registros.

Codificación en Pascal

```
UNIT TADfecha;
(* Manejo de fechas *)

INTERFACE
TYPE
    Tfecha = RECORD
        d: 0..31;
        m: 0..12;
        a: 00..99;
    END;

PROCEDURE LeeFecha(VAR f: Tfecha);
PROCEDURE EscribeFecha(f: Tfecha);
FUNCTION diasDesde1960 (f: Tfecha):integer;
FUNCTION DiasEntreFechas (f1, f2: Tfecha): integer;
FUNCTION igualdad (f1, f2: Tfecha):boolean;
FUNCTION posterior (f1, f2: Tfecha):boolean;

IMPLEMENTATION
{-----}

PROCEDURE LeeFecha(VAR f: Tfecha);
BEGIN
    WITH f DO
        Readln(d, m, a);
END;

{-----}

PROCEDURE EscribeFecha(f: Tfecha);
BEGIN
    WITH f DO
        Write(d, '-', m, '-', a);
END;

{-----}

FUNCTION diasDesde1960 (f: Tfecha): integer;
(* Esta función calcula el número de días transcurridos desde
```

ESTRUCTURA DE DATOS REGISTRO

```

    el 1 de enero de 1960 hasta una fecha dada *)
VAR
    nd:integer;
BEGIN
    WITH f DO
    BEGIN
        nd:= Trunc (30.42 * (m - 1)) + d;
        IF m = 2
            THEN nd:= nd + 1;
        IF (m > 2) AND (m < 8)
            THEN nd:= nd + 1;
        IF (a MOD 4 = 0) AND (m > 2)
            THEN nd:= nd + 1;
        IF (a - 60) DIV 4 > 0
            THEN nd:= nd + 1461 * ((a - 60) DIV 4);
        IF (a - 60) MOD 4 > 0
            THEN nd:= nd + 365 * ((a - 60) MOD 4) + 1;
    END;
    diasDesdel960:= nd
END;

{-----}

FUNCTION DiasEntreFechas (f1, f2: Tfecha): integer;
BEGIN
    IF posterior(f1,f2)
        THEN
            DiasEntreFechas:= diasDesdel960 (f1) - diasDesdel960 (f2)
        ELSE
            DiasEntreFechas:=-1; (* Error *)
    END;

{-----}

FUNCTION igualdad (f1, f2: Tfecha):boolean;
BEGIN
    WITH f1 DO
        IF (d = f2.d) AND (m = f2.m) AND (a = f2.a)
            THEN igualdad:= TRUE
            ELSE igualdad:= FALSE;
    END;

{-----}

FUNCTION posterior (f1, f2: Tfecha):boolean;
(* Verifica que f1 es posterior a f2 *)
BEGIN
    posterior:= FALSE;
    IF f1.a > f2.a
        THEN
            posterior:= TRUE
        ELSE
            IF (f1.a = f2.a)
                THEN
                    IF (f1.m > f2.m) OR ((f1.m=f2.m) AND (f1.d>f2.d))
                        THEN posterior:= TRUE;
    END;
END.

```

Veamos un ejemplo de utilización de la *unit* TADfecha dentro de un programa cliente. Obsérvese que el programa no necesita ni siquiera definir la estructura registro Tfecha, porque ya

REPRESENTACION INTERNA DE LOS REGISTROS

está definida en la *unit*, y se puede utilizar, al igual que las funciones *DiasDesde1960*, *diasEntreFechas*, *Igualdad* y *Posterior*. Para utilizarlas, hay que incluir esta *unit* en la cláusula *uses*. Si el programa no utiliza otras *units*, basta incluir la sentencia `Uses TADFecha` justo a continuación de la cabecera del programa.

```
PROGRAM Fechas(input, output);
Uses TADFecha;
VAR
    ...
    fecha1, fecha2: Tfecha;
BEGIN
    ...
    Writeln('Ejemplo de utilización de un TAD para manejo de fechas');
    Write('Introduzca una fecha (dd, mm, aa) :');
    LeeFecha(fecha1);

    Write('Días transcurridos desde el 1-1-1960 hasta ');
    EscribeFecha(fecha1);
    dias := DiasDesde1960(fecha1);
    Writeln(' : ', dias );

    Write('Introduzca otra fecha (dd, mm, aa) :');
    LeeFecha(fecha2);

    IF Igualdad(fecha1, fecha2)
    THEN Writeln('Las dos son la misma fecha ')
    ELSE
        IF posterior(fecha1, fecha2)
        THEN
            BEGIN
                Writeln('La primera fecha es posterior a la segunda. ');
                Write('Entre ambas fechas han transcurrido ');
                Writeln(diasEntreFechas(fecha1, fecha2), ' días');
            END
        ELSE
            BEGIN
                Writeln('La segunda fecha es posterior a la primera. ');
                Write('Entre ambas fechas han transcurrido ');
                Writeln(diasEntreFechas(fecha2, fecha1), ' días');
            END;
    ...
END.
```

10.8 REPRESENTACION INTERNA DE LOS REGISTROS

Las variables de tipo registro se representan en la memoria del ordenador como una secuencia de los campos que las componen. Así el primer campo declarado en el registro se almacena en la posición de memoria más baja, a continuación el siguiente campo, y así sucesivamente con todos los campos de la parte fija. Si el registro tiene parte variante, entonces cada parte variante comienza en la misma dirección de memoria.

Si los registros son jerárquicos, es decir si uno de sus campos es a su vez otro registro, se mantiene la representación anterior, pero en vez de un campo de tipo simple se coloca la representación interna del registro en la posición correspondiente. De la misma forma se representarían otros tipos estructurados dentro de un registro.

10.9 EXTENSIONES DEL COMPILADOR TURBO PASCAL

En Turbo Pascal no es necesario que en las etiquetas de la parte variante de un *registro* aparezcan todos los valores posibles del campo selector. Además, se puede utilizar un campo de la lista activa como parámetro por dirección, como se ilustra en el ejemplo 10.17.

Ejemplo 10.17

Supongamos el siguiente procedimiento, cuyos parámetros son declarados por dirección:

```
PROCEDURE Intercambia(VAR a, b: real);
VAR
  aux: real;
BEGIN
  aux := a;
  a := b;
  b := aux;
END;
```

Dadas las siguientes declaraciones:

```
TYPE
  tipoCoord = (cartesianas, polares);
  coordenadas = RECORD
    CASE coord: tipoCoord OF
      cartesianas: (x, y: real);
      polares     : (ro, theta: real);
    END;
VAR
  c: coordenadas;
```

y suponiendo que `c.coord` tiene el valor `cartesianas`, podemos hacer la siguiente llamada al procedimiento anterior:

```
Intercambia(c.x, c.y);
```

Análogamente, si `c.coord` vale `polares`, sería válida la llamada:

```
Intercambia(c.ro, c.theta);
```

10.10 REGISTROS DEL MICROPROCESADOR. INTERRUPCIONES

En este capítulo se ha estudiado la estructura de datos *registro*. Pero en Informática esta palabra no siempre se refiere a la estructura de datos incorporada por casi todos los lenguajes de alto nivel, cuyos campos o elementos (que pueden ser de distintos tipos), se referencian por su nombre. Según el contexto, puede tener otro significado completamente distinto. Por ejemplo, en ciertas ocasiones se accede desde un programa (sobre todo en lenguaje ensamblador) a los *registros del microprocesador*. A pesar de su nombre, estos registros no son estructuras de datos como las que acabamos de estudiar.

REGISTROS DEL MICROPROCESADOR. INTERRUPCIONES

Cuando la información viaja entre los componentes del ordenador (CPU, memoria, periféricos) se suele almacenar en distintos tipos de memorias temporales, ubicadas en distintas zonas. Dentro del microprocesador, estas posiciones se llaman *registros*. La familia de los microprocesadores 8086, 8088 y 80286 tienen registros de 16 bits. Los microprocesadores 80386 y 80486 tienen registros de 32 bits, pero los primeros 16 bits se mantienen iguales por compatibilidad. Existen varios tipos de registros, todos ellos representados en la figura 10.7. Su funcionamiento y características se resumen en los epígrafes siguientes.

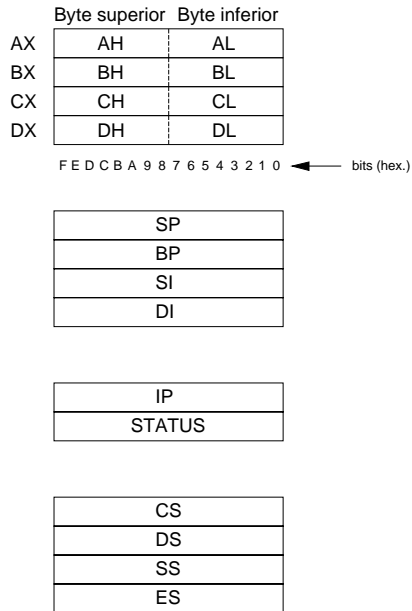


Figura 10.7 Registros del microprocesador

Registros generales

Son cuatro registros internos del microprocesador, identificados por *AX*, *BX*, *CX* y *DX*. Contienen posiciones de memoria de acceso muy rápido dentro de la CPU. Se llaman *registros de uso general* porque pueden almacenar cualquier tipo de información. Cada uno tiene 16 bits (2 bytes, una palabra), pero puede utilizarse también como si fuese 2 registros separados de 8 bits. Por ejemplo, el AX puede descomponerse en AH y AL; AH es el byte más significativo o de mayor orden (*High*), y AL el de orden bajo (*Low*). Estos cuatro registros son:

- **AX:** *Acumulador*. Se usa en operaciones aritméticas y de entrada/salida de datos.
- **BX:** *Base*. Contiene direcciones de memoria del *segmento de pila*, descrito en el siguiente grupo de registros.

ESTRUCTURA DE DATOS REGISTRO

- **CX:** *Contador*. Se usa como contador en bucles.
- **DX:** *Registro de datos*. Se usa en operaciones aritméticas y como extensión de AX cuando hay que almacenar datos de 32 bits.

Los microprocesadores 80386, 80486 y *Pentium* tienen registros generales de 32 bits denominados EAX, EBX, ECX y EDX. La E significa *extended*. Así en EAX sus dos bytes más bajos son AX, que a su vez se descompone en AL y AH. De igual forma se puede decir para el resto de los registros generales.

Los coprocesadores matemáticos de la familia 80x87 también tienen 8 registros adicionales de reales (80 bits), que constituyen la denominada pila de punto flotante. Los 8 elementos individuales de la pila de punto flotante se denominan: ST(0), ST(1), ST(2), ..., ST(7).

Ejemplo 10.18

Cuando se ejecuta un programa compilado con Turbo Pascal, los resultados de función de tipo *ordinal* se devuelven en estos registros de la CPU. Los bytes se devuelven en AL, las palabras en AX, y las palabras dobles se devuelven en DX:AX (la palabra de orden alto en DX, la de orden bajo en AX).

Los resultados de funciones de tipo *real* (tipo *real* de Turbo Pascal) se devuelven en DX:BX:AX. Los resultados de funciones de tipo *puntero* (el tipo puntero se estudiará en el capítulo 12) se devuelven en DX:AX.

Punteros e Índices

Se utilizan fundamentalmente en el direccionamiento de posiciones de memoria. Son todos de 16 bits y se describen a continuación:

- **SP:** *Puntero de pila (Stack Pointer)*. Contiene la dirección de la cabeza de la estructura *pila*. No puede usarse como registro de uso general. En el capítulo doce, sección 12.9, *Gestión de memoria dinámica en Turbo Pascal*, se explica en qué consiste la estructura *pila*.
- **BP:** *Puntero base (Base Pointer)*. Almacena desplazamientos dentro de la *pila*. También puede funcionar como *registro de uso general*.
- **SI:** *Indice Fuente (Source Index)*. Se usa en operaciones con cadenas de caracteres. También puede actuar como *registro de uso general*.
- **DI:** *Indice Destino (Destination Index)*. Funciona de manera similar a SI. En las operaciones de tratamiento de cadenas de caracteres SI contiene la dirección origen en transferencias, y DI la dirección de destino.

Registros de segmento

El espacio que ocupa un programa cargado en memoria puede descomponerse en tres partes: el código, los datos y la pila (*stack*). Además, el programa puede acceder a una cuarta zona *extra*, que puede utilizarse como zona secundaria de datos de uso general.

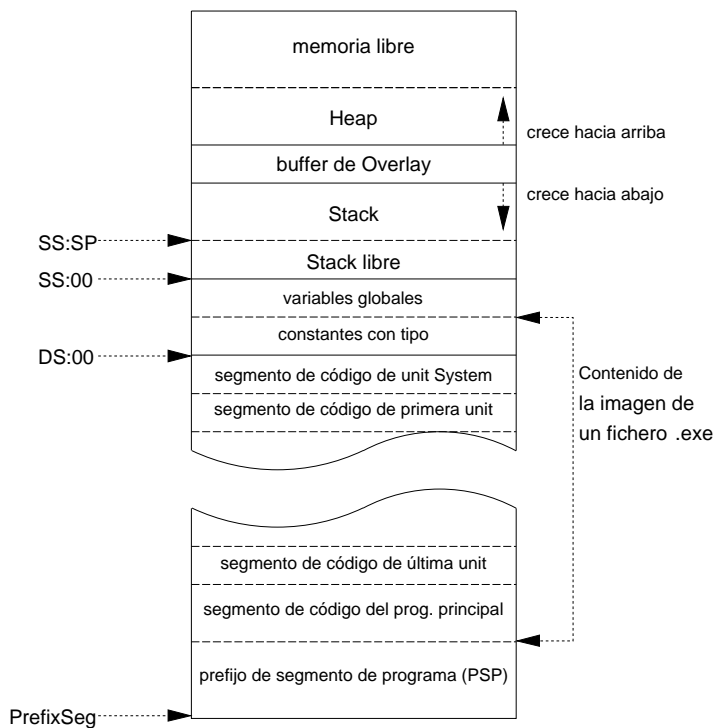


Figura 10.8 Mapa de memoria de un programa en Turbo Pascal

Estas partes de la memoria se llaman *segmentos*, y la dirección del principio de cada segmento se guarda en los *registros de segmento* de la CPU. Son los siguientes:

- CS: Contiene la dirección del *segmento de código (Code Segment)*, donde se almacena el código de los programas cuando se cargan en memoria para ser ejecutados.
- DS: Contiene la dirección del *segmento de datos (Data Segment)*, donde se almacenan los datos utilizados por un programa. Por ejemplo, en el caso de un programa compilado con Turbo Pascal, este segmento contiene todas las constantes con tipo y las variables globales.

ESTRUCTURA DE DATOS REGISTRO

- **SS:** Contiene la dirección del *segmento de pila (Stack Segment)*. Funciona junto con el *puntero de pila, SP*. Al entrar en un programa se carga este registro y el *puntero de pila*, de tal manera que SS:SP apunta al tope de la *pila (Stack)*. Durante la ejecución del programa SS no cambia, pero SP se puede desplazar hacia abajo hasta alcanzar la parte inferior del segmento. Los parámetros se transfieren a los procedimientos y funciones a través del segmento de pila (*stack*). Antes de llamar a un subprograma los parámetros se apilan en el *stack* por su orden de declaración, y antes de retornar el subprograma elimina todos los parámetros. Si el subprograma es recursivo, se guarda la copia de cada llamada en el *stack*.
- **ES:** Contiene la dirección del *segmento extra (Extra Segment)*, usado en operaciones de copia o transferencia de información.

En la figura 10.8 se observa la disposición de estos segmentos en la memoria RAM, suponiendo que está cargado en memoria un programa compilado con Turbo Pascal.

El *prefijo de segmento de programa (PSP)* es una zona de memoria de 256 bytes, construida por el DOS cuando se carga un fichero ejecutable (.exe). Cada módulo compilado (programa o unit) tiene su propio *segmento de código*. El programa principal ocupa el primer segmento de código; los siguientes van siendo ocupados por las *units*, en orden inverso al de su aparición en la cláusula *uses*; el último lo ocupa la *unit System* (que no necesita aparecer en la cláusula *uses*, está disponible siempre). El tamaño máximo de un segmento de código es de 64 Kb, pero el número de módulos sólo está limitado por la memoria disponible.

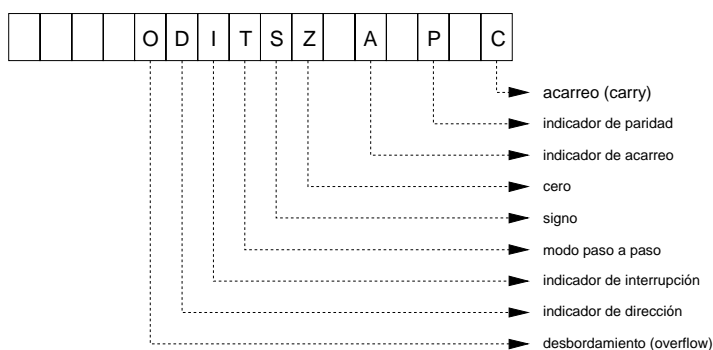


Figura 10.9 Registro de banderas (*flags*)

Otros registros

- **IP:** *Puntero de instrucciones (Instruction Pointer)*. Contiene la dirección de la siguiente instrucción a ejecutar.

- **STATUS:** *Registro de estado o registro de banderas (flags).* Contiene los indicadores de estado del microprocesador. Su formato se representa en la figura 10.9.

Programación en lenguaje máquina y ensamblador

El lenguaje nativo de los microprocesadores de la familia 80x86, es el lenguaje máquina de dicha familia, o su representación simbólica denominada lenguaje ensamblador. En algunas aplicaciones es necesario escribir directamente partes de su código en ensamblador o código máquina, por ejemplo para el manejo de ciertos dispositivos, o para incrementar considerablemente la velocidad de ejecución de algunas partes del programa. Sin embargo éstos casos seran excepciones a la regla general, ya que debe de evitarse el uso de fragmentos de programas en código de bajo nivel (ensamblador y máquina), al igual que debe de tratarse de usar siempre Pascal estándar, para que los programas sean fáciles de portar entre distintas máquinas y compiladores.

El compilador Turbo Pascal permite incluir fragmentos de código ensamblador o lenguaje máquina por medio de tres métodos:

- *Inline.* La sentencia *inline* permite incluir directamente instrucciones en código máquina (en notación hexadecimal) en medio del código escrito en Pascal. Véase ejemplo 10.19.
- *Asm.* La directiva *asm* permite incluir código ensamblador en medio del código escrito en Pascal. Véase ejemplo 10.20.
- *External (.OBJ).* La declaración de subprogramas *external* permite la utilización de subprogramas previamente compilados a ficheros con la extensión .OBJ, que pueden estar escritos en lenguaje máquina, ensamblador, C o C++. Véase ejemplo 7.19 (capítulo 7, *Subprogramas*).

Ejemplo 10.19

Se escribe una función y un procedimiento con sentencias *inline* para incrementar enteros. El procedimiento incrementa una variable global. La función incrementa el parámetro que recibe, devolviéndolo como resultado. Las instrucciones se escriben en código máquina directamente en base 16, como comentario es costumbre colocar su equivalente en ensamblador. Un subprograma que comienza con la directiva *inline* no tiene sentencias *BEGIN* y *END*. El compilador sustituye en la generación de código cada "llamada" a este subprograma por su código máquina correspondiente en el punto de la llamada. Por lo tanto un subprograma *inline* no tiene instrucción de retorno, y no se le debe añadir.

```
PROGRAM Ejemplo_Inline (Output);
VAR x1,x2:word;

PROCEDURE incrementaX2;
INLINE ($FE/$06/X2); (* INC x2 *)
```

ESTRUCTURA DE DATOS REGISTRO

```
FUNCTION incrementa(z:word):word;
INLINE (
    $58/          (* POP AX *)
    $40);         (* INC AX *)

BEGIN
    x1=10;
    x2=1;
    IncrementaX2;
    Writeln('X2=',x2);      (* Escribe 2 *)
    Writeln(incrementa(x1)); (* Escribe 11 *)
END.
```

Las instrucciones *inline* no deben alterar los contenidos de los registros DS, SS, SP o BP, ya que son utilizados por el resto del código generado por el compilador Turbo Pascal.

Ejemplo 10.20

El principal problema de las instrucciones *inline* es la obligatoriedad de escribirlas en base 16, lo que conlleva la consulta continua de la representación interna de cada uno de los nemotécnicos del lenguaje ensamblador. Una forma de evitarlo es el uso de la sentencia *asm*. La sintaxis de la instrucción *asm* es de la forma:

```
ASM
    Sentencia_ASM_1
    Sentencia_ASM_2
    ...
    Sentencia_ASM_n
END;
```

Se pueden poner más de una instrucción ensamblador por línea si se separan por un punto y coma (;) o por un comentario en Pascal. Las instrucciones *asm*, al igual que las *inline*, también deben preservar los registros BP, SP, SS y DS.

A continuación se muestra un ejemplo del uso de la directiva *asm* dentro de una función. Turbo Pascal reserva la variable *@Result* para almacenar el resultado de una función dentro de la parte de sentencias de la función.

```
PROGRAM Ejemplo_Asm (Output);

VAR
    x:word;

FUNCTION incrementa(z:word):word;
BEGIN
    BEGIN
        ASM
            inc z          (* incrementa el valor de z *)
            mov ax,z       (* Pone z en el registro AX *)
            mov @Result, ax (* Pone AX en @Result *)
        END;
    END;

BEGIN
    x:=333;
    x:=incrementa(x);
    Writeln('x=',x);     (* Escribe 334 *)
END.
```

Interrupciones

Una *interrupción* es una señal hecha a la CPU para que detenga temporalmente la tarea que estaba ejecutando, y atienda a una petición recibida desde el exterior. Una vez atendida la petición, la CPU continuará con la ejecución del proceso interrumpido.

Hay dos tipos de interrupciones:

- Interrupciones *físicas*, o de Hardware. Realizadas desde periféricos.
- Interrupciones *lógicas*, o de Software. Realizadas desde programas.

Con la aparición del microprocesador 8086, Intel introduce el concepto de interrupción por software. Anteriormente solo existían interrupciones por Hardware.

A cada interrupción, sea física o lógica, se le asocia un número del 0 al 255, que debe especificarse como argumento en los procedimientos que realizan las interrupciones lógicas. Las interrupciones más usadas se representan en la tabla 10.1. Por ejemplo, la interrupción 16 (10 hex.) controla los servicios de vídeo, y la interrupción 22 (16 hex.) controla el acceso al teclado.

Número de Interrupción (hexadecimal)	Periférico
10	vídeo
13	disco
14	puerta serie
16	teclado
17	impresora
8	reloj

Tabla 10.1 Interrupciones principales

Los primeros 1024 bytes de la memoria están organizados en 256 zonas de 4 bytes, y cada zona se dedica a una interrupción (la primera zona a la interrupción número 0, la segunda a la interrupción número 1, ...). Estas 256 zonas, también llamadas *vectores de interrupción*, constituyen la *tabla de interrupciones*, o *tabla de vectores de interrupción*.

Cada zona de 4 bytes se divide a su vez en otras dos de 2 bytes. Los dos primeros bytes forman el valor del puntero de instrucción (IP) para el comienzo de la rutina de servicio, y los dos segundos bytes son el valor del segmento de código en el que se encuentra dicha rutina de servicio (CS).

Como ejemplo de interrupciones físicas, pueden citarse las producidas por el circuito del reloj unas 18 veces por segundo (interrupción número 8). El programa en ejecución se interrumpe, se termina la instrucción máquina en curso pero no se empieza con la siguiente. La dirección de esta siguiente instrucción y los indicadores de estado se almacenan en la pila. La CPU provoca una bifurcación a la dirección determinada por CS:IP, cuyos valores se toman de la posición octava de la tabla de interrupciones. En esta posición de memoria (y en las siguientes) debe estar la rutina (o procedimiento) de interrupción correspondiente. Una vez finalizada, el sistema restaura automáticamente los indicadores de estado, y continúa con la ejecución del programa en el punto en que fue interrumpido.

Las interrupciones lógicas funcionan de manera similar, con la diferencia de que la llamada de atención a la CPU procede del propio programa en ejecución, no de un periférico externo. Sirven para ejecutar tareas del sistema, como por ejemplo ocultar un fichero, vaciar un buffer o memoria auxiliar, acceder directamente a dispositivos de entrada/salida o a posiciones de memoria, etc. Cuando el programador las va a utilizar no debe preocuparse por su localización en memoria, basta con conocer el número o código correspondiente e indicarlo como argumento en la llamada a la interrupción. En la fase de inicialización del sistema se establece el contenido de los vectores de interrupción (o zonas), para que a cada interrupción se le asigne la dirección de memoria donde está el código de ejecución que le corresponda.

El BIOS²⁰ de un ordenador personal es en realidad una colección de rutinas de servicio para realizar la mayor parte de las operaciones de entrada/salida, que funcionan mediante interrupciones lógicas grabadas en una ROM (Read Only Memory). De aquí viene el nombre de ROM BIOS. Por ejemplo, las interrupciones para controlar el vídeo y el teclado, citadas antes, son utilizadas por la ROM BIOS. Pero esto no quiere decir que todas las interrupciones lógicas estén reservadas para su utilización por la ROM BIOS. La mayoría de los controladores de periféricos utilizan también interrupciones lógicas.

Interrupciones lógicas en Turbo Pascal

Veamos como realizar *interrupciones* desde un programa en Turbo Pascal, y como se utilizan en este proceso los registros internos de la CPU. Como acabamos de ver, las interrupciones producen la ejecución de rutinas de servicio. Estas rutinas necesitan valores de entrada, y producen valores de salida, valores que se almacenan temporalmente en los registros del microprocesador.

En la *unit Dos* de Turbo Pascal está el procedimiento *Intr*, cuya llamada produce una interrupción lógica. En la llamada al procedimiento *Intr* hay que especificar dos argumentos, pues la cabecera de su declaración es:

```
PROCEDURE Intr(intNo: word; VAR regs: registers);
```

²⁰ BIOS (*Basic Input/Output System*) es el sistema básico de entrada/salida, que realiza gran parte del trabajo del ordenador personal.

REGISTROS DEL MICROPROCESADOR. INTERRUPCIONES

El parámetro `intNo` recibe el número de la interrupción correspondiente. El segundo parámetro, `regs` es de un tipo especial, `registers`, definido en la parte de interfaz de la *unit Dos*. Es un registro variante de unión libre, cuya estructura es:

```
Registers = RECORD
    CASE integer OF
        0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, flags: word);
        1: (AL, AH, BL, BH, CL, CH, DL, DH: byte);
    END;
```

Cada uno de los campos de la variante 0 representa a un registro del microprocesador. Cada uno de los campos de la variante 1 representa a un byte de los registros de uso general AX, BX, CX y DX.

Antes de ejecutar la interrupción especificada por `intNo`, *Intr* almacena el contenido de los campos del parámetro `regs` en los registros de la CPU (AX, BX, CX, DX, BP, SI, DI, DS, ES y STATUS). Una vez terminada la rutina de servicio, el contenido de los registros anteriores es almacenado otra vez en el parámetro `regs`.

La variante 1 se utiliza en ocasiones en las que es necesario colocar (u obtener) valores distintos en las dos mitades de un registro dado. Por ejemplo, es usada por la rutina de VIDEO de la ROM BIOS que sitúa el cursor en una posición determinada de la pantalla (interrupción 16). Antes de ejecutarse *Intr*, se almacenan en el registro DX las coordenadas (línea y columna) de la posición deseada. El número de línea se guarda en DH, y el de columna en DL.

Ejemplo 10.21

Como ejemplo de utilización del tipo *registers* y el procedimiento *Intr*, escribiremos una *unit* (TPU) que ayude a crear pantallas.

Algoritmo

```
INICIO
INTERFACE
    Ocultar_cursor
    Modo_pantalla (color o monocromo)
    Mostrar_cursor
    Dibujar_rectángulo
    Emitir_pitido

IMPLEMENTACION
    Ocultar_cursor
        Almacenar en los reg. CH y AH los valores $20 y $01
        Llamar a la interrupción número 10 (hex.)
```

ESTRUCTURA DE DATOS REGISTRO

Modo_pantalla

Almacenar en el registro AH el valor 15
Llamar a la interrupción número 10 (hex.)
Recuperar el valor del registro AL

Mostrar_cursor

Modo_valido := falso;
SEGUN Modo_pantalla HACER
 color: Almacenar en AH, CH y CL los valores 1, 6 y 7
 Modo_valido := cierto;
 monocromo: Almacenar en AH, CH, y CL los valores 1, 12 y 13
 Modo_valido := cierto;
FIN_SEGUN
SI Modo_valido
 ENTONCES
 Llamar a la interrupción número 10 (hex.)

Emitir_pitido

Emitir_sonido de 440 HZ
Esperar 500 milisegundos
Parar_sonido

Dibujar_rectángulo

Dibujar rectángulo de vértices opuestos (a,b) y (c,d)

FIN

Codificación en Pascal

```
UNIT pantalla;  
(* Ayuda a crear pantallas *)  
  
INTERFACE  
USES Crt,Dos;  
  
PROCEDURE oculta_cursor;  
FUNCTION modo_pantalla:integer;  
PROCEDURE muestra_cursor;  
PROCEDURE pita;  
PROCEDURE cuadro (a,b,c,d:integer);  
  
IMPLEMENTATION  
{-----}
```

REGISTROS DEL MICROPROCESADOR. INTERRUPCIONES

```
PROCEDURE Oculta_cursor;
VAR
    regs:registers;
BEGIN
    regs.CH:= $20;
    regs.AH:= $01;
    Intr ($10,regs);
END;
{-----}
FUNCTION Modo_pantalla:integer;
VAR
    regs:registers;
BEGIN
    regs.AH:= 15;
    Intr($10,regs);
    Modo_pantalla:= regs.AL;
END;
{-----}
PROCEDURE Muestra_cursor;
VAR
    regs:registers;
    modo_valido:boolean;
BEGIN
    modo_valido:= FALSE;
    CASE Modo_pantalla OF
    3: BEGIN
        Modo_valido:= TRUE; (* Modo color *)
        regs.AH:= 1;
        regs.CH:= 6;
        regs.CL:= 7;
        END;
    7: BEGIN
        Modo_valido:= TRUE; (* Modo monocromo *)
        regs.AH:= 1;
        regs.CH:= 12;
        regs.CL:= 13;
        END;
    END;
    IF Modo_valido
    THEN Intr($10,regs);
    END;
{-----}
PROCEDURE pita;
BEGIN
    Sound (440);
    Delay (500);
    NoSound
END;
```

ESTRUCTURA DE DATOS REGISTRO

```
{-----}
PROCEDURE Cuadro (a,b,c,d:integer);
(* Dibuja un cuadro de vértices opuestos (a,b) y (c,d) *)
VAR
  k:integer;
BEGIN
  GotoXY (a,b);
  Write (Chr(218));
  FOR k:= a + 1 TO (c - 1) DO
    Write (Chr(196));
  Writeln (Chr(191));
  FOR k:= b + 1 TO (d - 1) DO
    BEGIN
      GotoXY (a,k);
      Writeln (Chr(179));
      GotoXY (c,k);
      Writeln (Chr(179))
    END;
  GotoXY (a,d);
  Write (Chr(192));
  FOR k:= a + 1 TO (c - 1) DO
    Write (Chr(196));
  Writeln (Chr(217));
END;
END.
```

Llamadas al DOS

Las llamadas al sistema operativo DOS se realizan mediante interrupciones lógicas. Esto quiere decir que se pueden hacer mediante el procedimiento *Intr*. Pero Turbo Pascal proporciona otro método de llamada al DOS, más sencillo de utilizar, que es el procedimiento *MsDos*, de la *unit Dos*. Se utiliza así:

```
MsDos(regs);
```

donde *regs* es una variable del tipo *registers* que acabamos de ver en la sección anterior. Cada una de las funciones del sistema operativo tiene un número diferente, que se almacena en el registro AH. La mayoría de estas funciones necesita otros parámetros adicionales, que se guardan en otros registros, antes de hacer la llamada al DOS.

El efecto de una llamada al procedimiento *MsDos*, es el mismo que el de una llamada a *Intr* con un valor de \$21 para el parámetro *intNo*.

Para hacer una llamada al DOS usando *MsDos*, hay que definir una variable de tipo *registers*, cargar el número de servicio en AH, cargar los parámetros necesarios en los registros adecuados, y utilizar esta variable como argumento de *Intr*.

CUESTIONES Y EJERCICIOS RESUELTOS

Ejemplo 10.22

Para obtener el espacio libre en el disco de la unidad A:, se necesitan las sentencias:

```
regs.AH := $36;      {servicio "obtener espacio libre del disco"}
regs.DL := $01;      {petición sobre A: (unidad 1)}

MsDos(regs);        {hace la llamada al DOS}
```

Si se produce algún error, el código de error correspondiente se devuelve en AX. La variable *DosError*, de la *unit Dos*, almacena dicho código de error.

Escritura de procedimientos de interrupción

Se pueden escribir rutinas de servicio de interrupciones en Turbo Pascal, especificando la directiva de compilación *interrupt*. Los llamaremos *procedimientos de interrupción*. La cabecera de un procedimiento de interrupción tiene que tener la siguiente estructura:

```
PROCEDURE Interrupcion(flags, CS, IP, AX, BX, CX, DX,
                       SI, DI, DS, ES, BP: word); interrupt;
```

Puede observarse que los registros se pasan como parámetros, lo que permite usarlos y modificarlos en el código. Se pueden omitir uno o todos los parámetros, comenzando por *flags*, pero no se puede omitir un parámetro específico sin omitir también los precedentes. Es un error declarar más parámetros que los especificados.

Antes de ejecutarse el código, un procedimiento de interrupción salva automáticamente todos los registros (aunque no estén declarados los parámetros correspondientes) e inicializa el registro DS. Al finalizar se restauran los valores de los registros, y se ejecuta una instrucción de retorno de interrupción.

Un procedimiento de interrupción puede modificar sus parámetros, produciendo la modificación del registro correspondiente antes del retorno de interrupción.

10.11 CUESTIONES Y EJERCICIOS RESUELTOS

10.1 Dadas las declaraciones

```
TYPE
  fecha    = RECORD
    dia, mes: integer;
  END;
  registro = RECORD
    nombre: string[20];
    fechaNac: fecha;
    titulo: ARRAY[1..5,1..5] OF integer;
  END;
  matriz   = ARRAY[1..10] OF registro;

VAR
  a: matriz;
```

ESTRUCTURA DE DATOS REGISTRO

Indicar el tipo de las siguientes variables:

- 1) `a[5]`
- 2) `a[8].nombre`
- 3) `a[8].fechaNac.mes`
- 4) `a[5].titulo[1,2]`

Solución

- 1) tipo `registro`, definido en el enunciado
- 2) `string[20]`
- 3) `integer`
- 4) `integer`

10.2 Realizar las definiciones de tipo y declaraciones de variables pertinentes para poder hacer las siguientes asignaciones:

- a) `empleado.nombre[3] := 'Casamayor';`
`empleado.sueldo := 2500000;`
- b) `empleado[36].nombre[2] := 'Fontecha';`
`empleado[40].sueldo := 1850000;`
- c) `empleado[14].nombre := 'Adolfo';`
`empleado[20].sueldo := 2000000;`

Solución

- a)

```
TYPE
  reg = RECORD
    nombre: ARRAY[1..3] OF PACKED ARRAY[1..10] OF char;
    sueldo: real; (* >maxint *)
  END;
VAR
  empleado: reg;
```
- b)

```
TYPE
  reg = RECORD
    nombre: ARRAY[1..3] OF PACKED ARRAY[1..10] OF char;
    sueldo: real; (* >maxint *)
  END;
VAR
  empleado: ARRAY[1..40] OF reg;
```

CUESTIONES Y EJERCICIOS RESUELTOS

```
c) TYPE
    reg = RECORD
        nombre: PACKED ARRAY[1..10] OF char;
        sueldo: real; (* >maxint *)
    END;
VAR
    empleado: ARRAY[1...40] OF reg;
```

10.3 Determinar lo que escribe el siguiente programa:

```
PROGRAM Cuestion_10_3 (output);
CONST
    n=100 ;
TYPE
    estado_civil = (s,c,v,d);
    registro = RECORD
        dni:ARRAY [1..8] OF char;
        nombre:ARRAY [1..60] OF char;
        sexo:(hombre,mujer);
        estado:estado_civil;
        CASE estado_civil OF
            c:(Conyuge:RECORD
                dni:ARRAY [1..8] OF char;
                nombre:ARRAY [1..60] OF char
            END);
            s,v,d:()
        END;
    matriz = ARRAY [1..n] OF registro;
VAR
    a: matriz;
    i,j,k: integer;
    aux: char;
    regaux: registro;
BEGIN
    FOR i:=1 TO n DO
        WITH a[i] DO
            BEGIN
                FOR j:=1 TO 8 DO
                    dni[j]:=Chr(Ord('0')+i MOD 10);
                FOR j:=1 TO 60 DO
                    nombre[j]:=Chr(Ord('A')+ i MOD 10);
                IF Odd(i)
                    THEN sexo:=mujer
                    ELSE sexo:=hombre;
                k := i MOD 4;
                CASE k OF
                    0 : BEGIN
                            estado:=c;
                            WITH conyuge DO
                                BEGIN
                                    FOR k:=1 TO 8 DO
                                        dni[k]:=Chr(Ord('0')+(i+1) MOD 10);
                                    FOR k:=1 TO 60 DO
                                        nombre[k]:=Chr(Ord('A')+(i+1)MOD 10)
                                    END;
                                1 : estado := s;
                                2 : estado := v;
                                3 : estado := d;
                            END; (* CASE *)
                        END; (* WITH *)
```

ESTRUCTURA DE DATOS REGISTRO

```
FOR j:=2 TO n DO
  FOR i:=n DOWNT0 j DO
    IF a[i-1].dni<a[i].dni THEN
      BEGIN
        regaux:=a[i];
        a[i]:=a[i-1];
        a[i-1]:=regaux
      END;
END;

WITH a[27] DO
  BEGIN
    FOR j:=1 TO 8 DO
      Write (dni[j]);
    Writeln;
    FOR j:=1 TO 60 DO
      Write (nombre[j]);
    Writeln;
    IF sexo=hombre
      THEN Writeln ('Hombre')
      ELSE Writeln ('Mujer');
    CASE estado OF
      c: BEGIN
        WITH conyuge DO
          BEGIN
            Write('casado/a con ');
            FOR j:=1 TO 60 DO
              Write(nombre[j]);
            Writeln;
            FOR j:=1 TO 8 DO
              Write (dni[j]);
            Writeln
          END;
        END;
      s,v,d: Writeln('No casado');
    END; (* CASE *)
  END; (* WITH *)
END.
```

Solución

En primer lugar hay que aclarar que el programa no tiene ninguna utilidad concreta. Se ha escrito para ilustrar el uso de *registros variantes* y estructuras de datos combinadas, concretamente tablas (*arrays de registros*).

La primera sentencia del bloque principal del programa es un bucle *FOR*, cuya única finalidad es rellenar la estructura de datos para poder trabajar con ella, sin tener que teclear datos para cada campo de cada elemento (Tenemos cien elementos con cuatro o seis campos cada uno). Para que los datos cambien con el contador se utiliza el operador *MOD*. Después de este bucle *FOR*, el contenido de la estructura es el representado en la tabla 10.2.

CUESTIONES Y EJERCICIOS RESUELTOS

i	nombre	dni	sexo	estado	nombre C.	dni Con.
1	BBBB ... BBB	11111111	mujer	s	-	-
2	CCCC ... CCC	22222222	hombre	v	-	-
3	DDDD ... DDD	33333333	mujer	d	-	-
4	EEEE ... EEE	44444444	hombre	c	FFFF ... FFF	55555555
5	FFFF ... FFF	55555555	mujer	s	-	-
...
10	AAAA ... AAA	00000000	hombre	v	-	-
11	BBBB ... BBB	11111111	mujer	d	-	-
...
100	AAAA ... AAA	00000000	hombre	c	BBBB ... BBB	11111111

Tabla 10.2 Ejemplo de uso de una *tabla* (array de registros)

Obsérvase que cada diez elementos se repite el contenido de los campos *nombre* y *dni* (Se han rellenado utilizando la expresión $i \text{ MOD } 10$). Para los elementos pares el campo *sexo* toma el valor *hombre* y para los impares *mujer*. Los elementos cuyo campo *estado* toma el valor *c*, son los que tienen el campo variante *conyuge* (de tipo *registro*), y ocupan posiciones múltiplo de 4 ($i \text{ MOD } 4 = 0$).

A continuación mediante un doble bucle *FOR* se ordena la estructura mediante el *algoritmo de la burbuja*, ya estudiado, según el campo *dni* (de mayor a menor). Después de la ordenación, los campos *nombre* y *dni* quedan como aparecen en la tabla 10.3.

Se puede deducir como quedan los elementos conociendo como han sido introducidos, y como funciona el algoritmo de la burbuja. El último fragmento de programa es una sentencia *WITH*, para escribir por pantalla el contenido del elemento nº 27. En las posiciones 21 a 30 quedan los 10 elementos con *dni* = 77777777. Antes estaban en posiciones tales que $i \text{ MOD } 10 = 7$, es decir: 7, 17, 27, ..., 97. Todos ellos son impares, luego el campo *sexo* de todos es *mujer*. No hay ningún múltiplo de 4, luego ninguno tiene campo *conyuge*. Por lo tanto, para el elemento nº 27 el programa escribirá:

```
77777777
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
Mujer
No casado
```

ESTRUCTURA DE DATOS REGISTRO

i	nombre	dni	sexo	estado	nombre C.	dni Con.
1	JJJ ... JJJ	99999999	mujer			
2	JJJ ... JJJ	99999999	mujer			
...			
10	JJJ ... JJJ	99999999	mujer			
11	III ... III	88888888	hombre			
12	III ... III	88888888	hombre			
...			
20	III ... III	88888888	hombre			
21	HHH ... HHH	77777777	mujer			
...			
27	HHH ... HHH	7777777	mujer			
...			
100	AAA ... AAA	00000000	hombre			

Tabla 10.3 Ordenación de la tabla 10.2

10.4 Escribir un subprograma para listar por pantalla el contenido del array a del ejercicio anterior, deteniendo la salida cada 3 elementos para poder leer los datos.

Solución

```

PROCEDURE EscribeTabla(VAR t: matriz);
VAR
  i,j: integer;
BEGIN
  FOR i:=1 TO 100 DO
    WITH t[i] DO
      BEGIN
        Writeln('***** Elemento nº ',i,' *****');
        FOR j:=1 TO 8 DO Write (dni[j]);
        Writeln;
        FOR j:=1 TO 60 DO Write (nombre[j]);
        Writeln;
        IF sexo=hombre
          THEN Writeln ('Hombre')
          ELSE Writeln ('Mujer');
        CASE estado OF
          c:BEGIN
            WITH conyuge DO
              BEGIN
                Write('casado/a con ');
                FOR j:=1 TO 60 DO Write(nombre[j]);
                Writeln;
                FOR j:=1 TO 8 DO Write (dni[j]);

```

CUESTIONES Y EJERCICIOS RESUELTOS

```
        Writeln
        END;
    END;
    s,v,d:Writeln('No casado');
END; (* CASE *)
IF i MOD 3 = 0
    THEN Readln; (* Para detener la pantalla *)
END; (* WITH *)
END; (* EscribeTabla *)
```

10.5 Dadas las siguientes declaraciones:

```
TYPE
    estadoCivil = (soltero, casado, viudo, divorciado);
    sex = (m,f);
    info = RECORD
        nombre: String[30];
        edad: integer;
        sexo: sex;
    END;
    ficha = RECORD
        datos:info;
        estado: estadoCivil;
        CASE estadoCivil OF
            casado: (conyuge:info);
            soltero, viudo, divorciado: ();
        END;
    tabla = ARRAY [1..100] OF ficha;
```

a) Construir un subprograma en Pascal para mostrar por pantalla el contenido de una variable de tipo `ficha`.

b) Escribir otro subprograma que reciba como parámetros una variable de tipo `tabla` y el número de elementos que contiene, y utilizando el subprograma del apartado anterior, saque un listado por pantalla con los datos de los matrimonios en que ambos cónyuges tienen menos de 30 años.

Solución a)

```
PROCEDURE ListaFicha(VAR fichAux:ficha);
BEGIN
    WITH fichAux DO
        BEGIN
            Writeln('Datos personales:');
            Write(datos.nombre:30,' de ', datos.edad:2,' años, ');
            IF datos.sexo=m
                THEN Write('hombre, ')
                ELSE Write('mujer, ');
            CASE estado OF
                casado: BEGIN
                    Writeln(' casado/a con:');
                    Write(conyuge.nombre:30, ' de ');
                    Write(conyuge.edad:2,' años, ');
                    IF conyuge.sexo=m
                        THEN Writeln('mujer.')
                        ELSE Writeln('hombre.');
```

ESTRUCTURA DE DATOS REGISTRO

```
END; (* CASE estado *)
END; (* WITH aux *)
Write('Pulsa <INTRO> para ver siguiente ficha...');
Readln;
END; (* ListaFicha *)
```

Solución b)

```
PROCEDURE Listado(VAR t: tabla; n: integer);
VAR
  i:integer;
BEGIN
  Writeln('Matrimonios menores de 30 años:');
  FOR i:=1 TO n DO
    WITH t[i] DO
      IF estado = casado
        THEN IF (datos.edad<30)AND(conyuge.edad<30)
              THEN ListaFicha(t[i]);
    END;
  END;
```

10.6 Se dispone de un *array de registros*, en el que cada elemento contiene el nombre de un alumno y sus notas de teoría (n_t), práctica (n_p) y nota final (n_f) de cierta asignatura. Para los no presentados a alguna parte, la nota correspondiente vale -1.0. Se pide:

a) Declaraciones de tipo necesarias para manejar dicha estructura de datos.

b) Subprograma para leer de teclado el nombre de cada alumno y sus notas de teoría y práctica, que almacene dichos datos en un parámetro del tipo adecuado. Además debe contar el número de alumnos introducidos y devolverlo al punto de llamada.

c) Subprograma que calcule la nota final (n_f) de cada alumno y la almacene en dicha estructura de datos. n_f se calcula así:

- Si n_t y n_p son ≥ 5.0 , será la media de ambas

- En caso contrario, será la menor de ambas.

d) Subprograma que calcule la media de las notas finales de los presentados.

Solución a)

```
CONST
  nMax = 100;
TYPE
  ficha = RECORD
    nombre: string[50];
    nt,np,nf: real;
  END;
  vector = ARRAY[1..nMax] OF ficha;
```

Solución b)

```
PROCEDURE LeeDatos(VAR i:integer; VAR w:vector);
VAR
  i: integer;
```


CUESTIONES Y EJERCICIOS RESUELTOS

```
    otro: char;
BEGIN
  i:=0;
  REPEAT
    i:=i+1;
    WITH w[i] DO
      BEGIN
        Writeln('Alumno n° ',i,' : ');
        Write('¿ Nombre?: ');
        Readln(nombre);
        Write('¿ Nota de teoría?');
        Readln(nt);
        Write('¿ Nota de prácticas?');
        Readln(np);
      END;
    Write('¿ Más datos (s/n)?');
    Readln(otro);
  UNTIL (otro = 'n') OR (otro = 'N');
END;
```

Suponiendo declaradas las variables:

```
VAR
  num: integer;
  v: vector;
```

El subprograma anterior podría utilizarse mediante la llamada:

```
LeeDatos(num, v);
```

Solución c)

```
PROCEDURE CalculoNf( VAR n: integer; w: vector);
VAR
  i: integer;
  (*****
  FUNCTION NotaFinal (nt, np: real): real;
  CONST aprobado = 5.0;
  BEGIN
    IF ((nt >= aprobado) AND (np >= aprobado))
      THEN NotaFinal := (nt + np)/2
      ELSE
        IF (nt >= np) THEN NotaFinal := nt
        ELSE NotaFinal := np;
  END; (* NotaFinal *)
  (*****

BEGIN (* CalculoNf *)
  Writeln('Calculando notas finales ');
  FOR i:=1 TO n DO
    WITH w[i] DO
      BEGIN
        nf:=NotaFinal(nt,np);
        Write(' ');
      END;
    Writeln;
  END; (* CalculoNf *)
```

ESTRUCTURA DE DATOS REGISTRO

Con las declaraciones de variables del apartado anterior, la llamada a este subprograma sería:

```
CalculoNf(num, v);
```

Solución d)

```
FUNCTION MedP(n:integer; VAR w:vector):real;
(* Utilizamos VAR con w para ahorrar memoria *)
VAR
  i, cont:integer;
  suma:real;
BEGIN
  suma:=0;
  cont:=0;
  FOR i :=1 TO n DO
    IF (w[i].nf <> -1) THEN
      BEGIN
        suma:=suma + w[i].nf;
        cont:=cont+1;
      END;
    MedP:=suma/cont;
  END;
```

Con las declaraciones de variables de los apartados anteriores, la llamada a este subprograma podría ser:

```
Writeln('Nota final media de los presentados: ', MedP(num, w):4:1);
```

10.12 CUESTIONES Y EJERCICIOS PROPUESTOS

10.7 Escribir la declaración de una estructura de datos *registro* para almacenar la información del carnet de identidad de todos los españoles.

10.8 Dada la declaración:

```
TYPE
  fecha = RECORD
    dia, mes, anio: integer;
  END;
  registro = RECORD
    nombre: string[20];
    fechaNac: fecha;
    titulo: ARRAY [1..5, 1..5] OF char;
  END;
  matriz = ARRAY [1..10] OF registro;
VAR
  a: matriz;
```

Indicar el tipo de las siguientes variables:

a) a[5]

b) a[8].nombre

CUESTIONES Y EJERCICIOS PROPUESTOS

- c) `a[8].fechaNac`
- d) `a[8].fechaNac.mes`
- e) `a[5].titulo[1,2]`

10.9 Dadas las siguientes declaraciones:

```
TYPE
  est = (COU, FP);
  reg = RECORD
    clave: integer;
    nombre: string[30];
    edad: integer;
    CASE estudios: est OF
      COU: (n1, n2, n3: real;
            select: boolean);
      FP: (m1, m2, m3: real);
    END;
  tabla = ARRAY [1..100] OF reg;
```

Se pide:

- a) Subprograma para rellenar la estructura de datos.
- b) Porcentaje de estudiantes de COU de cada edad entre los 18 y los 40 años.
- c) Listado de estudiantes de COU con selectividad aprobada (campo `select=TRUE`)
- d) Listado de estudiantes de FP con nota media $((m1+m2+m3)/3)$ entre 5 y 7

10.10 El Municipio de Oviedo ha encargado un estudio con objeto de confeccionar zonas azules (zonas de estacionamiento de vehículos limitado y de pago). Supongamos que la información está organizada en dos tablas, llamadas `censo` y `vehiculos`, cuyos elementos constan de los siguientes campos:

Tabla `censo`:

- Apellidos y nombre, DNI
- Fecha de nacimiento, lugar de nacimiento (ciudad, provincia, país)
- Domicilio actual, teléfono
- Profesión
- Si tiene carnet de conducir: Fecha de expedición
- Si está casado, nombre y DNI del cónyuge

ESTRUCTURA DE DATOS REGISTRO

Tabla vehículos:

Matrícula
Nº de bastidor
Marca
Fecha de matriculación
Impuesto de circulación
DNI del propietario

Se pide:

- a) Escribir las declaraciones necesarias para utilizar ambas tablas.
- b) Diseñar subprogramas para mantenimiento de la tabla `censo` (Inserción, borrado y modificación de un elemento).
- c) Listado de los ciudadanos que tienen coche, incluyendo nombre, dni, nº de vehículos, y cuánto pagan de impuestos de circulación. El listado deberá estar ordenado por tasas de circulación de mayor a menor.
- d) Dada una determinada calle, listado de coches cuyos propietarios viven en ella, con objeto de confeccionar zonas azules.

10.11 El Ministerio de Hacienda nos ha encargado un listado de las personas que no han declarado, con el fin de realizar inspecciones fiscales. Disponemos de una tabla con la relación de todos los ciudadanos obligados a declarar, cuyos elementos son *registros* de la forma:

```
RECORD
  nombre: string[50];
  dni: string[10];
END;
```

Una vez concluido el periodo de declaración, hemos confeccionado otra tabla con la relación de aquellos que han presentado su declaración. Sus componentes son de la forma:

```
RECORD
  nombre: string[50];
  dni: string[10];
  cantidad: longInt; (* entero largo *)
END;
```

Se pide:

- a) Listado de aquellas personas que no han declarado.
- b) Listado de los ciudadanos que han declarado, ordenado de mayor a menor por el campo `cantidad`, separado en tres partes: Declaraciones positivas, nulas y negativas.

10.12 Completar el Tipo Abstracto de Datos Fecha implementado mediante una *unit* en el ejemplo 10.16. Añadir una función para comprobar si una fecha leída es válida, procedimientos para cambiar una fecha de formato *dd-mm-aa* a formato *aa-mm-dd*, un procedimiento para ordenar fechas cronológicamente, etc.

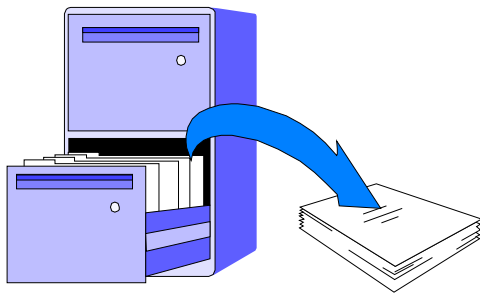
10.13 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

La estructura de datos *registro* se estudia en casi todos los libros de programación a nivel básico. En algunos textos se trata solo a nivel superficial, mientras que otros incluyen el estudio de registros jerárquicos y registros con variantes. Es raro encontrar ejemplos con variantes anidadas. El libro ya mencionado en las recomendaciones bibliográficas del capítulo 8, escrito por *William I. Salmon*, titulado *Introducción a la computación con Turbo Pascal*, y publicado por *Addison-Wesley Iberoamericana* en 1993; incluye un detallado estudio de los Tipos Abstractos de Datos, con ejemplos de TAD's para cada estructura de datos estudiada. En el caso de los *registros*, pone como ejemplo un TAD para manejar números complejos, con los procedimientos y funciones necesarios para operar con complejos. Entre los clásicos, se trata muy bien este tema en la obra *Programación en Pascal*, de *P. Grogono*, publicado también por *Addison Wesley Iberoamericana* en 1986.

La arquitectura de los microprocesadores de la familia 80x86 está descrita en profundidad en el libro titulado *Arquitectura, programación y diseño de sistemas basados en microprocesadores (8086/80186/80286)* de *Yu-Cheng Liu* y *Glenn A. Gibson*, publicado por la Ed. Anaya (1990). Sobre la programación con las directivas ASM e INLINE, consultar el capítulo titulado *El ensamblador incorporado* de la *guía del lenguaje*, dentro de los manuales del compilador Turbo Pascal. Sobre la programación en lenguaje ensamblador pueden verse las referencias hechas en el capítulo 7, en el apartado de ampliaciones y notas bibliográficas.

El listado con todas las interrupciones de la BIOS, y del MS-DOS puede encontrarse en las obras de *Ray Duncan* tituladas *La ROM BIOS de IBM* (Ed. Anaya, 1989), *Funciones del MS-DOS* (Ed. Anaya, 1989), y *Extensiones del MS-DOS* (Ed. Anaya, 1989). Un estudio más comentado puede verse en el libro de *Peter Norton* y *Richard Wilton* titulado *Guía del programador para el IBM PC y PS/2* (Ed. Anaya, 1989). La compañía *Microsoft*, a través de la editorial *Microsoft Press*, publica periódicamente el libro titulado *MS-DOS programmer's reference*, que contiene la definición oficial de las funciones soportadas por las distintas versiones del MS-DOS.

La programación de interrupciones con Turbo Pascal para el manejo de distintos periféricos se estudia en los libros: *Turbo Pascal para IBM-PC y compatibles*, de *P. Philipot* (Ed. Gustavo Gili, 1987); y *Turbo Pascal. Técnicas avanzadas de programación en entorno MS-DOS*, de *C.C. Edwards* (Ed. Anaya, 1989).



CAPITULO 11

FICHEROS

CONTENIDOS

- 11.1 Introducción
- 11.2 Definición de ficheros en Pascal
- 11.3 Ficheros internos y externos
- 11.4 Proceso de escritura en ficheros
- 11.5 Lectura de ficheros
- 11.6 Buffers de ficheros
- 11.7 Ficheros de texto
- 11.8 Ficheros estándar *Input y Output*
- 11.9 Extensiones del compilador Turbo Pascal
- 11.10 Representación interna de los ficheros
- 11.11 Los ficheros como Tipos Abstractos de Datos
- 11.12 Ficheros homogéneos y no homogéneos
- 11.13 Manejo de ficheros en redes
- 11.14 Ejercicios resueltos
- 11.15 Ejercicios propuestos
- 11.16 Ampliaciones y notas bibliográficas

INTRODUCCION

11.1 INTRODUCCION

Los ficheros, también denominados archivos, corresponden a la traducción de la palabra inglesa **FILE**. Se puede definir fichero como *una unidad de información que tiene asignado un nombre*. También se puede definir como *una estructura de datos consistente en una secuencia de componentes, todos del mismo tipo*. Los ficheros o conjuntos de datos tienen una existencia independiente de los programas en proceso (fig. 11.1).

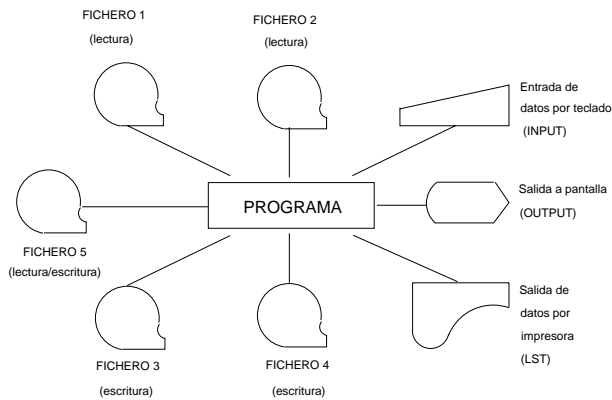


Figura 11.1 Ficheros y programas

En teoría se puede acceder a la información contenida en un fichero de distintas formas:

- acceso secuencial
- acceso aleatorio o directo
- acceso secuencial indexado

El lenguaje Pascal estándar utiliza solamente el **acceso secuencial**. Los ficheros secuenciales pueden representarse como varios componentes o elementos del mismo tipo, uno a continuación del otro. Para acceder a un elemento de un fichero secuencial es necesario recorrerlo desde el primer elemento del fichero.

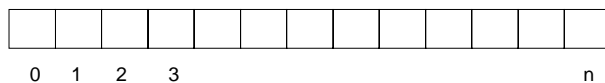


Figura 11.2 Ficheros secuenciales

Para leer o escribir en un fichero secuencial hay que recorrerlo elemento a elemento, desde el primero hasta donde se desee llegar, es decir se lee o escribe secuencialmente. La imagen de lectura/escritura en un fichero sería la de una cinta magnética moviéndose enfrente de una cabeza de lectura/escritura (fig. 11.3).

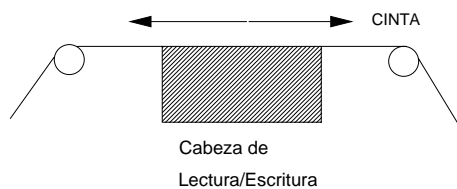


Figura 11.3 Acceso en ficheros secuenciales

Los ficheros de **acceso aleatorio o directo** se representan en la figura 11.4, de forma que dada la posición de un elemento (por ejemplo por medio de un índice o clave), se obtiene directamente el elemento del fichero, sin tener que recorrer los elementos anteriores.

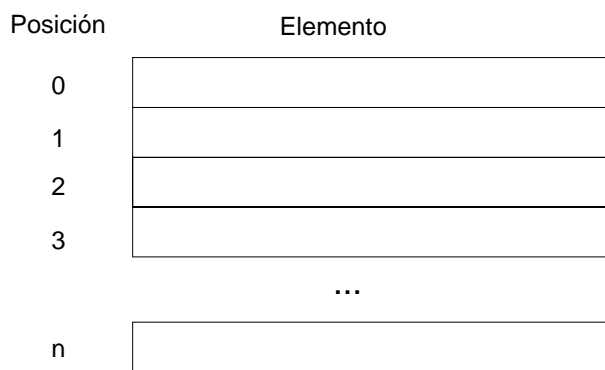


Figura 11.4 Ficheros directos

Para acceder a un elemento, sólo hace falta indicar la posición del elemento (dada en general por un índice numérico). La imagen del acceso directo es un disco, en el que para un radio y un ángulo dado se accede a cualquier zona:

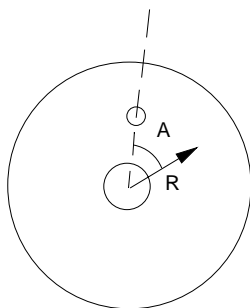


Figura 11.5 Acceso en ficheros directos

INTRODUCCION

Casi todas las implementaciones del lenguaje Pascal permiten el acceso aleatorio o directo a los ficheros, como ampliación del Pascal estándar.

En los ficheros de **acceso secuencial indexado**, existen tres áreas:

- área de índices
- área principal
- área de desbordamiento (overflow)

El *área de índices* contiene las claves de forma secuencial del último registro de cada bloque físico y la dirección de acceso al primer registro del bloque.

El *área principal* contiene, clasificados de forma ascendente por clave, los registros de datos.

El *área de desbordamiento* contiene aquellos registros que no pueden incluirse en el *área principal* cuando se realizan actualizaciones en el fichero.

Cada fichero tiene asociado un **descriptor de fichero** que es un bloque de control con información referente al fichero que necesita el *sistema operativo* para su administración. El bloque de control posee una estructura que depende mucho del sistema pudiendo contener

- nombre simbólico del fichero
- localización del fichero en disco (memoria secundaria)
- organización del fichero (secuencial, directo, ...)
- tipo de dispositivo
- datos para el control de acceso
- tipo de fichero (datos, objeto, ...)
- tratamiento (permanente o temporal)
- fecha y hora de creación
- fecha de borrado
- fecha y hora de la última modificación

Por lo general los descriptores están almacenados en memoria secundaria (disco duro o disquete) y se transfieren a memoria principal (RAM) cuando se abre un fichero.

Un usuario no puede hacer referencia directa a un descriptor ya que es controlado por el *sistema operativo*.

11.2 DEFINICION DE FICHEROS EN PASCAL

En el lenguaje Pascal estándar los ficheros son todos de *acceso secuencial*. Los ficheros se declaran en Pascal como variables de tipo fichero con unos componentes determinados, la sintaxis de la definición del tipo fichero en notación EBNF y el diagrama sintáctico se muestran a continuación.

```
<tipo archivo> ::= FILE OF <tipo>
```

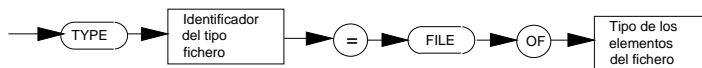


Figura 11.6 Definición de ficheros

Ejemplo 11.1

```
a) TYPE
    precipitacion = FILE OF real ;
VAR
    anio1, anio2, mes1, mes2 : precipitacion;
    ...
```

Aquí se han declarado cuatro ficheros `anio1`, `anio2`, `mes1` y `mes2` cuyos elementos son números reales.

```
b) TYPE
    fichas = RECORD
        nombre,
        apell1,
        apell2 : PACKED ARRAY[1..20] OF char;
        edad : integer;
        peso : real;
    END;
VAR
    personasSanas,
    personasEnfermas: FILE OF fichas;
    ...
```

Aquí se declaran dos ficheros `personasSanas` y `personasEnfermas` cuyos elementos son del tipo registro `fichas`.

Las variables de tipo fichero *se pueden usar como argumentos en las llamadas a procedimientos o funciones*, pero no de ninguna otra forma (por ejemplo, no es válida la sentencia de asignación de un fichero completo a otro). Además, en el paso a un procedimiento o función, deberá utilizarse obligatoriamente la transferencia **por dirección** (parámetros *VAR*).

No está permitido generalmente un fichero de ficheros.

11.3 FICHEROS INTERNOS Y EXTERNOS

En general, hay dos tipos de ficheros que pueden ser generados y a los que se puede acceder desde un programa: **ficheros permanentes** y **ficheros temporales**.

PROCESO DE ESCRITURA EN FICHEROS

Los *ficheros permanentes* se mantienen en un dispositivo de memoria auxiliar (por ejemplo: disco duro o disquetes), y por tanto se pueden guardar después de completar la ejecución de un programa. El contenido de tales ficheros, es decir, los componentes individuales del fichero, pueden ser recuperados y/o modificados en cualquier momento, bien por el programa que creó el fichero o por cualquier otro programa. Los ficheros permanentes se denominan en Pascal **ficheros externos**.

Los *ficheros temporales* se almacenan en la memoria principal (RAM) del ordenador, o en memoria auxiliar (disco duro o disquete). Un fichero temporal se pierde tan pronto como se termina la ejecución del programa que creó el fichero. Los ficheros temporales se denominan en Pascal **ficheros internos**. Se emplean como almacenamiento auxiliar, durante la ejecución del programa.

Los ficheros externos deben transferirse a un programa Pascal como parámetros. Esto se realiza incluyendo los nombres de los ficheros externos en la cabecera del programa.

Ejemplo 11.2

```
PROGRAM Muestra ( input, output, clientes );
TYPE
  estado = (deudor, alDia, atrasados) ;
  cuenta = RECORD
    nombre : PACKED ARRAY [1..80] OF char;
    numero : 1..20000;
    tipo   : estado;
    saldo  : real;
  END;
VAR
  clientes : FILE OF cuenta ;
  ...
```

Esta especificación de los ficheros como parámetros del programa es necesaria para indicarle al ordenador que se trata de un fichero externo.

El lenguaje Pascal incluye dos ficheros predefinidos estándar *input* y *output*, tal como se verá más adelante.

Obsérvese que *la longitud de un fichero nunca se especifica dentro de la definición de fichero*. Esta situación es diferente a la de otros tipos de datos estructurados, donde el número de componentes se especifica dentro de la definición de tipo, bien explícitamente (caso de los arrays) o bien implícitamente (caso de los registros). Como regla general, la longitud máxima de un fichero está determinada sólo por la *capacidad física del medio en que está almacenado*.

Esta propiedad hace que se usen los ficheros internos en algunas aplicaciones.

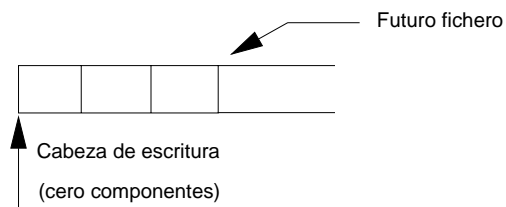
11.4 PROCESO DE ESCRITURA EN FICHEROS

Un fichero en Pascal se escribe o se crea componente a componente, por medio de dos procedimientos predefinidos por el lenguaje:

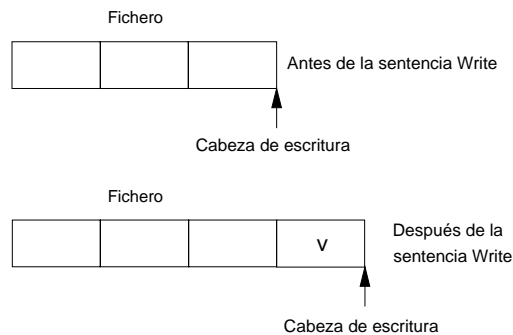
```
Rewrite (fichero);
Write (fichero, v);
```

FICHEROS

El primero de ellos, prepara el fichero para comenzar a escribir, apuntando la cabeza de escritura al principio del fichero. En ese momento el fichero tiene cero componentes. Si el fichero ya está creado, el procedimiento anterior borra la información que contenga.



La acción del procedimiento `Write (fichero, v)` es la siguiente



Para escribir en un fichero se le debe preparar llamando al procedimiento `Rewrite`. Sólo en el caso del fichero estándar de salida `output` no debe hacerse `Rewrite (output)`, pues ya lo hace automáticamente el programa.

Ejemplo 11.3

```
PROGRAM Muestra (Input, Output, clientes);
TYPE
  estado = (deudor, alDia, atrasados);
  cuenta RECORD
    nombre : PACKED ARRAY [1..80] OF char;
    numero : 1..20000;
    tipo   : estado;
    saldo  : real;
  END;
VAR
  clientes : FILE OF cuenta;
  v        : cuenta;
BEGIN
  Rewrite (clientes);
  Write (clientes, v);
  ...
```

El fragmento de programa anterior permite añadir al fichero `clientes` el componente almacenado en la variable `v` cuyo tipo debe ser compatible con los componentes del fichero.

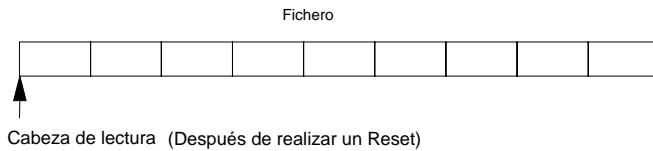
LECTURA DE FICHEROS

11.5 LECTURA DE FICHEROS

Un fichero en Pascal, se lee componente a componente, usando dos procedimientos pre-definidos

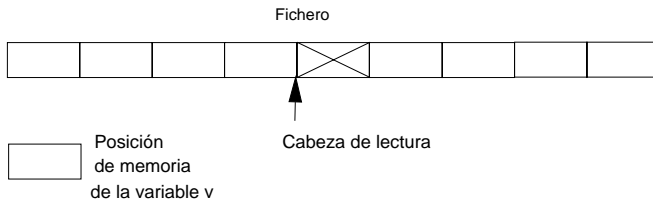
```
Reset ( fichero );
```

Prepara el fichero para comenzar la lectura, posicionándose en el primer componente.

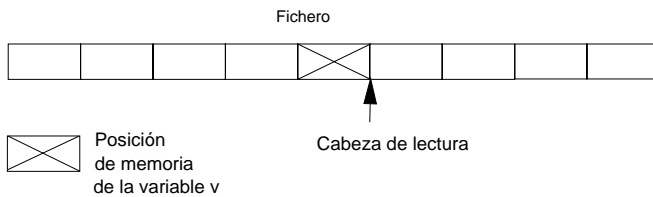


```
Read ( fichero, v );
```

Asigna la siguiente componente del fichero a la variable v y avanza una posición en el fichero. El tipo de v debe coincidir con el tipo del componente del fichero.



después de ejecutar la sentencia `Read(fichero,v);` se tiene:



Para leer un fichero se debe de situar la cabeza de lectura al principio del fichero, por medio de *Reset*. Sólo en el caso del fichero estándar de entrada *input* no debe hacerse `Reset(input)`, pues ya lo hace automáticamente el programa.

FUNCION LOGICA Eof (End of file)

Un procedimiento de lectura de un fichero, producirá un error si ya se ha terminado de leer el fichero. Para comprobar si se ha llegado al final de fichero existe la función lógica o booleana

```
Eof ( fichero )
```

que devuelve:

true si se ha llegado al final del fichero
false si no se ha llegado al final del fichero

Con todo lo anterior, el bucle típico utilizado para la lectura de un fichero es el que se muestra a continuación:

```
Reset (fichero);
WHILE NOT Eof (fichero) DO
BEGIN
  Read (fichero, variable);
  ...
  (* proceso *)
  ...
END;
```

Obsérvese que antes de efectuar cualquier operación de lectura nos aseguramos de que existen componentes en el fichero, es decir; que la función *Eof* no es *true*.

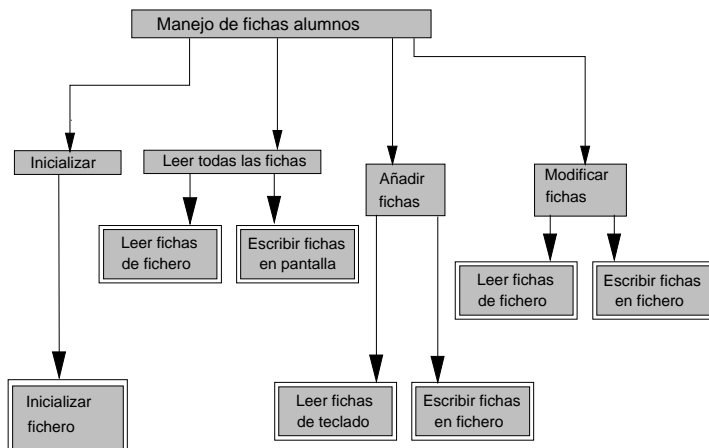
Ejemplo 11.4

Construir un programa que soporte las operaciones básicas con un fichero que contiene el nombre de los alumnos, la nota (calificación), y si han presentado o no las prácticas.

El programa deberá permitir escribir cada ficha de alumno y leerla. Así como añadir y modificar dichas fichas.

Observación: Todos los ejemplos que siguen hasta la sección 11.9 hacen referencia únicamente a Pascal estándar. Para poder implementarlos en Turbo Pascal es necesario incluir nuevas sentencias para el tratamiento de ficheros externos como se verá en la sección 11.9.

Diseño descendente



LECTURA DE FICHEROS

Algoritmo

INICIO

NIVEL 0

Inicializar-fichero
Leer-fichas
Añadir-fichas
Modificar-fichas

NIVEL 1

Inicializar-fichero
Abrir el fichero

Leer-fichas
MIENTRAS existan registros en el fichero HACER
Leer ficha
Visualizar contenido en pantalla
FIN_MIENTRAS

Añadir-fichas
Posicionarse en el último registro del fichero
Crear ficha con nuevos datos
Añadir el nuevo registro al fichero

Modificar-fichas
Leer el número de ficha a modificar
Escribir los nuevos datos
Actualizar el fichero

FIN

Codificación en Pascal

```
PROGRAM Fichas (input,output,alumnos);
CONST m=150;          (* n° máximo de fichas *)
TYPE
  persona = RECORD
    nombre :STRING[80];
    dni:string[10];
    practicas :boolean;
    nota :ARRAY [1..35] OF real;
    END;

  archivo = FILE OF persona;

VAR
  aux:persona;
  vtaux:ARRAY [1..m] OF persona;
  alumnos :archivo;
  opcion,letra : char;
  i,n:integer;
  flag:boolean;

(*****)
```

FICHEROS

```

PROCEDURE Leerfichas(VAR faux:persona);
VAR
  i:integer;
  respu :char;
BEGIN
  WITH faux DO
  BEGIN
    Write('Nombre del alumno: ');
    Readln( nombre);
    Write ('D.N.I: ');
    Readln(dni);
    Write('Entregó las prácticas (s/n) : ');
    Readln(respu);
    IF (respu='s') OR (respu='S')
      THEN practicas:=true
      ELSE practicas:=false;
    Writeln;
    i:=0;
    REPEAT
      i:=i+1;
      Writeln;
      Writeln('Introduzca las notas: valor -1 para ficnalizar ');
      Writeln;
      Write('nota nº',i:2,' = ');
      Read(nota[i]);
    UNTIL nota[i]=-1;
  END;
END;

(***** )

PROCEDURE Escribirfichas(VAR faux:persona);
VAR i:integer;
    respu :char;
BEGIN
  WITH faux DO
  BEGIN
    Writeln('Nombre del alumno: ',nombre);
    Writeln('D.N.I: ',dni);
    Writeln;
    IF practicas
      THEN Write('Si ')
      ELSE Write('No ');
    Writeln('entregó prácticas');
    Writeln;
    i:=1;
    WHILE nota[i] <> -1 DO
    BEGIN
      Writeln('nota nº',i:2,' = ',nota[i]:2:1);
      i:= i + 1;
    END
  END
END;

(***** )

PROCEDURE AnadirFichas (VAR f:Archivo);
BEGIN
  Reset(alumnos);
  i:=0;
  WHILE NOT Eof(alumnos) DO
  BEGIN
    i:=i+1;
    Read(alumnos,vtaux[i])
  END;
  i:=i+1;
  Writeln;

```


LECTURA DE FICHEROS

```

    Writeln('Ficha nº',i:3);
    Writeln;
    leerfichas(vtaux[i]);
    n:=i; (* número total de fichas leídas *)
    Rewrite(alumnos);
    FOR i:=1 TO n DO
        Write(alumnos,vtaux[i])
END;

(*****

PROCEDURE ModificarFichas (VAR f:Archivo);
BEGIN
    Reset(alumnos);
    i:=0;
    WHILE NOT Eof(alumnos) DO
        BEGIN
            i:=i+1;
            Read(alumnos,vtaux[i])
        END;
        n:=i;
        Writeln;
        Write('Introduzca el número de ficha a modificar : ');
        Readln(i);
        Writeln;
        Escribifichas(vtaux[i]);
        Writeln;
        Writeln('Escriba los nuevos datos :');
        Writeln;
        Leerfichas(vtaux[i]);
        Rewrite(alumnos);
        FOR i:=1 TO n DO
            Write(alumnos,vtaux[i])
        END;
END;

(***** Programa principal *****)

BEGIN
    flag:=true;
    REPEAT
        REPEAT
            Writeln;
            Writeln('*****');
            Writeln('*                                     *');
            Writeln('*           MANEJO DE FICHAS DE ALUMNOS           *');
            Writeln('*                                     *');
            Writeln('*****');
            Writeln;
            Writeln('    Introduzca la opción que desea :           ');
            Writeln;
            Writeln('        a) Inicializar                           ');
            Writeln;
            Writeln('        b) Listar fichas                          ');
            Writeln;
            Writeln('        c) Añadir fichas                          ');
            Writeln;
            Writeln('        d) Modificar fichas                       ');
            Writeln;
            Writeln('        e) Fin                                     ');
            Writeln;
            Writeln('        ¿ Que opción desea ? ');
            Readln(opcion);
            UNTIL opcion IN ['a','b','c','d','e','A','B','C','D','E'];
            CASE opcion OF
                'a','A' :BEGIN
                    Rewrite(alumnos);
                    Writeln(' Fichero inicializado ');

```

```

        END;
    'b', 'B' :BEGIN
        Reset(alumnos);
        i:=0;
        WHILE NOT Eof(alumnos) DO
            BEGIN
                i:=i+1;
                (* lee la ficha del fichero y la muestra en pantalla *)
                Read(alumnos,aux);
                Writeln;
                Writeln('Ficha n°',i:3);
                Writeln;
                Escribirfichas(aux);
                Write('Pulse una letra para continuar ');
                Readln(letra)
            END;
        END;
    'c', 'C' :AnadirFichas(alumnos);
    'd', 'D' :ModificarFichas(alumnos);
    'e', 'E' :flag:=false;
END
UNTIL flag=false;
END.

```

11.6 BUFFER DE FICHERO

El lenguaje Pascal utiliza para leer y escribir en los ficheros un *área de memoria intermedia* o **buffer**, en la que *sólo cabe un componente del fichero*.

Se puede acceder directamente a este buffer, ya que siempre que se declara una variable de tipo fichero, *automáticamente se crea otra variable llamada variable buffer del fichero*.

La variable buffer del fichero tiene como identificador el nombre del fichero seguido por una flecha hacia arriba (↑). En la mayoría de los ordenadores esta flecha se representa por: ^ (acento circunflejo).

Si se declara:

```
VAR persona : FILE OF real;
```

La variable buffer del fichero persona es

```
persona^
```

Esta variable buffer es la *ventana* a través de la cual se pueden leer y escribir las componentes del fichero.

Siempre que se hace una operación de lectura o de escritura, se manipula la variable buffer del fichero. De esta manera con el procedimiento *Read (fichero,v)* por ejemplo, se realiza lo siguiente:

- se asigna el siguiente componente del fichero a v
- se avanza una posición de lectura

Antes de leer el fichero:

BUFFER DE FICHERO

```
fichero          a b c d e f g
                  ^cabeza de lectura
variable v = c    variable buffer fichero^ = d
```

Después de leer el fichero:

```
fichero          a b c d e f g
                  ^cabeza de lectura
variable v = d    variable buffer fichero^ = e
```

La variable buffer del fichero contiene una copia de la componente del fichero situada a la derecha de la cabeza de lectura o escritura.

Es decir que la explicación de la sentencia *Read(fichero,v)*, puede hacerse como sigue:

- se asigna a *v* el contenido de la variable buffer del fichero
- se avanza una posición la cabeza de lectura.

Si se desea *se puede utilizar directamente la variable buffer del fichero* sin tener que copiarla en otra variable.

El lenguaje Pascal tiene un procedimiento auxiliar *Get(fichero)* que avanza a la siguiente componente del fichero y actualiza la variable buffer. Así por ejemplo:

```
Read(fichero,v);
```

es equivalente a:

```
BEGIN
  v:= fichero^;
  Get (fichero)
END
```

El concepto de variable buffer de fichero también se puede aplicar a ficheros de salida.

También existe un procedimiento auxiliar *Put(fichero)* que añade el valor de la variable buffer al fichero, y avanza una posición la cabeza de escritura.

El procedimiento *Write(fichero,v)*; equivale a

```
BEGIN
  fichero^ := v;
  Put (fichero);
END;
```

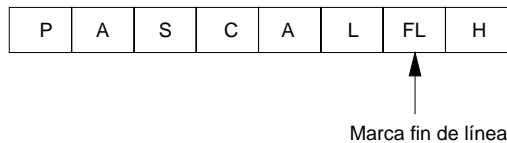
La variable *buffer* es del tipo de los elementos del fichero. Si el fichero es de *texto* es de tipo *char*.

El buffer de fichero tal y como lo define el lenguaje Pascal estándar no está disponible en el compilador Turbo Pascal. Sin embargo define para los ficheros de texto y los ficheros sin tipo (que se estudian en los epígrafes siguientes) otras formas de *buffers* de fichero.

11.7 FICHEROS DE TEXTO

Los ficheros cuyos componentes son caracteres se denominan *ficheros de texto*. La importancia y características peculiares de este tipo de ficheros, justifican su estudio separado del resto.

Los caracteres en un fichero de texto se agrupan en líneas de longitud variable, separadas cada una de ellas entre sí por una *marca de fin de línea*, como se muestra en la figura siguiente.



Las marcas de fin de línea varían de un sistema a otro, pero esto es totalmente transparente al Pascal. Por ejemplo, en los ordenadores que utilizan el código ASCII la marca está constituida por la secuencia de caracteres CR/LF (*carriage-return / line-feed*).

Estas marcas de fin de línea no se pueden leer mediante una sentencia *Read*, (en su lugar se lee un carácter blanco), pero sí podremos detectarlas mediante la función *Eoln* (*End of line*) como se verá en apartados posteriores.

Para declarar variables de tipo *fichero de texto* el Pascal incorpora un tipo predefinido: el tipo *text*. No obstante, este tipo no es equivalente a:

```
FILE OF char      o      PACKED FILE OF char
```

pues la marca de fin de línea puede ser generada y reconocida por procedimientos especiales para manejo de fichero de texto como son: *Writeln*, *Readln*, *Eoln* como se verá a continuación.

Para declarar una variable *fich* como fichero de texto basta poner:

```
VAR
  fich : text ;
```

Además, como ocurre con cualquier otro tipo de fichero, si ese fichero *fich* es permanente (es decir, si existe con anterioridad, o va a existir con posterioridad a la ejecución del programa), deberá incluirse en la lista de parámetros de la cabecera del programa.

PROCESAMIENTO DE FICHEROS DE TEXTO

Los procedimientos incorporados *Reset* y *Rewrite* se usan con ficheros de texto exactamente igual que con otro tipo de ficheros. Así, el procedimiento:

```
Reset (fich);
```

prepara el fichero *fich* para ser leído, posicionándose al principio del mismo y asignando el primer carácter del fichero al *buffer* asociado al fichero (denotado por: *fich^*).

Dado que un fichero de texto es un fichero de tipo *char*, su *buffer* será también de tipo *char*.

FICHEROS DE TEXTO

Análogamente el procedimiento:

```
Rewrite (fich);
```

prepara al fichero para escritura, borrando todo el texto que pudiera contener antes si el fichero ya existía.

Los procedimientos estándar *Get(fich)* y *Put(fich)*, se usan con ficheros de texto exactamente igual que con otros tipos de ficheros.

PROCEDIMIENTOS READ Y WRITE

Los procedimientos *Read* y *Write* ofrecen unas posibilidades ampliadas cuando se usan con ficheros de texto, pues además de la lectura/escritura de caracteres simples, permiten convertir automáticamente datos enteros, reales y booleanos en cadenas de caracteres, en las operaciones de escritura; y viceversa para la lectura (salvo booleanos que no pueden ser leídos).

En concreto, con la sentencia:

```
Read (fich, variable);
```

la acción realizada dependerá del tipo de *variable*: si es de tipo *char* se lee un solo carácter del fichero asignándolo a la variable, a la vez que se avanza la posición de lectura en el fichero y se actualiza el *buffer*. Es decir, en este caso el fichero se procesa de la forma usual, leyendo un único componente del mismo (un carácter).

Sin embargo, si *variable* es de tipo *integer* o *real*, se leerán del fichero una serie de caracteres consecutivos, hasta encontrar un blanco, que compongan un número de ese tipo, el cual será asignado a la variable.

Análogamente, la sentencia:

```
Write (fich, elemento);
```

escribe el *elemento* en el fichero *fich*. El *elemento* puede ser una constante, variable o expresión de los tipos *char*, *integer*, *real* o *boolean*. En los tres últimos casos, sus valores son automáticamente convertidos en cadenas de caracteres.

Otra ampliación de las sentencias *Read* y *Write* para ficheros de texto, consiste en la posibilidad de incluir varios parámetros en una sola sentencia:

```
Read (fich, v1, v2, ... , vN);
```

es equivalente a la secuencia de sentencias:

```
Read (fich, v1);  
Read (fich, v2);  
...  
Read (fich, vN);
```

análogamente ocurre con la sentencia *Write*.

PROCEDIMIENTOS READLN Y WRITELN

Los procedimientos *Readln* y *Writeln* son exclusivos para ficheros de texto. La sentencia

```
Readln (fich);
```

salta al principio de la siguiente línea del fichero de texto *fich* que se está leyendo. Es decir: coloca el puntero de lectura en la posición inmediatamente siguiente a la próxima marca de *fin de línea* que se encuentre.

La sentencia de escritura

```
Writeln (fich);
```

coloca una *marca de fin de línea* en el fichero *fich*. Quiere esto decir que las siguientes salidas hacia el fichero *fich* comenzarán en una nueva línea.

Estos procedimientos también pueden contener varios parámetros, según se muestra a continuación:

```
Readln (fich, v1, v2, ... , vN);
Writeln (fich, v1, v2, ... , vN);
```

las cuales son, respectivamente, equivalentes a:

```
Read (fich, v1, v2, ... , vN);
Readln (fich);
```

y

```
Write (fich, v1, v2, ... , vN);
Writeln (fich);
```

es decir, se leen (o escriben) los valores de (en) la línea actual, y a continuación se salta a la línea siguiente.

FUNCIONES INCORPORADAS PARA FICHEROS DE TEXTO

El Pascal incorpora dos funciones booleanas *Eof* y *Eoln* y un procedimiento *Page* que facilitan el procesamiento de ficheros de texto.

- La función *Eof(fich)* se aplica a los ficheros de texto de igual manera que a otros tipos de ficheros. Esta función devuelve el valor *true* al alcanzar la marca de fin de fichero.
- La función *Eoln(fich)* sirve para detectar las marcas de fin de línea en el fichero *fich* que se le pasa como parámetro. Esta función toma el valor *true* cuando se ha leído el último carácter de una línea; es decir: cuando el puntero de lectura apunta a una marca de fin de línea. En este caso, el buffer *fich^* toma el valor: ' ' (carácter blanco), ya que las marcas de fin de línea no se pueden leer.

FICHEROS DE TEXTO

Ejemplo 11.5

El siguiente programa cuenta el número de espacios en blanco en un fichero de texto.

```
PROGRAM CuentaBlancos (FicheroTexto,output);
CONST
  blanco = ' ';
VAR
  FicheroTexto: text; (* Tipo predefinido *)
  caracter: char;
  nb,nnb: integer;
BEGIN
  nb:= 0;
  nnb:= 0;
  Reset (FicheroTexto);
  WHILE NOT Eof (FicheroTexto) DO
  BEGIN
    WHILE NOT Eoln (FicheroTexto) DO
    BEGIN
      Read (FicheroTexto,caracter);
      IF caracter = blanco
      THEN nb:= nb + 1
      ELSE nnb:= nnb + 1;
    END;
    Readln (FicheroTexto) (* Salta el final de línea *)
  END;
  Writeln ('Número de blancos:',nb);
  Writeln ('Número de no blancos:',nnb)
END.
```

- El procedimiento estándar *Page(fich)* hace que cualquier salida subsiguiente empiece en la cabecera de una nueva página; suponiendo que el dispositivo asociado a *fich* (ver sección 11.9 de este capítulo) pueda reconocer páginas, como por ejemplo una impresora de líneas. Si no es así, el procedimiento *Page* se ignora; si bien en algunos sistemas que utilizan monitor de video, su efecto puede ser borrar la pantalla, aunque esto dependerá de la implementación.

Procesamiento en lectura de ficheros de texto

El bucle generalmente usado para leer, carácter a carácter, un fichero de texto es el siguiente:

```
Reset (fichText);
WHILE NOT Eof (fichText) DO
BEGIN
  WHILE NOT Eoln (fichText) DO
  BEGIN
    Read (fichText, carácter);
    ...
  END;
  Readln (fichText);
END;
```

En escritura es igual pero cambiando *Read* y *Readln* por *Write* y *Writeln*.

11.8 LOS FICHEROS ESTANDAR INPUT Y OUTPUT

Input es el fichero estándar de entrada, por lo que solo está permitido efectuar operaciones de lectura sobre este fichero. Suele estar asignado al teclado.

Análogamente, el fichero *Output* es el fichero estándar de salida. Generalmente se asigna a la pantalla o a la impresora.

Estos ficheros no deben ser declarados como variables (aunque sí deben figurar en la cabecera del programa como ya sabemos); ni tampoco deben usarse los procedimientos *Reset* y *Rewrite* con ellos, pues esto lo hace automáticamente el sistema al comienzo del programa.

Además, cuando en los procedimientos *Read*, *Readln*, *Write* y *Writeln* no se especifica ningún nombre de fichero, el compilador sobreentiende que se refieren a los ficheros *Input* y *Output* respectivamente.

Igualmente ocurre con las funciones *Eof* y *Eoln* si no llevan parámetros. Esto se ilustra a continuación.

Expresión	Expresión equivalente
Eof	Eof (input)
Eoln	Eoln (input)
Read (ch)	Read (input, ch)
Readln	Readln (input)
Write (ch)	Write (output, ch)
Writeln	Writeln (output, ch)
Page	Page (output)

11.9 EXTENSIONES DEL COMPILADOR TURBO PASCAL

La implementación del lenguaje Pascal estándar en Turbo Pascal, requiere a veces incorporar en los programas nuevas sentencias para que puedan ejecutarse sin errores. A continuación se presentan las ampliaciones necesarias para la ejecución de programas que tratan ficheros. No obstante, se puede consultar el manual de Turbo Pascal para un seguimiento más detallado.

ESCRITURA Y LECTURA EN FICHEROS

Se debe poner el procedimiento:

```
Assign (VariableFichero, NombreFichero);
```

donde

- *VariableFichero* es la variable de tipo *FILE* declarada en *VAR*.
- *NombreFichero* es el nombre físico del fichero que se va a utilizar y será almacenado en un dispositivo físico. Puede ser una cadena de caracteres constante o una variable de tipo *string*.

EXTENSIONES DEL COMPILADOR TURBO PASCAL

Por tanto, sirve para asociar una variable de fichero con un fichero del sistema de ficheros del sistema operativo.

Esta sentencia debe estar antes de la primera aparición del procedimiento de escritura

```
Rewrite (NombreFichero);
```

o de lectura

```
Reset (NombreFichero);
```

También debe incluirse el procedimiento

```
Close (NombreFichero);
```

cuando se deje de usar al fichero y cuando se quiera usar el mismo identificador de variable para manejar otro fichero del mismo tipo. Solamente se puede prescindir de la llamada a este procedimiento con ficheros utilizados para lectura, aunque si se incluye no es incorrecto.

Ejemplo 11.6

```
PROGRAM Muestra ( input, output, clientes);
TYPE
  estado = (deudor, alDia, atrasados);
  cuenta RECORD
    nombre : PACKED ARRAY [1..80] OF char;
    numero : 1..20000;
    tipo : estado;
    saldo : real;
  END;
VAR
  clientes : FILE OF cuenta;
  v : cuenta;
BEGIN
  Assign (clientes, 'A:\CLIENTES.DAT');
  Rewrite (clientes);
  Write (clientes, v);
  ...
  Close (clientes);
END.
```

Ejemplo 11.7

```
...
Assign (clientes, 'A:\CLIENTES.DAT');
Reset (clientes);
WHILE NOT Eof (clientes) DO
  BEGIN
    Read (clientes, v);
    ...
    (* proceso *)
    ...
  END;
Close (clientes); (* Se puede suprimir *)
...
```

Todo lo anterior es aplicable en la implementación de ficheros de texto tanto de escritura como de lectura.

```

...
Assign (fichText, 'A:\DATOS.DAT');
Reset (fichText);
WHILE NOT Eof (fichText) DO
  BEGIN
    WHILE NOT Eoln (fichText) DO
      BEGIN
        Read (fichText, carácter);
        ...
      END;
    Readln (fichText);
  END;
Close (fichText);
...

```

Se debe observar que en los ejemplos anteriores se crea o recupera un fichero almacenado físicamente en la unidad *A* del ordenador. Por defecto si no se incluye unidad se buscará el fichero en la unidad activa y directorio activo de esa unidad.

Otro punto a destacar es que en Turbo Pascal se permite abrir un fichero mediante la operación *Reset*, y después escribir en él con el procedimiento *Write*. Este aspecto es importante cuando se desean realizar simultáneamente operaciones de lectura y escritura.

FICHEROS DE ACCESO DIRECTO

Como ya se ha dicho, en el lenguaje Pascal estándar los ficheros son todos de acceso secuencial. Para trabajar con ficheros de acceso directo es necesario incluir algunas de las siguientes llamadas de funciones y procedimientos

- **Procedimiento** `Seek (Fichero, Posicion);`

donde:

Fichero es una variable de tipo fichero.

Posicion es una expresión de tipo *longInt*.

Mueve la cabeza de lectura/escritura de la posición actual a la indicada por *Posicion*. La posición es un entero que indica el número de orden del elemento del fichero. Se comienza a contar por el cero.

- **Función** `SeekEof (Fichero):boolean`

Devuelve un valor booleano indicando si se ha llegado o no al final de fichero.

- **Función** `SeekEoln (Fichero):boolean`

Devuelve un valor booleano indicando si se ha llegado o no al final de línea.

- **Función** `Filepos (Fichero):longInt`

Devuelve la posición actual de la cabeza de lectura/escritura en el fichero. El resultado es de tipo *longInt*.

EXTENSIONES DEL COMPILADOR TURBO PASCAL

- **Función** `Filesize (Fichero):longInt`

Devuelve el tamaño del fichero en *bytes*, y es de tipo *longInt*.

Ejemplo 11.8

Se va a crear un programa que permita ordenar un fichero cuyos registros sean números enteros, de forma ascendente, utilizando el método de la burbuja.

Solución. Si se considera el siguiente vector de números enteros:

Vector A	8	10	3	11
----------	---	----	---	----

para ordenarlo de menor a mayor por el método de la burbuja habrá que dar tres pasadas comparando de dos en dos los elementos consecutivos comenzando, por ejemplo, por el lado derecho. El número menor debe situarse siempre a la izquierda cuando se comparan, de esta manera se asegura que el número más pequeño ocupa la primera posición en la primera pasada y así sucesivamente.

Todo lo anterior se puede resumir en los dos bucles *FOR* siguientes siendo *n* el tamaño del vector *A*.

```
FOR i:= 1 to n-1 DO
  FOR j := n DOWNT0 i+1 DO
    IF A[j] < A[j-1]
      THEN
        BEGIN
          k := A[j];
          A[j] := A[j-1];
          A[j-1] := A[k]
        END;
```

Algoritmo

INICIO

NIVEL 0

Crear-fichero
Ordenar-fichero
Listar-fichero

NIVEL 1

Crear-fichero
MIENTRAS entero <> señal de parada HACER
 Leer número entero
 Escribirlo en el fichero
FIN_MIENTRAS
Ordenar-fichero
 Aplicar el método de la burbuja
Listar-fichero
PARA i = 1 HASTA Tamaño del fichero HACER
 Leer número entero

FICHEROS

```
        Escribirlo en pantalla
    FIN_PARA
FIN
```

Codificación en Pascal

```
PROGRAM FicheroEnteros (input, output, Fenteros);
Uses crt;
TYPE
    Fichero = FILE OF integer;
VAR
    Fenteros:Fichero;
    i:integer;
    st:string[10]; (* Se almacena el nombre del fichero externo *)

{-----}

PROCEDURE Crearfich (VAR Fenteros:Fichero);
{ Crea un fichero cuyo contenido serán números enteros }
VAR
    i:integer;
BEGIN
    Clrscr;
    Writeln(' SE VA A CREAR UN FICHERO DE ENTEROS  ');
    Writeln;
    Writeln(' Introduzca el número -1 para finalizar ');
    Readln(i);
    Rewrite(Fenteros);
    WHILE i <> -1 DO
        BEGIN
            Write(Fenteros, i);
            Readln(i);
        END;
    Close(Fenteros);
END;

{-----}

PROCEDURE Ordenarfich (VAR Fenteros:Fichero);
{ Ordena el fichero previamente creado por el método de la burbuja en
orden ascendente}
VAR
    i,j,k1,k2:integer;
    n:longint;
BEGIN
    Reset(Fenteros);
    n:= filesize(Fenteros); (* Calcula el tamaño del fichero *)
    FOR i:= 1 TO n-1 DO (* Algoritmo del método de la burbuja *)
        FOR j:= n-1 DOWNTO i DO
            BEGIN
                Seek(Fenteros,j); (* Acceso directo a la posición j *)
                Read(Fenteros,k1);
                Seek(Fenteros, j-1);
                Read(Fenteros,k2);
                IF k1 < k2
                    THEN
                        BEGIN
                            Seek(Fenteros,j);
                            Write(Fenteros,k2);
                            Seek(Fenteros, j-1);
                            Write(Fenteros,k1);
                        END;
            END;
        END;
    Close(Fenteros);
END;
```

EXTENSIONES DEL COMPILADOR TURBO PASCAL

```
{-----}

PROCEDURE Listarfich (VAR Fenteros:Fichero);
{ Lista el contenido del fichero }
VAR
  i,k:integer;
BEGIN
  Reset(Fenteros);
  FOR i:= 1 TO Filesize(Fenteros) DO
    BEGIN
      Read(Fenteros, k);
      Write(k:4);
    END;
  END;

{***** Programa principal *****)}

BEGIN
  Clrscr;
  st:= 'A:\enteros.dat'; (* Crea el fichero enteros.dat en la unidad A *)
  Assign(Fenteros,st);
  Crearfich(Fenteros);
  Clrscr;
  Writeln(' CONTENIDO DEL FICHERO ANTES DE SER ORDENADO ');
  Listarfich(Fenteros);
  Ordenarfich(Fenteros);
  Writeln;
  Writeln;
  Writeln(' CONTENIDO DEL FICHERO DESPUES DE SER ORDENADO ');
  Listarfich(Fenteros);
  Writeln;
  Writeln;
  Writeln(' Pulse una tecla para continuar ');
  Readln;
END.
```

DETECCION DE ERRORES DE ENTRADA/SALIDA

Los errores de entrada/salida producen una interrupción brusca de la ejecución del programa, con un mensaje de la forma I/O Error 01, PC=xxxx, donde xxxx es el código de error. Estos errores pueden tener diversas causas: intentar abrir un fichero que no existe, escribir en una unidad de disco sin espacio libre, etc...

El compilador Turbo Pascal incluye las directivas de compilación *{I-}* y *{I+}* para proporcionar un número entero, por medio de la función *IoResult*, que indica el código de error conectado a la última operación de entrada/salida; si este código es cero la operación se ha desarrollado sin problemas, si es distinto de cero es el código del último error detectado. Resumiendo:

- *{I-}* desactiva el control de errores de E/S
- *{I+}* activa el control de errores de E/S
- *IoResult* devuelve un número entero que indica el código de error de la última operación de entrada/salida. Si el código es cero, la operación se ha desarrollado sin

FICHEROS

problemas. A continuación se muestra el procedimiento *ErrorEntradaSalida* que determina la causa de un error de entrada/salida a partir del valor devuelto por *IoResult*.

```
PROCEDURE ErrorEntradaSalida (CodError:integer);
(* Determina la causa de un error de entrada/salida a
partir del código de error suministrado por IoResult *)
BEGIN
Write('Código de error: ', CodError);
CASE CodError OF
  2: Writeln('- Fichero no encontrado');
  3: Writeln('- Path no encontrado cuando se abre');
  4: Writeln('- Demasiados ficheros abiertos');
  5: Writeln('- Apertura de Modo de fichero no válida');
 100: Writeln('- Error de lectura de disco');
 101: Writeln('- Error de escritura en disco');
 102: Writeln('- Fichero no asignado');
 103: Writeln('- Fichero no abierto');
END;
halt(1); (* Detiene la ejecución del programa *)
END;
```

Una forma habitual de tratar el manejo de ficheros es la que se muestra en el siguiente esquema de programa:

```
...
VAR
...
nombreFichero:String;
ok:boolean;
codIoError:integer;
f1:FILE OF ...
...
BEGIN
...
REPEAT
Write('Deme el nombre del fichero: ');
Readln(nombreFichero);
Assign(f1,nombreFichero);
{ $I- } Reset(f1) { $I+ };
codIoError:=IoResult;
ok:=(codIoError=0);
IF NOT ok
THEN
ErrorEntradaSalida(CodIoError);
UNTIL ok;
...
END.
```

Programas completos se muestran en los ejemplos 11.9, y 11.10.

FICHEROS SIN TIPO

El compilador Turbo Pascal también admite un tercer tipo de ficheros: *los ficheros sin tipo*. Los ficheros sin tipo son canales de entrada/salida que se usan principalmente para el acceso directo a cualquier tipo de fichero, independientemente de su tipo y estructura. Un fichero sin tipo se declara con la palabra única *file*, de la forma siguiente:

EXTENSIONES DEL COMPILADOR TURBO PASCAL

```
VAR
    fich:file;
```

A diferencia de los ficheros de texto y de los ficheros con tipo, los procedimientos *Reset* y *Rewrite* con los ficheros sin tipo, pueden llevar un parámetro opcional (un entero de tipo *word*), que indica la longitud del registro en *bytes*. Si se pone de 1 byte, se puede leer cualquier tipo de fichero. Por defecto Turbo Pascal mantiene por razones históricas el valor de 128 bytes. A continuación se muestran dos ejemplos de la nueva forma de uso de *Reset* y *Rewrite*:

```
Reset(fich,1); (* longitud del registro de 1 byte *)
Rewrite(fich, 512) (* longitud de registro de 512 bytes *)
```

Los procedimientos estándar *Read* y *Write* **no se pueden utilizar** con los ficheros sin tipo, el acceso a dichos ficheros se realiza con los procedimientos *BlockRead* y *BlockWrite*, cuyo formato es el siguiente:

```
BlockRead(f, buffer, nRegistros, resultado)
BlockWrite(f, buffer, nRegistros, resultado)
```

donde:

f es una variable de tipo fichero sin tipo.

buffer es una estructura de datos dentro de la cual se van a colocar los datos de forma temporal.

nregistros es el nº de registros que hay que leer o escribir (es de tipo *word*). Una forma habitual de trabajar es la siguiente: si se fijó que la longitud de registro es de un *byte*, con la instrucción *Reset(f,1)*, entonces se puede utilizar la función *SizeOf* sobre la variable *buffer*. *SizeOf(buffer)* devuelve el tamaño en bytes de la estructura de datos *buffer*, dado que se fijó el tamaño de registro en un *byte*, se puede aprovechar al máximo esta estructura, y en el caso de que cambiase la estructura *buffer* no sería necesario modificar los argumentos de los procedimientos *BlockRead* y *BlockWrite*. El tamaño máximo de transferencia de una sólo vez es 65.535 bytes (64K).

resultado es un argumento opcional, que especifica el nº de registros que realmente se leen o escriben (es de tipo *word*). Si no se especifica el parámetro *resultado*, y *Nregistros* es diferente de los realmente escritos o leídos se producirá un error de entrada/salida.

Programa completos que utilizan ficheros sin tipo se muestran en los ejemplos 11.9 y 11.10.

OTRAS OBSERVACIONES SOBRE EL COMPILADOR TURBO PASCAL

- El compilador Turbo Pascal no dispone del procedimiento estándar *Page(fich)*, sin embargo, es equivalente escribir

```
Write (fich,chr(12))
```

donde *chr(12)* es el carácter de salto de página.

- El compilador Turbo Pascal no soporta los procedimientos estándar *Put(fichero)* y *Get(fichero)* para acceder directamente al buffer en lectura y escritura de ficheros.
- *Borrado de ficheros.* El compilador Turbo Pascal incorpora el procedimiento *erase*. La forma de uso es la siguiente:

```
Assign(f, 'basura.dat');
Erase(f);
```

- *Renombrar ficheros.* El compilador Turbo Pascal incorpora el procedimiento *rename*. La forma de uso es la siguiente:

```
Assign(f, 'antiguo.dat');
Rename(f, 'nuevo.dat');
```

- *Determinación del tamaño de un fichero.* El compilador Turbo Pascal incorpora la función *FileSize*, para ficheros con tipo o sin tipo, pero no para los de texto. Devuelve el número de *bytes* (de tipo *longInt*). La forma de uso es la siguiente:

```
...
VAR
  f:file; (* fichero sin tipo *)
...
BEGIN
  Assign(f, 'prueba.dat');
  Reset(f);
  Writeln('Tamaño: ', FileSize(f), ' bytes');
...

```

En el ejemplo 11.9 se utiliza la función *FileSize* para determinar el tamaño de un fichero sin tipo.

- *Definición de un buffer de E/S en ficheros de texto.* El compilador Turbo Pascal incorpora el procedimiento *SetTextBuf* para asignar un *buffer* de entrada/salida a un fichero de texto. También incluye el procedimiento *Flush* para vaciar el *buffer* de un fichero de texto abierto para salida. Para más información consultar el apartado 11.10 *Representación interna de los ficheros* y el manual *Referencia del programador* del compilador Turbo Pascal.
- *Manejo de dispositivos.* El compilador Turbo Pascal permite el manejo de las unidades lógicas del sistema operativo MS-DOS (PRN, CON, AUX, LPT1, LPT2, LPT3, COM1, COM2, COM3, COM4, y NUL) como si fueran ficheros. Se puede acceder a estas unidades lógicas, asignándolas a variables asociadas a ficheros de tipo texto habituales. A continuación se muestra un ejemplo:

```
PROGRAM Probando_LPT1(impresora);
VAR
  impresora: Text;
BEGIN
  Assign(impresora, 'LPT1');
  Rewrite(impresora);
  Writeln(impresora, 'Esta es la línea que sale impresa');
...

```


EXTENSIONES DEL COMPILADOR TURBO PASCAL

- *Secuencias de escape de control de impresoras.* Como se indicó en el punto anterior las impresoras se manejan en Turbo Pascal como si fueran ficheros de texto, pero cada tipo de impresora maneja unos caracteres especiales, o secuencias de caracteres, denominadas *secuencias de escape*. Se denominan secuencias de escape, dado que suelen comenzar con el carácter escape: ESC (127 en la tabla ASCII). Las secuencias de escape permiten colocar el texto en cursiva, negrita, subrayado, o cambiar el tipo de letra. Son diferentes para cada tipo de impresora, y deben consultarse en el manual de la impresora. A continuación se muestra un ejemplo de uso de las secuencias de escape en las impresoras compatibles EPSON e IBM, que se expresan siguiendo las declaraciones del punto anterior:

```
...
Writeln(impresora, Chr(12)); (* Salto de página *)
Writeln(impresora, Chr(27), Chr(69)); (* Negrita activada *)
Writeln(impresora, 'Esta frase sale en negrita');
Writeln(impresora, Chr(27), Chr(70)); (* Negrita desactivada *)
Writeln(impresora, Chr(27), Chr(45), Chr(1)); (* Subrayado activo *)
Writeln(impresora, 'Esto sin negrita, pero subrayado');
Writeln(impresora, Chr(27), Chr(45), Chr(0)); (* Quita subrayado *)
...
```

- *Argumentos en línea de comandos.* El compilador Turbo Pascal incorpora dos funciones *ParamCount* y *ParamStr* para que el programa principal pueda tomar argumentos en línea de comandos. *ParamCount* contiene el número de parámetros que se le pasan al programa principal. Los parámetros son cadenas de caracteres separadas por uno más espacios en blanco. *ParamStr(1)* es una cadena que contiene al primer parámetro, *ParamStr(2)* contiene al segundo parámetro, *ParamStr(3)* al tercero, etc... *ParamStr(0)* contiene el nombre del programa ejecutable completo, junto con su *path*. Su forma de aplicación se puede ver en los ejemplos 11.9 y 11.10.
- *Subprogramas y tipos de datos para el manejo de los ficheros del DOS.* El compilador Turbo Pascal incorpora los subprogramas: *FindFirst*, *FindNext*, *SetFAttr*, *GetFAttr*, *GetFTime*, *SetFTime*, *SetVerify*, *FExpand*, y *DosError*; y los tipos de datos: *SearchRec*, y *FileRec* (véase *unit DOS*). Para más información consultar el apartado siguiente *Representación interna de los ficheros* y el manual *Referencia del programador* del compilador Turbo Pascal.

Ejemplo 11.9

Escribir una versión propia del comando *copy* del MS-DOS. Este ejemplo ilustra como se toman argumentos de la línea de comandos y el manejo de ficheros sin tipo.

Solución. El nuevo comando se denominará *copia*, y su funcionamiento será:

```
copia NombreFicheroOrigen NombreFicheroDestino
```

```
PROGRAM Copia (Origen, Destino);
(* Versión reducida del comando copy del DOS *)
```

FICHEROS

```
VAR
  Origen, Destino: FILE;
  NumLeidos, NumEscritos: Word;
  Buf: ARRAY [1..40960] OF byte; (* 40K de buffer *)
BEGIN
  IF ParamCount<>2 (* Si número de argumentos en línea de comandos <> 2 *)
  THEN
    BEGIN
      Writeln('Forma de uso: COPIA Fich_Origen Fich_Destino');
      Halt(1);
    END;
  Assign(Origen, ParamStr(1)); (* ParamStr(1) es la primera cadena *)
  {$I-} Reset(Origen, 1); {$I+} { Tamaño de registro = 1 byte}
  IF IoResult<>0
  THEN
    BEGIN
      Writeln('No se encontró el fichero: ', ParamStr(1));
      Halt(1);
    END;
  Assign(Destino, ParamStr(2)); (* ParamStr(2) es la segunda cadena *)
  {$I-} Rewrite(Destino, 1); {$I+} { Tamaño de registro = 1 byte}
  IF IoResult<>0
  THEN
    BEGIN
      Writeln('No se puede escribir en el fichero: ', ParamStr(2));
      Halt(1);
    END;
  Writeln('Copiando ', FileSize(Origen), ' bytes');
  REPEAT
    Write('.'); (* Indicador visual del proceso de copia *)
    BlockRead(Origen, Buf, SizeOf(Buf), NumLeidos);
    BlockWrite(Destino, Buf, NumLeidos, NumEscritos);
  UNTIL (NumLeidos = 0) OR (NumEscritos <> NumLeidos);
  Close(Origen);
  Close(Destino);
END.
```

11.10 REPRESENTACION INTERNA DE LOS FICHEROS

La representación de los ficheros se realiza en el soporte de memoria secundaria (disco duro, disquete, etc...), pero cuando se abre un fichero con las operaciones *Reset* o *Rewrite* cada compilador realiza unas determinadas operaciones en memoria principal, que dependen de la organización del sistema de ficheros de cada sistema operativo.

Cuando el compilador Turbo Pascal abre un fichero, usando los procedimientos estándar *Reset* o *Rewrite*, llama internamente a la función 3DH del DOS. Turbo Pascal reserva 128 bytes de memoria para almacenar la información devuelta por la llamada a la función del DOS. La estructura que recibe esta información es el registro *FileRec* definido en la *unit* DOS, y es válida para ficheros con tipo y sin tipo (pero no para fichero de texto).

```
TYPE
  FileRec=RECORD
    Handle:word;
    Mode:word;
    RecSize:word;
    Private:ARRAY[1..26] OF byte;
    UserData:ARRAY[1..16] OF byte;
    Name:ARRAY[1..79] OF char;
  END;
```

REPRESENTACION INTERNA DE LOS FICHEROS

Cada vez que se abre un fichero, el sistema operativo DOS le asigna un descriptor interno único denominado *handle* del fichero. El *handle* es un entero de tipo *word*. Para usar cualquier interrupción o función del DOS relativa a ficheros siempre es necesario pasar el *handle* al procedimiento *MsDos*. Turbo Pascal almacena el *handle* en el primer campo de *FileRec*.

El segundo campo *Mode* de *FileRec* contiene el código de la forma de acceso al fichero. Este campo toma uno de los valores definidos por las constantes *fmxxx* definidas en la *unit DOS*: *fmClosed* \$D7B0, *fmInput* \$D7B1, *fmOutput* \$D7B2, y *fmInOut* \$D7B3.

El tercer campo *RecSize* contiene el tamaño del registro en bytes, definido cuando se abrió el fichero. Este tamaño se utilizará por procedimientos como *Seek* para acceder directamente a un registro, o *FileSize* para determinar el número de elementos del fichero.

El cuarto campo es *Private*, que Turbo Pascal no utiliza. El quinto campo es *UserData* que Turbo Pascal no utiliza, y puede ser utilizado por el usuario para almacenar información interna sobre el fichero. El último campo es una cadena de caracteres reservada para almacenar el nombre del fichero.

En el ejemplo 11.10 se ilustra como acceder a la estructura interna de un fichero.

La representación interna de los fichero de texto ocupa 256 bytes, que se distribuyen según la declaración de tipo *TextRec*.

```
TYPE
  TextBuf=ARRAY[1..127] OF char;
  TextRec=RECORD
    Handle:word;
    Mode:word;
    BufSize:word;
    Private:word;
    BufPos:word;
    BufEnd:word;
    BufPtr:^TextBuf;
    OpenFunc:pointer;
    InOutFunc:pointer;
    FlushFunc:pointer;
    UserData:ARRAY[1..16] OF byte;
    Name:ARRAY[0..79] OF char;
    Buffer:TextBuf;
  END;
```

Los campos *handle*, *mode*, *private*, *userData*, y *name* tienen el mismo significado que los equivalentes de los ficheros con tipo y sin tipo. El resto de los campos están relacionados con el *buffer* del fichero de texto, y algunos de ellos son de tipo *pointer* (puntero sin tipo), que se estudiará en el capítulo 12.

Ademas de la estructura interna de los ficheros, el compilador Turbo Pascal proporciona una variable denominada *FileMode* de tipo *byte*, que determina el código de acceso que se pasa al sistema operativo DOS cuando se abren ficheros con tipo y sin tipo usando el procedimiento *Reset*. Los valores posibles de *FileMode* son los siguientes:

- 0 Sólo lectura
- 1 Sólo escritura
- 2 Lectura/escritura
- 64 Sólo permite acceso de lectura en red
- 65 Sólo permite acceso de escritura en red
- 66 Permite acceso de lectura/escritura en red

Los cuatro últimos valores de *FileMode* son para el uso de ficheros en redes. El valor por defecto de *FileMode* es 2, lo que permite tanto la lectura como la escritura.

Ejemplo 11.10

Este ejemplo ilustra el acceso a la estructura interna de un fichero.

```
PROGRAM Informa (fichero);
USES Dos;
VAR
  Fichero: FILE;
  i: integer;
BEGIN
  IF ParamCount<>1 (* Si número de argumentos en línea de comandos <> 1 *)
  THEN
    BEGIN
      Writeln('Forma de uso: INFORMA nombre_Fichero');
      Halt(1);
    END;
  Assign(Fichero, ParamStr(1)); (* ParamStr(1) es la primera cadena *)
  {$I-} Reset(Fichero, 1); {$I+} { Tamaño de registro = 1 byte}
  IF IoResult<>0
  THEN
    BEGIN
      Writeln('No se encontró el fichero: ', ParamStr(1));
      Halt(1);
    END;
  Writeln('Información interna del fichero');
  Writeln('Handle: ', FileRec(fichero).handle);
  Writeln('Modo: ', FileRec(fichero).mode);
  Writeln('Tamaño de registro: ', FileRec(fichero).RecSize);
  Write('Privado: ');
  FOR i:=1 TO 26 DO Write(FileRec(fichero).Private[i]);
  Writeln;
  Write('Datos del usuario: ');
  FOR i:=1 TO 16 DO Write(FileRec(fichero).UserData[i]);
  Writeln;
  Writeln('Nombre: ', FileRec(fichero).Name);
  Close(fichero);
END.
```

11.11 LOS FICHEROS COMO TIPOS ABSTRACTOS DE DATOS

A continuación se presenta un ejemplo que implementa una Unit englobando las estructuras y los subprogramas necesarios para realizar determinadas operaciones con ficheros.

LOS FICHEROS COMO TIPOS ABSTRACTOS DE DATOS

Ejemplo 11.11

Construir una *unit* llamada *tadFich* que permita realizar operaciones básicas sobre un fichero que almacena datos académicos de alumnos de un determinado curso. La definición de estas operaciones se muestra a continuación.

```
PROCEDURE Inicializar (VAR F:fichero);
PROCEDURE LeerRegistro (VAR R:registro);
PROCEDURE AñadirRegistro (VAR F:fichero);
PROCEDURE ModificarRegistro (VAR F:fichero; C:clave);
PROCEDURE ListarRegistros (VAR F:fichero);
```

Como se puede observar, se va a realizar una segunda versión del ejemplo 11.4 operando con acceso directo sobre el fichero.

Codificación en Pascal

```
Unit tadFich;
(* Permite realizar operaciones básicas sobre un fichero que contiene
   fichas de alumnos *)

INTERFACE
Uses crt;
TYPE
  persona = RECORD
    nombre:string [80];
    dni:string[10];
    practicas:boolean;
    nota:ARRAY [1..35] OF real;
  END;
  Archivo = FILE OF persona;
  cadena = string[10];

PROCEDURE Inicializar_fichero (VAR f:Archivo);
PROCEDURE Leer_fichas (VAR alumno:persona);
PROCEDURE Anadir_fichas (VAR f:Archivo);
PROCEDURE Modificar_fichas (VAR f:Archivo; VAR clave:cadena);
PROCEDURE Listar_fichas (VAR f:Archivo);

IMPLEMENTATION
PROCEDURE Inicializar_fichero (VAR f:Archivo);
VAR
  alumno:persona;
  car:char;
BEGIN
  clrscr;
  Rewrite(f);
  Writeln ('¿ Desea introducir fichas ? ');
  Readln (car);
  WHILE car IN ['s','S'] DO
  BEGIN
    Leer_fichas(alumno);
    Write(f,alumno);
    Writeln;
    Writeln (' ¿Más fichas ? ');
    Readln(car);
  END;
  Close(f);
END;
{-----}
```

FICHEROS

```

PROCEDURE Leer_fichas (VAR alumno:persona);
VAR
    respu:char;
    i:integer;

BEGIN
    clrscr;
    WITH alumno DO
        BEGIN
            Writeln (' Nombre del alumno: ');
            Readln(nombre);
            Writeln (' DNI del alumno: ');
            Readln (dni);
            Writeln (' Entregó las prácticas s/n ');
            Readln(respu);
            IF (respu = 's') OR (respu = 'S')
                THEN practicas := true
                ELSE practicas := false;
            i:= 0;
            Writeln(' Introduzca las notas: -1 para finalizar ');
            Writeln;
            REPEAT
                i := i+1;
                Readln(nota[i]);
            UNTIL nota[i] = -1;
        END;
    END;

{-----}

PROCEDURE Anadir_fichas (VAR f:Archivo);
VAR
    alumno:persona;
    pos:longint;

BEGIN
    Reset(f);
    Leer_fichas(alumno);
    pos := Filesize(f);
    Seek(f,pos);
    Write(f,alumno);
    Close(f);
END;

{-----}

PROCEDURE Modificar_fichas (VAR f:Archivo; VAR clave:cadena);
VAR
    encontrado:boolean;
    aux:persona;
    contreg:longint; (* contador de registros *)

BEGIN
    clrscr;
    Reset(f);
    encontrado := false;
    contreg := 0;
    WHILE (NOT Eof(f)) AND (NOT encontrado) DO
        BEGIN
            Read(f,aux);
            IF aux.dni = clave
                THEN
                    BEGIN
                        encontrado := true;
                        Leer_fichas(aux);
                        Seek(f,contreg);
                        Write(f,aux);
                    END;
        END;
    END;

```

LOS FICHEROS COMO TIPOS ABSTRACTOS DE DATOS

```
        END;
        contreg := contreg+1;
    END;
    IF encontrado = false
    THEN
        BEGIN
            Writeln (' La ficha de clave ', clave , ' no existe ');
            Readln;
        END;
    Close(f);
END;

{-----}

PROCEDURE Listar_fichas (VAR f:Archivo);
VAR
    aux:persona;
    i:integer;

BEGIN
    clrscr;
    Reset(f);
    WHILE NOT Eof(f) DO
        BEGIN
            Read(f,aux);
            WITH aux DO
                BEGIN
                    Writeln (' Nombre del alumno: ',nombre);
                    Writeln (' DNI: ',dni);
                    IF practicas
                        THEN Writeln (' SI entregó prácticas ')
                        ELSE Writeln (' NO entregó prácticas ');
                    i := 1;
                    WHILE nota[i] <> -1 DO
                        BEGIN
                            Writeln (' nota ', i:2, ' = ', nota[i]:2:1);
                            i:=i+1;
                        END;
                    END;
                    Writeln;
                END;
            Readln;
        END;
    END.
END.
```

Codificación en Pascal del programa que utiliza la *unit* tadFich

```
PROGRAM Fichas (Input,Output,alumnos);
(* Segunda versión del ejemplo 11.4 *)

Uses crt,tadfich;
VAR
    alumnos:archivo;
    opcion:char;
    nombre:string[20];
    flag:boolean;
    clave:cadena;

BEGIN
    clrscr;
    Writeln (' Introduzca el nombre del fichero ');
    Readln (nombre);
    Assign (alumnos,nombre);
    flag := true;
    REPEAT
        clrscr;
```

```

REPEAT
  Writeln ( '*****' );
  Writeln ( '* * * * *');
  Writeln ( '*           MANEJO DE FICHAS DE ALUMNOS           *');
  Writeln ( '* * * * *');
  Writeln ( '*****' );
  Writeln ( '      Introduzca la opción que see:      ');
  Writeln;
  Writeln ( '      a) Inicializar                          ');
  Writeln;
  Writeln ( '      b) Listar fichas                             ');
  Writeln;
  Writeln ( '      c) Añadir fichas                            ');
  Writeln;
  Writeln ( '      d) Modificar fichas                         ');
  Writeln;
  Writeln ( '      e) Fin                                       ');
  Writeln;
  Writeln ( '      ¿ Que opción desea ?      ');
  Readln (opcion);
UNTIL opcion IN ['a','b','c','d','e','A','B','C','D','E'];
CASE opcion OF
  'a','A': Inicializar_fichero(alumnos);
  'b','B': Listar_fichas(alumnos);
  'c','C': Anadir_fichas(alumnos);
  'd','D': BEGIN
    clrscr;
    Writeln ( ' Introduzca D.N.I. del alumno  ');
    Readln (clave);
    Modificar_fichas(alumnos,clave);
  END;
  'e','E': flag := false
END;
UNTIL flag = false
END.

```

11.12 FICHEROS HOMOGENEOS Y NO HOMOGENEOS

El lenguaje Pascal estándar obliga a que todos los componentes de un fichero sean del mismo tipo, y así se declara explícitamente *FILE OF <tipo>*. A esta clase de ficheros se le denomina *ficheros homogéneos*. Los ficheros de texto también son ficheros homogéneos, dado que sus componentes son todas del tipo *char*. Sin embargo otros lenguajes de programación, o programas informáticos crean ficheros cuyos componentes no son todos del mismo tipo. A este clase de ficheros se les denomina *ficheros heterogéneos*.

El compilador Turbo Pascal permite el tratamiento de ficheros heterogéneos binarios, por medio de los ficheros sin tipo, y declarando el tamaño de registro de 1 *byte*. Es decir los trata como si fuesen homogéneos, manejándolos byte a byte. El ejemplo 11.9 mostró como copiar cualquier tipo de fichero, también se podría haber tratado cualquier tipo de información, con tal de saber su estructura interna en el fichero a nivel de bytes.

11.13 MANEJO DE FICHEROS EN REDES

La instalación de una red permite compartir datos y recursos entre grupos de trabajo. En la figura 11.7 se muestra una red de área local (*Local Area Network, LAN*) con topología en bus.

MANEJO DE FICHEROS EN REDES

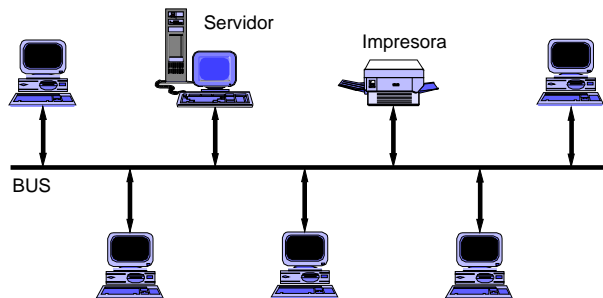


Figura 11.7 Esquema de una red en bus

Existen otras topologías como son la red local en estrella, en árbol, en anillo y en anillo-estrella. La más común es la que se muestra en la figura anterior donde todas las estaciones de trabajo se conectan a un bus bidireccional y en la que existe una estación principal o servidor. Posee grandes ventajas como por ejemplo modularidad y entre las desventajas cabe citar el fallo del bus que puede afectar a varias estaciones.

Desde la introducción de la versión 3.0 del MS-DOS y de la NetBIOS se ha generalizado el uso de redes para compartir periféricos, ficheros, impresoras, así como para realizar aplicaciones multiusuario con ordenadores personales PC.

El acceso de los programas desde varias estaciones a un mismo fichero plantea el problema *¿Qué ocurre si dos programas deciden escribir simultáneamente en el mismo fichero?* Si escriben en el mismo fichero, pero no en la misma zona no pasa nada. Pero si pretenden escribir simultáneamente en el mismo fichero y en la misma zona del fichero pueden surgir problemas. El sistema operativo MS-DOS a partir de su versión 3.0, incluye el comando *share.exe*, que cuando se ejecuta establece en memoria una *tabla de bloqueos (lock table)*. Esta tabla de bloqueos almacena la información sobre los ficheros y zonas de ficheros bloqueadas usando las funciones estándar de manejo de ficheros del DOS. El compilador Turbo Pascal usa las funciones estándar del DOS cuando llama a los procedimientos estándar *Reset* y *Rewrite*, pero no incluye mandatos que permitan bloquear un fichero o zonas de un fichero que se abre por un programa. Es necesario utilizar el procedimiento *MsDos*, de la *unit DOS*, para llamar a la función *\$5C* del DOS.

La llamada a la función *\$5C* del DOS permite a un programa bloquear una zona de un fichero. Sólo se bloquea la zona del fichero a modificar por el programa que accede al fichero. El resto de los programas pueden trabajar sobre las otras zonas del fichero no bloqueadas. Si otro programa necesita acceder a la zona bloqueada deberá esperar a que dicha zona quede desbloqueada.

La unidad de bloqueo más común es el registro, dado que los ficheros en Pascal contienen datos agrupados en registros. Se escribe una función para bloquear y desbloquear registros llamando a la función *\$5C* del DOS:

FICHEROS

```
USES Dos;

FUNCTION BloqueaRegistro (Operacion:integer;
                        Pfichero:pointer;
                        Nregistro:longInt): integer;

(* Recibe:
   - Operacion: 0 bloqueo y 1 desbloqueo
   - Pfichero: puntero a variable de tipo fichero
   - Nregistro: número del registro a bloquear, comenzando
               a contar por el cero. *)

(* Devuelve 0 si la operación de bloqueo/desbloqueo fue correcta,
   en caso contrario devuelve un código de error del DOS:
   1: Código de función no válido
   6: Handle de fichero no válido
  33: Registro bloqueado/desbloqueado en la actualidad
  36: Desbordado el buffer del comando share *)

VAR
  regs:Registers;
  PrimerByte, TotalBytes:LongInt;
BEGIN
  TotalBytes:=FileRec(Pfichero^).RecSize; (* Tamaño del registro *)
  PrimerByte:=Nregistro*TotalBytes;
  WITH regs DO
    BEGIN
      AH:=$5C;
      AL:=Operacion; (* 0 bloquea y 1 desbloquea *)
      BX:=FileRec(Pfichero^).Handle;
      DX:=PrimerByte;
      CX:=PrimerByte SHR 16;
      DI:=TotalBytes;
      SI:=TotalBytes SHR 16;
    END;
  MsDos(regs);
  BloqueaRegistro:=0;
  IF (Regs.flags AND 1=1) THEN BloqueaRegistro:=Regs.AX;
END;
```

La función *BloqueaRegistro* es válida tanto para ficheros con tipo o sin tipo, sin embargo no es válida para ficheros de texto, dado que su representación interna se define por el registro *FileRec* (y no *TextRec*). El tipo *pointer* (puntero sin tipo) se estudia en el capítulo 12. El operador *SHR* realiza un desplazamiento a la derecha de 16 bits, se utiliza para acceder al word bajo y al word alto de un tipo *longInt*. La última línea de la función *BloqueaRegistro* comprueba si el *flag* de acarreo (*carry*) en el registro de *flags* es 1.

El compartir datos y recursos lleva consigo tener que bloquear tanto ficheros como registros de ficheros para que la información almacenada no sea incongruente, por lo tanto antes de acceder a un registro es necesario bloquearlo, después se realiza el acceso, y por último se desbloquea.

Cuando se va a bloquear un registro puede darse el caso de que este ocupado, el programa debe de reintentarlo, hasta que dicho registro quede libre. Cuando el programa accede a un registro bloqueado, debe de manejarse rápidamente y desbloquearse, para entorpecer lo menos posible a otros posibles accesos (véase ejemplo 11.12).

También es posible acceder a la información de red del protocolo *NetBIOS* por medio del uso de interrupciones.

MANEJO DE FICHEROS EN REDES

Ejemplo 11.12

Para probar este ejemplo, debe cargarse previamente SHARE.EXE desde el sistema operativo. En primer lugar se crea la *unit redes* para el manejo de ficheros en red.

```
Unit Redes;

INTERFACE

PROCEDURE Desbloquea (Pfichero:pointer; Nregistro:longInt);
PROCEDURE Bloquea (Pfichero:pointer; Nregistro:longInt);

IMPLEMENTATION

USES Dos;

FUNCTION BloqueaRegistro (Operacion:integer;
                          Pfichero:pointer;
                          Nregistro:longInt): integer;

(* Recibe:
   - Operacion: 0 bloqueo y 1 desbloqueo
   - Pfichero: puntero a variable de tipo fichero
   - Nregistro: número del registro a bloquear, comenzando
                 a contar por el cero. *)
(* Devuelve 0 si la operación de bloqueo/desbloqueo fue correcta,
   en caso contrario devuelve un código de error del DOS:
   1: Código de función no válido
   6: Handle de fichero no válido
  33: Registro bloqueado/desbloqueado en la actualidad
  36: Desbordado el buffer del comando share *)

VAR
  regs:Registers;
  PrimerByte, TotalBytes:LongInt;
BEGIN
  TotalBytes:=FileRec(Pfichero^).RecSize; (* Tamaño del registro *)
  PrimerByte:=Nregistro*TotalBytes;
  WITH regs DO
    BEGIN
      AH:=$5C;
      AL:=Operacion; (* 0 bloquea y 1 desbloquea *)
      BX:=FileRec(Pfichero^).Handle;
      DX:=PrimerByte;
      CX:=PrimerByte SHR 16;
      DI:=TotalBytes;
      SI:=TotalBytes SHR 16;
    END;
  MsDos(regs);
  BloqueaRegistro:=0;
  IF (Regs.flags AND 1=1) THEN BloqueaRegistro:=Regs.AX;
END;

PROCEDURE ErrorBloqueo (CodError:integer; Nregistro:longInt);
BEGIN
  IF CodError=0
  THEN Writeln('Operación de bloqueo/desbloqueo realizada con éxito')
  ELSE
    BEGIN
      Write ('Error bloqueando el registro nº ', Nregistro, ' - ');
      CASE CodError OF
        1: Writeln ('Código de función del DOS no válido ');
        6: Writeln ('Handle no válido ');
        33: Writeln ('Registro actualmente bloqueado/desbloqueado');
      END;
    END;
END;
```

FICHEROS

```
END;
END;
END;

PROCEDURE Desbloquea (Pfichero:pointer; Nregistro:longInt);
VAR
    exitoDesbloqueo:integer;
BEGIN
    REPEAT
        exitoDesbloqueo:=BloqueaRegistro(1, Pfichero, Nregistro);
        ErrorBloqueo(exitoDesbloqueo, Nregistro);
    UNTIL exitoDesbloqueo=0;
END;

PROCEDURE Bloquea (Pfichero:pointer; Nregistro:longInt);
VAR
    exitoBloqueo:integer;
BEGIN
    REPEAT
        exitoBloqueo:=BloqueaRegistro(0, Pfichero, Nregistro);
        ErrorBloqueo(exitoBloqueo, Nregistro);
    UNTIL exitoBloqueo=0;
END;

END.
```

Utilización de la *unit redes* en la *unit fichas* de manejo de fichas de alumnos del ejemplo 11.11.

```
Unit fichas;
(* Permite realizar operaciones básicas sobre un fichero que contiene
   fichas de alumnos, con redes *)
INTERFACE
Uses crt, redes;
CONST
    ModoFicheroRed=66;
TYPE
    persona = RECORD
        nombre:string [80];
        dni:string[10];
        practicas:boolean;
        nota:ARRAY [1..35] OF real;
    END;
    Archivo = FILE OF persona;
    cadena = string[10];
PROCEDURE Inicializar_fichero (VAR f:Archivo);
PROCEDURE Leer_fichas (VAR alumno:persona);
PROCEDURE Anadir_fichas (VAR f:Archivo);
PROCEDURE Modificar_fichas (VAR f:Archivo; VAR clave:cadena);
PROCEDURE Listar_fichas (VAR f:Archivo);
IMPLEMENTATION
PROCEDURE Inicializar_fichero (VAR f:Archivo);
VAR
    alumno:persona;
    car:char;
    cont:integer;
BEGIN
    clrscr;
    fileMode:=64;
    Rewrite(f);
    Write ('¿ Desea introducir fichas ? (s/n) ');
    Readln(car);
    cont:=-1;
    WHILE car IN ['s', 'S'] DO
        BEGIN
            Leer_fichas(alumno);
```

MANEJO DE FICHEROS EN REDES

```

        cont:=cont+1;
        bloquea(@f,cont);
        Write(f,alumno);
        desbloquea(@f,cont);
        Writeln;
        Write(' ¿ Más fichas ? (s/n) ');
        Readln(car);
    END;
    Close(f);
END;
{-----}
PROCEDURE Leer_fichas (VAR alumno:persona);
VAR
    respu:char;
    i:integer;
BEGIN
    clrscr;
    WITH alumno DO
        BEGIN
            Write ( ' Nombre del alumno: ');
            Readln(nombre);
            Write ( ' DNI del alumno: ');
            Readln (dni);
            Write ( ' Entregó las prácticas s/n ');
            Readln(respu);
            IF (respu = 's') OR (respu = 'S')
                THEN practicas := true
                ELSE practicas := false;
            i:= 0;
            Writeln(' Introduzca las notas: -1 para finalizar ');
            REPEAT
                i := i+1;
                Readln(nota[i]);
            UNTIL nota[i] = -1;
        END;
    END;
{-----}
PROCEDURE Anadir_fichas (VAR f:Archivo);
VAR
    alumno:persona;
    pos:longint;
BEGIN
    FileMode:=ModoFicheroRed;
    Reset(f);
    Leer_fichas(alumno);
    pos := Filesize(f);
    bloquea(@f,pos);
    Seek(f,pos);
    Write(f,alumno);
    desbloquea(@f,pos);
    Close(f);
END;
{-----}
PROCEDURE Modificar_fichas (VAR f:Archivo; VAR clave:cadena);
VAR
    encontrado:boolean;
    aux:persona;
    contreg:longint; (* contador de registros *)
BEGIN
    clrscr;
    fileMode:=ModoFicheroRed;
    Reset(f);
    encontrado := false;
    contReg :=-1;
    WHILE (NOT Eof(f)) AND (NOT encontrado) DO
        BEGIN
            Read(f,aux);

```

FICHEROS

```

contReg := contReg +1;
IF aux.dni = clave
  THEN
  BEGIN
    encontrado := true;
    Leer_fichas(aux);
    bloquea(@f,contReg);
    Seek(f,contReg);
    Write(f,aux);
    desbloquea(@f,contReg);
  END;
END;
IF encontrado = false
  THEN
  BEGIN
    Writeln (' La ficha de clave ', clave , ' no existe ');
    Readln;
  END;
Close(f);
END;
{-----}
PROCEDURE Listar_fichas (VAR f:Archivo);
VAR
  aux:persona;
  i:integer;
BEGIN
  clrscr;
  FileMode:=ModoFicheroRed;
  Reset(f);
  WHILE NOT Eof(f) DO
  BEGIN
    Read(f,aux);
    WITH aux DO
    BEGIN
      Writeln (' Nombre del alumno: ',nombre);
      Writeln (' DNI: ',dni);
      IF practicas
      THEN Writeln (' SI entregó prácticas ')
      ELSE Writeln (' NO entregó prácticas ');
      i := 1;
      WHILE nota[i] <> -1 DO
      BEGIN
        Writeln (' nota ', i:2, ' = ', nota[i]:2:1);
        i:=i+1;
      END;
    END;
    Writeln;
  END;
  Readln;
END;
END.

```

Por último el programa principal, que maneja la aplicación completa:

```

PROGRAM Fichas_redes(Input,Output,alumnos);
Uses crt,fichas;
VAR
  alumnos:archivo;
  opcion:char;
  nombre:string[20];
  flag:boolean;
  clave:cadena;

```

EJECICIOS RESUELTOS

```
BEGIN
  clrscr;
  fileMode:=ModoFicheroRed;
  Write (' Introduzca el nombre del fichero de datos: ');
  Readln (nombre);
  Assign (alumnos,nombre);
  flag := true;
  REPEAT
    clrscr;
    REPEAT
      Writeln ('*****');
      Writeln ('*');
      Writeln ('*          MANEJO DE FICHAS DE ALUMNOS          *');
      Writeln ('*');
      Writeln ('*****');
      Writeln (' Introduzca la opción que desee: ');
      Writeln;
      Writeln (' a) Inicializar ');
      Writeln;
      Writeln (' b) Listar fichas ');
      Writeln;
      Writeln (' c) Añadir fichas ');
      Writeln;
      Writeln (' d) Modificar fichas ');
      Writeln;
      Writeln (' e) Fin ');
      Writeln;
      Write (' ¿ Que opción desea ? ');
      Readln (opcion);
    UNTIL opcion IN ['a','b','c','d','e','A','B','C','D','E'];
    CASE opcion OF
      'a','A': Inicializar_fichero(alumnos);
      'b','B': Listar_fichas(alumnos);
      'c','C': Anadir_fichas(alumnos);
      'd','D': BEGIN
        clrscr;
        Write (' Introduzca D.N.I. del alumno ');
        Readln (clave);
        Modificar_fichas(alumnos,clave);
      END;
      'e','E': flag := false
    END;
  UNTIL flag = false
END.
```

11.14 EJECICIOS RESUELTOS

11.1 Crear un programa que cuente el número de caracteres distintos del caracter blanco ' ' y el número de caracteres igual a ' ' de un fichero de texto. El nombre y el directorio en el que se encuentra se leerán por teclado.

Solución

```
PROGRAM CuentaBlancos (FicheroTexto,output);
CONST
  blanco = ' ';
VAR
  FicheroTexto: text; (* Tipo predefinido *)
  caracter: char;
  nb,nnb: integer;
  nombreFichero: string[80];
```

FICHEROS

```
BEGIN
  Writeln ( ' Introduzca el nombre del fichero de texto ');
  Readln (nombreFichero);
  Assign (FicheroTexto, nombreFichero);
  (* Esta sentencia solamente es necesaria en TURBO PASCAL *)
  nb:= 0;
  nnb:= 0;
  Reset (FicheroTexto);
  WHILE NOT Eof (FicheroTexto) DO
  BEGIN
    WHILE NOT Eoln (FicheroTexto) DO
    BEGIN
      Read (FicheroTexto,caracter);
      IF caracter = blanco
      THEN nb:= nb + 1
      ELSE nnb:= nnb + 1;
    END;
    Readln (FicheroTexto) (* Salta el final de línea *)
  END;
  Writeln ('Número de blancos:',nb);
  Writeln ('Número de no blancos:',nnb)
END.
```

- 11.2** Hacer un programa que cuente el número total de líneas; número total de caracteres y número de líneas que comiencen por la letra **A**, de un fichero de texto llamado *informe*. Se supondrá que este fichero se encuentra en el directorio A:\Datos. El resultado se escribirá por el fichero de salida *output*.

Solución

```
PROGRAM Contar (output, informe);
VAR
  informe:text; (* fichero de texto *)
  ch :char; (* variable para leer del fichero *)
  contChar, (* contador de caracteres *)
  contLin, (* contador de líneas *)
  contLinA :integer; (* contador de líneas comiencen por "A" *)
  primerCar :boolean; (* se pone a true cada vez que vayamos a *)
  (* procesar el primer carácter de cada *)
  (* línea *)
BEGIN
  contChar:=0;
  contLin:=0;
  contLinA:=0;
  Assign (informe,'a:\Datos\informe.dat');
  Reset (informe); (* Abrimos el fichero para lectura *)
  WHILE NOT Eof (informe) DO
  BEGIN
    primerCar:=true; (* Comienza una nueva línea *)
    WHILE NOT Eoln (informe) DO
    BEGIN
      Read (informe,ch);
      IF primerCar
      THEN
      BEGIN
        primerCar:=false;
        IF ch='A'
        THEN contLinA:=contLinA+1;
      END;
      contChar:=contChar+1;
    END;
    Readln (informe); (* Saltamos a la línea siguiente *)
    contLin:=contLin+1;
  END;
```


EJECICIOS RESUELTOS

```
END;
Close (informe);
Writeln('+++RESULTADOS +++');
Writeln('Número total de líneas= ',contLin);
Writeln('Número total de caracteres= ',contChar);
Writeln('Líneas que comiencen por A ',contLinA);
END.
```

11.3 Crear un fichero de texto llamado *informe2* que contenga:

- Como primera línea, una cabecera que se introducirá por teclado (fichero *input*).
- A continuación un texto que se encuentra en un fichero ya existente llamado *informe1*.
- Finalmente un texto (que puede constar de varias líneas), introducido desde teclado.

Recuerde que *Eof* y *Eoln* sin parámetros, se refieren al fichero *input*.

Solución

```
PROGRAM Crear (input, output, informe1, informe2);
VAR
    informe1, informe2 :text;
    ch :char;
BEGIN
    Assign (informe1,'a:\inf1.dat');
    Assign (informe2,'a:\inf2.dat');
    Rewrite (informe2);
    Writeln ('INTRODUZCA LA CABECERA');
    WHILE NOT Eoln DO (* Eoln= Eoln(input) *)
    BEGIN
        Read (ch);
        Write (informe2, ch);
    END;
    Writeln (informe2);
    Readln;
    Reset (informe1);
    WHILE NOT Eof (informe1) DO (* Copiar informe1 en informe2 *)
    BEGIN
        WHILE NOT Eoln (informe1) DO
        BEGIN
            Read(informe1,ch);
            Write(informe2,ch);
        END;
        Writeln (informe2);
        Readln (informe1);
    END;
    Writeln('INTRODUZCA TEXTO FINAL');
    WHILE NOT Eof DO
    BEGIN
        WHILE NOT Eoln DO
        BEGIN
            Read(ch); (* Algunos autores aconsejan poner: Read (input,ch)*)
            Write (informe2,ch);
        END;
        Readln;
        Writeln(informe2);
    END;
    Close (informe1);
    Close (informe2);
END.
```

11.4 Escribir un programa que nos permita crear un fichero *fDatos* desde la entrada estándar *input*. Los componentes de *fDatos* tendrán la siguiente estructura:

```
clave : número entero
nombre: cadena de caracteres
```

si es un hombre:

- a) edad: entero
- b) un conjunto de datos del tipo t

si es una mujer:

- a) altura: número real
- b) peso : número real.

El tipo *t* indicado más arriba es el siguiente:

```
TYPE
  datos= (fumador, alto, bajo);
  t = SET OF datos;
```

Solución

```
PROGRAM Crear (input, output, fDatos);
TYPE
  datos=(fumador, alto, bajo);
  T=SET OF datos;
  sexos=(hombre, mujer);
  reg=RECORD
    clave:integer;
    nombre:string[20];
    CASE sexo:sexos OF
      hombre:(edad:integer; conjunto:T);
      mujer :(altura, peso:real);
    END; (*de RECORD *)
VAR
  fDatos:FILE OF reg;
  r : reg;
PROCEDURE LeerRegistro (VAR r:reg);
  (* Lee los campos del registro r desde teclado *)
VAR
  ch: char; (* Para leer los datos de tipo enumerado a través de
             una variable carácter; ya que no se pueden leer
             directamente *)
{-----}
PROCEDURE LeerConjunto (VAR c:T);
  (* Se utilizará en la procedure LeerRegistro para *)
  (* Rellenar (leer) el campo de tipo conjunto *)
VAR
  ch: char;
BEGIN
  c:=[]; (*Inicialmente, conjunto vacío *)
REPEAT
  Writeln(' Teclee:F=Fumador;A=Alto;B=bajo;T=Terminar');
  Readln(ch);
CASE ch OF
  'A': c:=c+[alto];
  'B': c:=c+[bajo];
  'F': c:=c+[fumador];
  'T': ;
```

EJECICIOS RESUELTOS

```
END;
UNTIL ch= 'T';
END; (* LeerConjunto *)

{***** Cominezo de LeerRegistro *****}

BEGIN
  Writeln('¿Clave?');
  Readln(r.clave);
  IF r.clave <> 0
  THEN (* Si la clave es 0, no leemos más datos *)
    BEGIN
      Writeln ('¿Nombre?');
      Readln (r.nombre);
      Writeln('Teclee H para hombre; otro carácter para mujer ');
      Readln(ch);
      IF ch IN ['H','h']
      THEN
        BEGIN
          r.sexo:=hombre; (* ; Atención !, el tag-field es un
                           campo más del registro, y hay que darle
                           valor.*)
          Writeln('¿Edad?');
          Readln(r.edad);
          LeerConjunto(r.conjunto);
        END
      ELSE
        BEGIN (* es mujer*)
          r.sexo:=mujer;
          Writeln('¿Altura, Peso?');
          Readln(r.altura,r.peso);
        END;
      END; (* de IF r.clave...*)
  END; (* LeerRegistro*)

{***** Programa Principal *****}

BEGIN
  Assign (fDatos,'a:\dat.dat');
  Rewrite(fDatos);
  REPEAT
  LeerRegistro (r);
  IF r.Clave <> 0
  THEN Write (fDatos,r);
  UNTIL r.clave=0;
  Close (fDatos);
END.
```

11.5 Escribir un programa que recorra el fichero *fDatos* creado en el ejercicio anterior y escriba:

- Porcentaje de mujeres que midan más de 1.70 metros.
- Porcentaje de hombres menores de 18 años que sean fumadores, respecto al total de hombres menores de 18 años.
- Porcentaje de personas cuya primera letra del nombre sea la 'a' o la 'b'.

Solución

FICHEROS

```
PROGRAM Procesar (output, fDatos);
TYPE
  datos=(fumador, alto, bajo);
  T=SET OF datos;
  sexos=(hombre, mujer);
  reg=RECORD
  clave:integer;
  nombre:string[20];
  CASE sexo:sexos OF
  hombre:(edad:integer; conjunto:T);
  mujer :(altura, peso:real)
  END; (* de RECORD *)
VAR
  fDatos : FILE OF reg ;
  r:reg;
  totMujeres, mujeres170,
  totPersonas, totHombres18,
  fumadores18, nombreAoB :integer;
BEGIN
  totMujeres:=0;
  totPersonas:=0;
  mujeres170:=0;
  totHombres18:=0;
  fumadores18:=0;
  nombreAoB:=0;
  Assign (fDatos, 'a:\dat.adat');
  Reset (fDatos);
  WHILE NOT Eof (fDatos) DO
  BEGIN
    Read (fDatos,r);
    totPersonas:=totPersonas+1;
    (*Se comprueba si el nombre empieza por A o B *)
    IF r. Nombre [1] IN ['A', 'B']
    THEN nombreAoB:=nombreAoB+1;
    CASE r. sexo OF
    mujer :BEGIN
      totMujeres:=totMujeres+1;
      IF r. altura > 1.70
      THEN mujeres170:=mujeres170+1;
      END;
    hombre:IF r. edad < 18
      THEN
        BEGIN
          totHombres18:=totHombres18+1;
          IF fumador IN r. conjunto
          THEN fumadores18:=fumadores18+1;
          END
        END;
    END; (* CASE *)
  END; (* WHILE *)
  Close (fDatos);
END. (*Programa*)
```

11.6 Se tiene un *fichero1* de carnets de identidad perdidos, cuyos componentes contienen la siguiente información:

```
RECORD
  dni :longInt;
  nombre :string[20];
  domicilio:string[80];
END;
```

Asimismo existe un *fichero2* de carnets encontrados, cuyos componentes son de la forma:

EJECICIOS RESUELTOS

```
RECORD
  dni   :longInt;
  personaEnc:string[20]; { persona que lo encontró }
  tfno  :integer;
END;
```

Escribir un programa que liste por impresora (output) los siguientes datos:

Nº DNI Nombre Nombre de la persona que lo encontró Tfno.

y que además cree un *fichero3* con la siguiente estructura de sus componentes:

```
RECORD
  dni   :longInt;
  nombre :string[20];
  personaEnc:string[80];
  tfno  :longInt;
END;
```

Observación: los ficheros no están ordenados.

Solución

```
PROGRAM Buscar (fichero1, fichero2, fichero3, output);
TYPE
  c1 = RECORD
    dni   :longInt;
    nombre :string[20];
    domicilio:string[80];
  END;
  c2 = RECORD
    dni   :longInt;
    personaEnc :string[20];
    tfno  :longInt;
  END;
  c3 = RECORD
    dni   :longInt;
    nombre :string[20];
    personaEnc:string[80];
    tfno  :longInt;
  END;
VAR
  fichero1:FILE OF c1;
  fichero2:FILE OF c2;
  fichero3:FILE OF c3;
  r1:c1;
  r2:c2;
  r3:c3;
  encontrado:boolean; (*se pone a true al encontrar un dni en el
                        fichero2 que coincida con el dni del fichero1
                        en estudio *)
BEGIN
  Writeln('D.N.I.   NOMBRE PERSONA QUE LO ENCONTRO   TELEFONO');
  Writeln('');
  Assign (fichero1, 'a:\d1.dat');
  Assign (fichero3, 'a:\d3.dat');
  Assign (fichero2, 'a:\d2.dat');
  Reset (fichero1);
  Rewrite (fichero3);
  WHILE NOT Eof (fichero1) DO
  BEGIN
    Read (fichero1,r1);
    Reset (fichero2);
    encontrado:=false;
    WHILE (NOT Eof(fichero2)) AND (NOT encontrado) DO
```

FICHEROS

```
BEGIN
  Read (fichero2,r2);
  IF r1.dni=r2.dni
  THEN
  BEGIN
    encontrado:=true;
    r3.dni:=r1.dni;
    r3.nombre:=r1.nombre;
    r3.personaEnc:=r2.personaEnc;
    r3.tfno:=r2.tfno;
    Write (fichero3,r3);
    Writeln (r3.dni,r3.nombre,r3.personaEnc,r3.tfno);
  END;
END;
Close (fichero1);
Close (fichero2);
Close (fichero3);
END.
```

11.7 Escribir un programa que halle la intersección de las letras de cada línea de un texto contenido en un fichero.

Solución

```
PROGRAM Frases (letras,output);
  TYPE
    conjuntoLetras= SET OF char;
  VAR
    letras : text;
    letrasLinea:conjuntoLetras; (* las letras de cada línea *)
    letrasTexto:conjuntoLetras; (* intersección de las letras de cada
                                línea.*)
  (***** )

  PROCEDURE LeerConjunto (VAR elConjunto: conjuntoLetras);
  VAR
    car : char;
  BEGIN
    elConjunto:=[];
    WHILE NOT Eoln(letras) DO
    BEGIN
      Read(letras,car);
      elConjunto:=elConjunto+[car]
    END;
    Readln(letras);
  END;
  (***** )

  PROCEDURE EscribirConjunto (elConjunto: conjuntoLetras);
  VAR
    car:char;
  BEGIN
    IF elConjunto=[]
    THEN Write('Conjunto vacio')
    ELSE FOR car:=Chr(33) TO Chr(126) DO
    IF car IN elConjunto
    THEN Write(car);
    Writeln;
  END;
  (***** Programa principal ***** )
```

EJECICIOS RESUELTOS

```
BEGIN
  Assign(letras,'LETRAS.DAT'); (* Será buscado en la unidad y
                               directorio activo *)

  Reset(letras);
  LeerConjunto(letrastexto);
  WHILE NOT Eof(letras) DO
    BEGIN
      LeerConjunto(letrasLinea);
      letrasTexto:=letrasTexto * letrasLinea;
    END;
  Close(letras);
  EscribirConjunto(letrasTexto);
  Write (' Pulse una tecla ');
  Readln;
END.
```

- 11.8** Crear un programa que procese un fichero de enteros creado a partir del contenido de una matriz $n \times m$ utilizando acceso directo.

Solución

```
PROGRAM Fad (input,output,f);
Uses crt;
CONST n=3;
      m=2;
TYPE
  matriz = ARRAY[1..n,1..m] OF integer;
  fich= FILE OF integer;
VAR
  f:fich;
  a:matriz;
  i:integer;
  st:string[10];

{-----}

PROCEDURE menu (VAR i:integer);
BEGIN
  Writeln(' 1.- Tratar fichero ');
  Writeln(' 2.- Lectura fichero ');
  Writeln(' 3.- Modificar fichero ');
  Writeln(' 4.- Fin ');
  Writeln (' introduzca opción');
  Readln(i)
END;

{-----}

PROCEDURE leermat (VAR a:matriz);
VAR
  i,j:integer;
BEGIN
  Writeln (' Introduzca 6 enteros para la matriz 3x2 ');
  FOR i:=1 TO n DO
    FOR j:=1 TO m DO
      Readln(a[i,j])
    END;
  END;

{-----}
```

FICHEROS

```
PROCEDURE leerfich (VAR f:fich);
(* visualiza el porcentaje de fichero que va leyendo *)
VAR
  i:integer;
  r:real;
  long,pos:longint;
BEGIN
  Clrscr;
  Reset(f);
  long:=filesize(f);
  WHILE NOT Eof(f) DO
  BEGIN
    Read(f,i);
    pos:=filepos(f);
    r:=(pos/long)*100;
    Writeln('leido el ',r:8:2, ' por ciento');
  END;
  Close(f);
  Readln;
END;

{-----}

PROCEDURE grabarfich (VAR f:fich; VAR a:matriz);
(* pasa los datos de la matriz al fichero *)
VAR
  i,j:integer;
BEGIN
  Rewrite(f);
  FOR i:= 1 TO n DO
  FOR j:= 1 TO m DO
  Write(f,a[i,j]);
  END;
END;

{-----}

PROCEDURE procesar (VAR f:fich);
(* visualiza el contenido del fichero por columnas *)
VAR pos,long:longint;
    i,j,k:integer;
BEGIN
  Clrscr;
  Reset(f);
  FOR i:=1 TO m do
  BEGIN
    pos:=i-1;
    FOR j:= 1 TO n DO
    BEGIN
      Seek(f,pos);
      Read(f,k);
      Writeln('posicion ', pos:3, ' contenido ',k:4);
      pos:=pos+2
    END;
  END;
  Readln;
END;

{-----}

PROCEDURE modificar (VAR f:fich);
(* permite modificar posiciones del fichero *)
VAR j:integer;
```


EJECICIOS RESUELTOS

```
BEGIN
  Clrscr; (* Limpia pantalla *)
  Reset(f);
  REPEAT
  Writeln('introduzca posición ');
  Readln(j);
  UNTIL (j>=0) AND (j<=filesize(f));
  Seek (f,j);
  Writeln(' introduzca valor ');
  Readln(j);
  Write (f,j);
  Close(f);
END;

{***** Programa principal *****}

BEGIN
  Clrscr;
  st:='a:\bb.dat'; (* El fichero bb.dat se creará en la unidad A *)
  Assign(f,st);
  leermat(a);
  grabarfich(f,a);
  REPEAT
  clrscr;
  menu(i);
  CASE i OF
  1: procesar(f);
  2: leerfich(f);
  3: modificar(f);
  4;
  END;
  UNTIL i=4;
END.
```

- 11.9** Dado un fichero de tipo *text*, que contiene un texto en castellano, realizar un programa que calcule la frecuencia de aparición de palabras de igual número de caracteres. No deben contarse los signos de puntuación (, ; : .). Las palabras pueden estar separadas por uno o más blancos. Ninguna palabra del fichero está cortada entre dos líneas. La salida del programa debe ser de la siguiente forma:

```
PALABRAS CON 1 LETRA ..... 117
PALABRAS CON 2 LETRAS..... 2376
...
PALABRAS CON 80 LETRAS..... 1
```

Se establece de antemano, que 80 es el número máximo de caracteres que puede tener una palabra o cadena.

Solución

```
PROGRAM CuentaFrecuenciaDeLongitudesDePalabras(texto,output);
CONST
  n=80; (* Longitud máxima de una palabra o cadena *)
VAR
  texto : text;
  car : char;
  signosPuntuacion: SET OF char;
  contador : ARRAY[1..n] OF integer;
  i, j : integer ;
  nombreFichero : string [12];
```

```

BEGIN
Write('Introduzca el nombre del fichero.....');
Readln(nombreFichero);
Assign(texto,nombreFichero);
Reset(texto);
FOR j:=1 TO n DO
  contador[j]:=0;
signosPuntuacion:= [' ','',',',';',':',''.'];
WHILE NOT Eof(texto) DO
BEGIN
  WHILE NOT Eoln(texto) DO
  BEGIN
    Read(texto,car);
    IF ([car] * signosPuntuacion[])
      THEN i:=1
      ELSE i:=0;
    WHILE ( ([car] * signosPuntuacion=[]) AND NOT Eoln(texto)) DO
    BEGIN
      Read(texto,car);
      IF ([car] * signosPuntuacion[])
        THEN i:=i+1;
      END;
      contador[i]:=contador[i]+1;
      IF i <> 0
        THEN Writeln('Contador',i,'=',contador[i]);
    END; (* WHILE NOT Eoln *)
    Readln(texto);
  END; (* WHILE NOT Eof *)
  Writeln('Palabras conletra.....',contador[1]);
  FOR j:=2 TO 80 DO
    Writeln('Palabras con ',j:2,' letras.....', contador[j]);
  END.

```

11.10 Escribir un programa que permita cifrar y descifrar ficheros de texto dependiendo del carácter de la primera posición del fichero, siguiendo el criterio que a continuación se especifica:

- El carácter 0 indica que el fichero está cifrado.
- El carácter 1 indica que el fichero está sin cifrar.

Para cifrar un texto se calculará tres veces el sucesor de cada caracter y para descifrarlo se usará el proceso contrario

Si el fichero está cifrado pasará a estar sin cifrar y al revés.

Solución

```

PROGRAM Cifrar(entrada,salida,output);
VAR
  car:char;
  encif:boolean;
  entrada,salida :text;
BEGIN
  Assign (entrada,'entrada.dat');
  Assign (salida,'salida.dat');
  Reset(entrada);
  Rewrite(salida);
  Readln(entrada,car);
  IF car='0'
    THEN (* El 0 indica que está cifrado *)
      BEGIN
        encif:=true;

```

EJECICIOS RESUELTOS

```
        Writeln(salida,'1'); (* El 1 indica que está sin cifrar *)
        Writeln('1')
      END
    ELSE
      BEGIN
        encif:=false;
        Writeln(salida,'0');
        Writeln('0')
      END;
  WHILE NOT Eof(entrada) DO
  BEGIN
    WHILE NOT Eoln(entrada) DO
    BEGIN
      Read(entrada,car);
      IF encif
        THEN car:=Succ(Succ(Succ(car)))
        ELSE car:=Pred(Pred(Pred(car)));
      Write(salida,car);
      Write(car)
    END;
    Writeln(salida); (* marca el final de línea *)
    Writeln;
    Readln(entrada)
  END;
  Close (salida)
END.
```

11.11 Escribir un programa que lea un fichero de texto e indique si contiene todas las vocales o no.

Solución

```
PROGRAM Vocales(texto,output);
VAR letra:char;          (* Caracter que se lee *)
    texto:text;         (* Fichero de texto *)
    conjunto :SET OF 'A'..'Z'; (* Conjunto de letras del texto *)
BEGIN
  (* Preparación del fichero de texto *)
  Assign(texto,'VOCAL.DAT');
  Reset(texto);

  (* Inicialización a vacío del conjunto de letras del texto *)
  conjunto:=[];

  (* Bucles de lectura del fichero y llenado del conjunto que contiene
  las letras del texto dado *)

  WHILE NOT Eof(texto) DO
  BEGIN

    (* lee línea a línea el fichero de texto *)

    WHILE NOT Eoln(texto) DO
    BEGIN
      Read(texto,letra);

      (* Conversión de las letras minúsculas a mayúsculas, pues sólo
      hemos definido un conjunto de letras mayúsculas *)

      IF (letra>='a') AND (letra<='z')
      THEN
        letra:=Chr(Ord(letra)-Ord('a')+Ord('A'));
    END;
  END;
END.
```

FICHEROS

```
(* Incorporación del caracter leído al conjunto mediante la unión *)
conjunto:=conjunto+[letra]
END;
Readln(texto)
END;

(* Si el conjunto de letras del texto incluye al subconjunto de las
vocales, entonces contiene todas las vocales *)

Writeln;
IF conjunto=['A','E','I','O','U']
THEN Writeln('Contiene todas las vocales')
ELSE Writeln('Le falta alguna vocal');
Writeln ('Pulse <Return> para volver al editor');
Readln;
END.
```

11.12 Dadas las siguientes declaraciones globales:

```
TYPE
asignaturas = (Estructura, Algebra, Metodologia, Fisica, Logica);
tipos = (parcial, junio, septiembre, febrero);
exámenes = RECORD
    asignatura: asignaturas;
    alumnos, npresentados, nprobados: 1..500;
    practicas: boolean;
    CASE tipo: tipos OF
        parcial: (compensable: boolean);
        junio, septiembre, febrero:();
    END;
ficheros = FILE OF exámenes;
```

escribir un programa en Pascal que, leyendo datos de un fichero del tipo *ficheros*, genere un fichero de texto con la siguiente información:

- Una cabecera .
- Una línea por cada elemento del fichero leído, en la que se reflejen los datos de dicho elemento.

Ejemplo del contenido del fichero de texto:

```
          E. U. I. T. DE INFORMATICA
          RESUMEN DE EXAMENES. CURSO 1º

Asignatura   Tipo   Prácticas  NºAlumnos  Nºpres.  Nº aprob.
Estructura   parcial   SI          400         250       150
Algebra      junio     NO          390         300       168
Física       septiembre SI          150         110        69
Metodol.    parcial   SI          410         298       213
Lógica      febrero  NO          10          10         7
```

EJECICIOS RESUELTOS

Solución

```
PROGRAM ProcesarFich(input,output,fich);
Uses crt;
TYPE
  asignaturas = (Estructura, Algebra, Metodologia, Fisica,
Logica);
  tipos = (parcial, junio, septiembre, febrero);
  examenes = RECORD
    asignatura: asignaturas;
    nalumnos, npresentados, naprobados: 1..500;
    practicas: boolean;
    CASE tipo: tipos OF
      parcial: (compensable: boolean);
      junio, septiembre, febrero:();
    END;
  ficheros = FILE OF examenes;
VAR
  examen: examenes;
  fich: ficheros;
  texto: text;
  ch: char;

(*****)

PROCEDURE CreaFichero(VAR fichero: ficheros);
  VAR
    n: integer;
    examen: examenes;
    ch:char;

{-----}

PROCEDURE Leeficha(VAR ficha: examenes);
VAR
  ch: char;
BEGIN
  WITH ficha DO
  BEGIN
  REPEAT
    ClrScr;
    Writeln;
    Writeln;
    Writeln;
    Writeln('E...Estructura');
    Writeln('A...Algebra');
    Writeln('F...Física');
    Writeln('M...Metodología');
    Writeln('L...Lógica');
    Write('Teclee la inicial de la asignatura correspondiente...');
    Readln(ch);
  UNTIL ch IN ['c','a','f','p','i','C','A','F','P','I'];
  CASE ch OF
    'e','E': asignatura := Estructura;
    'a','A': asignatura := Algebra;
    'f','F': asignatura := Fisica;
    'm','M': asignatura := Metodologia;
    'l','L': asignatura := Logica;
  END; (* CASE *)
  Write('¿Número alumnos?'); Readln(nalumnos);
  Write('¿Número alumnos presentados?'); Readln(npresentados);
  Write('¿Número alumnos aprobados?'); Readln(naprobados);
  Write('¿Prácticas (s/n)?'); Readln(ch);
  IF ch IN ['s','S']
  THEN practicas:= true
  ELSE practicas:= false;
```

FICHEROS

```

Writeln('TIPO DE EXAMEN:');
REPEAT
  Writeln('P=PARCIAL  J=JUNIO  S=SEP  F=FEB');
  Write('Elija tipo...'); Readln(ch);
UNTIL ch IN ['p','j','s','f','P','J','S','F'];
CASE ch OF
  'p','P': BEGIN
    tipo := parcial;
    Write('¿Compensable (s/n)?');
    Readln(ch);
    IF ch IN ['s','S']
      THEN compensable:=true
      ELSE compensable:=false;
    END;
  'j','J': tipo := junio;
  's','S': tipo := septiembre;
  'f','F': tipo := febrero;
END; (* case *)
END; (* WITH ficha *)
END; (* LeeFicha *)

{----- CrearFichero -----}

BEGIN
Rewrite(fichero); n:=0;
REPEAT
  n:=n+1;
  Writeln('DATOS DEL EXAMEN Número ',n);
  Leeficha(examen);
  Write(fichero, examen);
  Write('¿Más fichas (s/n)?');
  Readln(ch);
UNTIL NOT(ch IN ['s','S']);
Close(fichero);
END;

(*****

PROCEDURE CreaTexto(VAR fichero:ficheros; VAR texto:text);
VAR examen:examenes;
BEGIN
  Reset(fichero);
  Rewrite(texto);

(***** Cabecera del texto *****)

Writeln(texto,'          E. U. I. T. DE INFORMATICA ');
Writeln(texto,'          RESUMEN DE EXAMENES. CURSO 1º');
Writeln(texto);
Writeln(texto,'Asignatura Tipo Prácticas N°Alumnos N°pres.
N°aprob. ');
WHILE NOT eof(fichero) DO
  BEGIN
  Read(fichero, examen);
  WITH examen DO
  BEGIN
  CASE asignatura OF
    Estructura: Write(texto,' Estructura ');
    Algebra: Write(texto,' Algebra ');
    Fisica: Write(texto,' Física ');
    Metodologia: Write(texto,' Metodología ');
    Logica : Write(texto,' Lógica ');
  END; (*case *)
  CASE tipo OF
    parcial:IF compensable
      THEN Write(texto,' parcial comp. ')
      ELSE Write(texto,' parcial no comp. ');

```

EJECICIOS RESUELTOS

```
        junio: Write(texto,' junio ');
        septiembre: Write(texto,' septiembre ');
        febrero : Write(texto,' febrero ');
    END; (* CASE *);
    IF practicas
    THEN Write (texto,' SI ')
    ELSE Write (texto,' NO ');
    Writeln(texto,nalumnos:8, npresentados:11, naprobados:11);
    END; (* WITH examen *)
    END; (* WHILE *)
    Close(texto);
END;

(***** programa principal *****)

BEGIN
    Assign(fich, 'a:\fich.dat');
    Assign(texto, 'a:\fich.txt');
    Write('¿ Desea crear el fichero de datos (s/n)?');
    Readln(ch);
    IF ch IN ['s', 'S']
    THEN CreaFichero(fich);
    Createxto(fich,texto);
END.
```

- 11.13** Escribir un programa que permita conocer si el contenido de dos ficheros es idéntico. La estructura de sus registros es la siguiente:

```
Datos = RECORD
    nombre:PACKED ARRAY [1..20] OF char;
    equipo:PACKED ARRAY [1..3] OF char;
    tiempo:integer;
END;
```

Solución

```
PROGRAM FichIdenticos(input, output, f1, f2);
Uses crt;
TYPE
    tipoDatos= RECORD
        nombre:PACKED ARRAY [1..20] OF char;
        equipo:PACKED ARRAY [1..3] OF char;
        tiempo:integer;
    END;
    tipoFichero=FILE OF tipoDatos;
VAR
    f1, f2:tipoFichero;
    nom1, nom2: string[30];

(*****)

FUNCTION Identicos(VAR f1, f2: tipoFichero):boolean;
VAR
    dato1, dato2: tipoDatos;
    iguales:boolean;
BEGIN
    iguales:=true;
    Reset(f1);
    Reset(f2);
    WHILE (NOT Eof(f1)) AND (NOT Eof(f2)) AND iguales DO
    BEGIN
        Read(f1, dato1); Read(f2, dato2);
        IF NOT( (dato1.nombre = dato2.nombre) AND
            (dato1.equipo = dato2.equipo) AND
```

FICHEROS

```
        (dato1.tiempo = dato2.tiempo) )
    THEN iguales:=false;
END;
IF NOT Eof(f1) OR NOT Eof(f2)
    THEN iguales:=false;
    Identicos:=iguales;
END;

(***** Programa principal *****)

BEGIN
  ClrScr;
  Write('¿Nombre del fichero 1 ?');
  Readln(nom1);
  Write('¿Nombre del fichero 2 ?');
  Readln(nom2);
  Assign(f1,nom1); Assign(f2,nom2);
  IF Identicos(f1,f2)
    THEN Writeln('Ficheros idénticos')
    ELSE Writeln('Ficheros no idénticos');
  Writeln;
  Write('Pulse una tecla para continuar ');
  REPEAT UNTIL Keypressed;
END.
```

11.14 Se supone creado un fichero (no ordenado), cuyos elementos son del tipo:

```
TYPE alumnos = RECORD
  nombre:string[30];
  notaTeoria,notaPracticas,notaFinal:real;
END;
```

El valor -1 en un campo de notas significa no presentado (*notaFinal* es -1 si lo son *notaTeoria* o *notaPracticas*).

Se utilizarán los subprogramas cuyas llamadas se indican a continuación:

- *LeeVector(fich1,v,n)*: Almacena en el vector *v* el contenido del fichero *fich1*. Devuelve al punto de llamada el vector *v* y su número de elementos, *n*.
- *OrdNombre(v,n)*: Devuelve al punto de llamada el vector *v* ordenado alfabéticamente por el campo nombre.
- *OrdNotaF(v,n)*: Devuelve al punto de llamada el vector *v* ordenado por notas finales crecientes.
- *Npresentados(v,n)*: Función cuyo resultado es el número de alumnos de *v* cuya nota final es positiva.
- *Naprobados(v,n)*: Función cuyo resultado es el número de alumnos de *v* cuya nota final es igual o mayor que 5.0.
- *NotaMedia(v,n)*: Función cuyo resultado es la media entre las notas finales positivas de *v*.

Se pide:

EJECICIOS RESUELTOS

a) Escribir las cabeceras de los subprogramas anteriores (utilizando comunicación por dirección sólo cuando sea necesario), y las declaraciones globales necesarias para hacer dichas llamadas.

b) Escribir una función que calcule la mediana de las notas finales de los alumnos presentados.

c) Escribir un subprograma para ser llamado por la sentencia *CreaTexto(fich2,v,n)*; siendo *fich2* una variable global de tipo text, que tras la llamada deberá contener:

- Una cabecera.

- Un listado ordenado alfabéticamente de los datos de v

- Unas líneas finales indicando el número de presentados, de aprobados y de suspensos, la media y la mediana de las notas finales de los alumnos presentados.

Ejemplo del contenido del fichero *fich2*:

Alumno	NOTAS FINALES		
	Teoría	Prácticas	Final
Alvarez Fernández, Pablo	4.5	7.5	4.5
Antuña López, Javier	No pres.	8.5	No pres.
Busto Rodríguez, Ana	6.5	9.0	8.0
Carvajal Iglesias, J. Luis	7.0	No pres.	No pres.
Fanjul Sánchez, Josefina	No pres.	No pres.	No pres.
Fernández Fernández, Tomás	8.0	3.0	3.0
...			
Zapico Suárez, Ernesto	5.0	10.0	7.5
Número de alumnos presentados	48		
Número de alumnos aprobados	29		
Número de alumnos suspensos	19		
Media de las notas finales	5.7		
Mediana de las notas finales	6.0		

Solución

```
PROGRAM ProcesarFich(input, output, fich1, fich2);
Uses crt;
CONST
    aprobado=5.0;
    noP=-2.0;
TYPE
    alumnos = RECORD
        nombre:string[40];
        dni:string[10];
        notaT,notaP,notaF:real;
    END;
    fichero = FILE OF alumnos;
    vector = ARRAY[1..100] OF alumnos;
VAR
    fich1: fichero;
    fich2: text;
    v: vector;
    respu:char;
    n:integer;
    nomfich:string[12];
```

(*****)

FICHEROS

```

FUNCTION NotaFinal(nT,nP:real):real;
BEGIN
  IF ((nT>=aprobado)AND(nP>=aprobado))
    THEN NotaFinal:=(nT+nP)/2
    ELSE
      IF nT < aprobado
        THEN NotaFinal:=nT
        ELSE NotaFinal:=nP;
  IF (nT = noP) OR (nP = noP)
    THEN NotaFinal:=noP;
END;

(***** )

PROCEDURE CreaFichero(VAR fich:fichero);
VAR
  aux:alumnos;
  i:integer;
  respu:char;
BEGIN
  Rewrite(fich);
  i:=0;respu:='s';
  WHILE Upcase(respu)='S' DO
  BEGIN
    i:=i+1;
    WriteLn('DATOS DEL ALUMNO Número ',i,' : ');
    WITH aux DO
    BEGIN
      Write('¿Nombre? ');
      ReadLn(nombre);
      Write('¿dni? ');
      ReadLn(dni);
      Write('¿NT, Np? ');
      ReadLn(Notat,Notap);
    END;
    Write(fich,aux);
    Write('¿Más? (S/n) ');
    respu:=Readkey;
  END;
  Close(fich);
END;

(***** )

PROCEDURE LeeVector(VAR fich:fichero;VAR w:vector; VAR n:integer);
BEGIN
  Reset(fich);
  n:=0;
  WHILE NOT Eof(fich) DO
  BEGIN
    n:=n+1;
    Read(fich,v[n]);
  END;
END;

(***** )

PROCEDURE Intercambia(VAR reg1,reg2:alumnos);
VAR aux: alumnos;
BEGIN
  aux:=reg1;
  reg1:=reg2;
  reg2:=aux;
END;

(***** )

```

EJECICIOS RESUELTOS

```
PROCEDURE OrdNombre(VAR w:vector; n:integer);
VAR
    i,j:integer;
BEGIN
    FOR i:=2 TO n DO
        FOR j:=n DOWNTO i DO
            IF v[j].nombre < v[j-1].nombre
                THEN Intercambia(v[j], v[j-1]);
        END;
    END;

(***** )

PROCEDURE OrdNotaF(VAR w:vector; n:integer);
VAR i,j:integer;
BEGIN
    FOR i:=2 TO n DO
        FOR j:=n DOWNTO i DO
            IF v[j].notaF < v[j-1].notaF
                THEN Intercambia(v[j], v[j-1]);
        END;
    END;

(***** )

PROCEDURE ActualizaFichero(VAR fich:fichero;VAR w:vector; n:integer);
VAR
    i:integer;
BEGIN
    Rewrite(fich);
    FOR i:=1 TO n DO Write(fich,v[i]);
    Close(fich);
END;

(***** )

FUNCTION Max(w:vector;n:integer):real;
VAR i:integer;
    maximo:real;
BEGIN
    maximo:=0;
    FOR i:=1 TO n DO
        IF v[i].notaF > maximo
            THEN maximo:=v[i].notaF;
    END;
    Max:=maximo;
END;

(***** )

FUNCTION Min(w:vector;n:integer):real;
VAR
    i:integer;
    minimo:real;
BEGIN
    minimo:=10;
    FOR i:=1 TO n DO
        IF (v[i].notaF < minimo) AND (v[i].notaF<>-2)
            THEN minimo:=v[i].notaF;
    END;
    Min:=minimo;
END;

(***** )

FUNCTION Naprobados(w:vector;n:integer):integer;
VAR
    m,i:integer;
```

FICHEROS

```

BEGIN
  m:=0;
  FOR i:=1 TO n DO
    IF v[i].notaF>=aprobado
      THEN m:=m+1;
  Naprobados:=m;
END;

( ***** )

FUNCTION Npresentados(w:vector;n:integer):integer;
VAR
  i,p:integer;
BEGIN
  p:=0;
  FOR i:=1 TO n DO
    IF v[i].notaF<>noP
      THEN p:=p+1;
  Npresentados:=p;
END;

( ***** )

FUNCTION NotaMedia(w:vector;n:integer):real;
VAR
  suma:real;
  i:integer;
BEGIN
  suma:=0;
  FOR i:=1 TO n DO
    IF v[i].notaF<>noP
      THEN suma:=suma+v[i].notaF;
  NotaMedia:=suma/Npresentados(w,n);
END;

( ***** )

FUNCTION Mediana(w:vector;n:integer):real;
VAR
  medio,np,nnp:integer;
BEGIN
  OrdNotaF(w,n);
  np:=Npresentados(w,n);
  nnp:=n-np;
  IF Odd(np)
    THEN
      BEGIN
        medio:=(np+1) DIV 2;
        medio:= nnp+medio;
        Mediana := v[medio].notaF;
      END
    ELSE
      BEGIN
        medio:= np DIV 2;
        medio:=nnp+medio;
        Mediana := (v[medio].notaF+v[medio+1].notaF)/2;
      END;
END;

( ***** )

PROCEDURE Createxto(VAR fich:text; VAR w:vector; n:integer);
VAR
  i,j:integer;

```

EJECICIOS RESUELTOS

```

BEGIN
  Rewrite(fich);
  Writeln(fich,' ');
  Writeln(fich);
  Writeln(fich,'          NOTAS FINALES ');
  Writeln(fich);
  Writeln(fich,' ');
  Writeln(fich);
  Writeln(fich,' Alumno  Teoría  Prácticas  Final');
  OrdNombre(w,n);
  FOR i:=1 TO n DO
    WITH w[i] DO
      BEGIN
        Write(fich,nombre);
        FOR j:=length(nombre)+1 TO 32 DO
          Write(fich,' ');
        NotaF:=NotaFinal(NotaT,NotaP);
        IF NotaT = -2
          THEN Write(fich,' No pres. ')
          ELSE Write(fich,NotaT:5:1,' ':6);
        IF NotaP = -2
          THEN Write(fich,' No pres. ')
          ELSE Write(fich,NotaP:5:1,' ':6);
        IF NotaF = -2
          THEN Writeln(fich,' No pres. ')
          ELSE Writeln(fich,NotaF:5:1);
        END;
        Writeln(fich,'Número de presentados....',Npresentados(w,n));
        Writeln(fich,'Número de aprobados.....',Naprobados(w,n));
        Writeln(fich,'Número de suspensos .....',Npresentados(w,n) -
Naprobados(w,n));
        Writeln(fich,'Nota media.....',NotaMedia(w,n):4:1);
        Writeln(fich,'Nota mínima.....',Min(w,n):4:1);
        Writeln(fich,'Nota máxima.....',Max(w,n):4:1);
        OrdNotaF(w,n);
        Writeln(fich,'Mediana de las notas..',Mediana(w,n):4:1);
        Close(fich);
      END;
  END;

(***** Programa principal *****)

BEGIN
  Write('¿Nombre del fichero de entrada?');
  Readln(nomfich);
  IF nomfich=''
    THEN Assign(fich1,'alumnos.dat')
    ELSE Assign(fich1,nomfich);
  Write('¿Nombre del fichero de salida?');
  Readln(nomfich);
  IF nomfich=''
    THEN Assign(fich2,'texto.dat')
    ELSE Assign(fich2,nomfich);
  Write('¿Desea crear el fichero(s/n)?');
  Readln(respu);
  IF Uppcase(respu)='S'
    THEN CreaFichero(fich1);
  LeeVector(fich1,v,n);
  CreaTexto(fich2,v,n);
  CreaTexto(output,v,n);
  OrdNombre(v,n);
  ActualizaFichero(fich1,v,n);
  Readln;
END.

```

- 11.15** Teniendo en cuenta las siguientes declaraciones, escribir dos procedimientos llamados *CreaFichero* y *ListaFichero* que permitan crear y listar el contenido de un fichero respectivamente. La estructura de los registros viene dada a continuación.

```

TYPE estadoCivil = (soltero, casado, viudo, divorciado);
sex = (m,f);
info = RECORD
  nombre: String[30];
  edad: integer;
  sexo: sex;
END;
ficha = RECORD
  datos:info;
  estado: estadoCivil;
  CASE estadoCivil OF
    casado: (conyuge:info);
    soltero, viudo, divorciado: ();
  END;
fichero = FILE OF ficha;
VAR fich: fichero;

```

Solución

```

PROCEDURE CreaFichero(VAR fich:fichero);
VAR
  aux:ficha;
  aux1:char;
BEGIN
  Assign(fich, 'fich1.dat');
  Rewrite(fich);
  REPEAT
  WITH aux DO
  BEGIN
    Write('¿Nombre.....?'); Readln(datos.nombre);
    Write('¿Edad.....?'); Readln(datos.edad);
    Write('¿sexo (M/F).....?'); Readln(aux1);
    IF Uppcase(aux1)='M'
    THEN datos.sexo:=m
    ELSE datos.sexo:=f;
    Write('¿Estado civil (s,c,v,d)...?');Readln(aux1);
    CASE aux1 OF
      's','S': estado := soltero;
      'c','C': estado := casado;
      'v','V': estado := viudo;
      'd','D': estado := divorciado;
    END; (* CASE aux1*)
    CASE estado OF
      casado: BEGIN
        Write('¿Nombre.....?');
        Readln(conyuge.nombre);
        Write('¿Edad.....?');
        Readln(conyuge.edad);
        Write('¿sexo (M/F).....?');
        Readln(aux1);
        IF Uppcase(aux1)='M'
        THEN conyuge.sexo:=m
        ELSE conyuge.sexo:=f;
        END;
      soltero, viudo, divorciado:;
    END; (* CASE estado *)
  END; (* WITH aux *)
  Write(fich, aux);
  Write('¿Más datos (s/n)?'); aux1:=Uppcase(Readkey);

```

EJECICIOS RESUELTOS

```
UNTIL aux1='N';
Close(fich);
END; (* CreaFich *)

(*****)

PROCEDURE ListaFichero(VAR fich:fichero);
VAR
    aux: ficha;
    aux1:char;
BEGIN
    Assign(fich, 'fich1.dat');
    Reset (fich);
    WHILE NOT Eof(fich) DO
    BEGIN
        Read(fich, aux);
        WITH aux DO
        BEGIN
            Write(datos.nombre:30,' ', datos.edad:2,' ');
            IF datos.sexo=m
            THEN Write('hombre ')
            ELSE Write('mujer' );
            CASE estado OF
            casado: BEGIN
                Writeln(' Casado con:');
                Write(conyuge.nombre:30, ' ');
                Write(conyuge.edad:2,' ');
                Write(';sexo (M/F).....?');
                Readln(aux1);
                IF conyuge.sexo=m THEN Write('Mujer')
                ELSE Write('Hombre');
            END;
            soltero: Writeln(' Soltero');
            viudo: Writeln(' Viudo');
            divorciado: Writeln(' Divorciado');
            END; (* CASE estado *)
        END; (* WITH aux *)
        Write('Pulse una tecla para continuar ');
        REPEAT UNTIL Keypressed;
    END; (* WHILE *)
END; (* ListaFich *)
```

11.16 Teniendo en cuenta las siguientes declaraciones globales:

```
CONST max=10; necmax=10;
TYPE vector = ARRAY [0..max] OF real;
    matriz = ARRAY [1..necmax] OF vector;
```

se puede representar un polinomio por una variable de tipo `vector` (el elemento que ocupa la posición `i` del vector será el coeficiente de grado `i` del polinomio), y un sistema de ecuaciones polinómicas por una variable del tipo `matriz`. Se pide:

- Escribir un subprograma que reciba un polinomio como argumento (el elemento que ocupa la posición `i` del vector será el coeficiente de grado `i` del polinomio) y devuelva al punto de llamada la derivada de dicho polinomio.
- Escribir un subprograma que lea de un fichero de texto los coeficientes de un sistema de ecuaciones polinómicas (una ecuación por línea) y los almacene en una estructura `ARRAY`. Debe calcular también el número de ecuaciones del sistema y devolverlo al punto de llamada.

c) Escribir un programa que:

- Lea un sistema de ecuaciones de un fichero de texto utilizando el subprograma anterior.
- Liste el sistema formado por las derivadas de las ecuaciones del sistema de partida, utilizando el subprograma del apartado a).

Observaciones: Si un polinomio P es de grado n ($n < \max$), serán ceros los elementos $P[n+1]$, $P[n+2]$, ..., $P[\max]$. En el fichero de texto figuran para cada polinomio los ceros correspondientes a coeficientes intermedios, pero no los ceros finales de relleno del vector.

El elemento que ocupa la posición cero del vector es el término independiente.

Solución

```
PROGRAM Polinomios(input,output,texto);
USES crt;
CONST
    max = 10;
    necmax = 10;
    precision = 1E-6;
TYPE
    vector = ARRAY [0..max] OF real;
    matriz = ARRAY [1..necmax] OF vector;
VAR
    P,dP,d2P: vector;
    S,dS: matriz;
    i,j,nec: integer;
    texto:text;

{-----}

PROCEDURE Derivar(VAR P, dP: vector);
(* Devuelve en DP la derivada de P *)
VAR
    i: integer;
BEGIN
    FOR i:=0 TO max-1 DO
        dP[i]:= (i+1)*P[i+1];
    dP[max]:=0;
END;

{-----}

PROCEDURE EscribePol(P: vector);
VAR i:integer;
BEGIN
    FOR i:=max DOWNT0 0 DO
        IF Abs(P[i]) > precision (* No se deben comparar reales con el
pequeño *)
            operador = pero si con un valor muy
        THEN
            BEGIN
                IF P[i] < 0 THEN Write ('-')
                    ELSE Write ('+');
                Write(Abs(P[i]):4:1,' x^',i);
            END;
        Writeln(' = 0');
    END;
```


EJECICIOS RESUELTOS

```

{-----}

PROCEDURE Inicializa(VAR S:matriz);
VAR i,j:integer;
BEGIN
  FOR i:=1 TO necmax DO
    FOR j:= 0 TO max DO S[i,j]:=0;
END;

{-----}

PROCEDURE LeeSistema(VAR texto:text; VAR S:matriz; VAR nec:integer);
VAR i,j:integer;
BEGIN
  Reset(texto);
  Writeln;
  Writeln('SISTEMA DE ECUACIONES INICIAL:');
  Writeln;
  i:=0;
  Inicializa(S);
  WHILE NOT (Eof(texto)) AND (i< necmax) DO
    BEGIN
      i:=i+1; j:=0;
      WHILE NOT Eoln(texto) AND (j<=max) DO
        BEGIN
          Read(texto, S[i,j]);
          j:=j+1;
        END;
      EscribePol(S[i]);
      Readln(texto);
    END;
  nec:=i;
END;

{-----}

PROCEDURE EscribeSistema(VAR texto:text; VAR S: matriz; nec:integer);
VAR i,j:integer;
BEGIN
  Rewrite(texto);
  FOR i:=1 TO nec DO
    BEGIN
      FOR j:=0 TO max DO Write(texto, S[i,j]:6:1, ' ');
      Writeln(texto);
    END;
  Close(texto);
END;

{***** Programa principal *****)}

BEGIN
  Clrscr;
  Assign(texto, 'a:\sistema.dat');
  LeeSistema(texto,S,nec);
  Inicializa(S);
  Writeln;
  Writeln('SISTEMA DE ECUACIONES DERIVADO:');
  Writeln;
  FOR i:=1 TO nec DO
    BEGIN
      Derivar(S[i],dS[i]);
      EscribePol(dS[i]);
    END;
  Writeln;
  Writeln (' Pulse un tecla ');
  Readln;
END.

```

11.15 EJERCICIOS PROPUESTOS

11.17 Diseñar los ficheros de datos necesarios para la gestión en una federación deportiva de: fichas de jugadores, partidos jugados, amonestaciones, sanciones, y clasificación. Realizar los programas necesarios para manejar los ficheros diseñados. En el caso de no entender muy bien el enunciado, realizar el análisis en la federación deportiva más próxima.

11.18 Un fichero de texto *f1* está constituido por una única línea muy larga. Se quiere reformatar dicho fichero dividiéndolo en varias líneas atendiendo al siguiente criterio:

- Se pasará a una nueva línea cada vez que se encuentre una palabra que termine en *e* o en *r*.

Ejemplo:

Si el fichero *f1* contiene el texto:

'Hay un lugar donde viven los gnomos con el que sueñan los niños'

se deberá crear un fichero *f2* cuya estructura sea:

```
'Hay un lugar
donde
viven los gnomos con el que
sueñan los niños'
```

- Para simplificar, se supone que las palabras están separadas exclusivamente por blancos.

11.19 A partir de un fichero de texto escrito en castellano, diseñar un programa que realice el cómputo de:

- a) Número total de letras.
- b) Número total de palabras.
- c) Número total de líneas que contengan información escrita, es decir, que no estén en blanco.

Se entiende por *palabra* toda secuencia de letras (mayúsculas, minúsculas, vocales acentuadas y eñes), que esté delimitada por alguno de los siguientes caracteres separadores:

- Espacio en blanco.
- Signos de puntuación (, . : ; ...)
- Marcas de fin de línea.

EJERCICIOS PROPUESTOS

- 11.20** Diseñar y escribir un programa de facturación para empresas. Deberá tener los ficheros de clientes y de artículos.
- 11.21** Diseñar y escribir un programa de gestión de un almacén.
- 11.22** Diseñar y escribir un programa de reservas en un hotel.
- 11.23** Diseñar la gestión de una biblioteca y escribir el programa. Se deberán contemplar: altas, bajas, modificaciones, consultas y préstamos.
- 11.24** Escribir un programa que reciba como entrada un fichero de texto y genere como salida otro fichero de texto formateado según unas condiciones leídas por teclado referentes a:
- Márgenes superior, inferior, izquierdo y derecho.
 - Número de líneas por página.
 - Espaciado (simple, doble, triple ...).
- 11.25** Escribir un programa que lea un fichero de texto y escriba otro fichero de texto que contenga las líneas del anterior ordenadas por orden alfabético. Suponer una longitud máxima de línea de 255 caracteres.
- 11.26** Escribir una función booleana que tome dos ficheros como entrada y devuelva *true* si ambos ficheros contienen tres registros idénticos y *false* en caso contrario. El tipo de los ficheros tiene la siguiente declaración de tipo:
- ```
TYPE
 TipoFichero = FILE OF tipoDatos;
```
- 11.27** Realizar nuevamente el ejercicio resuelto 11.2 sin utilizar la variable *primerCar*.
- 11.28** Modificar el ejercicio resuelto 11.14 de forma que permita realizar acceso directo sobre el fichero.

**11.29** Modificar el ejemplo 11.4 para que en el acceso secuencial no se utilice el vector `vtaux`.

### 11.16 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

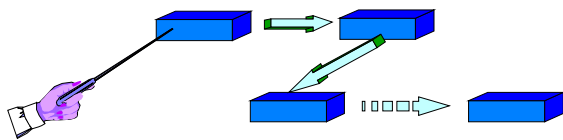
El lector que desee profundizar en la ordenación de ficheros, por distintos métodos, puede consultar la obra *Algoritmos+estructuras de datos= programas* de N. Wirth (Editorial Del Castillo, 1980), en su apartado 2.3 se dedica a la ordenación de ficheros secuenciales. Los algoritmos están implementados en Pascal.

El tipo abstracto de datos fichero y las técnicas de indexación se pueden consultar en el libro *Estructuras de datos. Realización en Pascal*. de M. Collado Machuca, R. Morales Fernández, y J.J. Moreno Navarro (Ed. Díaz de Santos, 1987).

Si se desea profundizar en la gestión del sistema de ficheros por el sistema operativo, la obra *Sistemas operativos* de H. M. Deitel (Addison-Wesley Iberoamericana, 1993) ofrece un estudio detallado.

Para aquellos lectores que encuentren atractivo el mundo de la Teleinformática y en particular las configuraciones de redes de transmisión de datos pueden remitirse a la obra *Introducción a la Teleinformática* de Eduardo Alcalde y Jesús García (McGraw-Hill, 1993). Para un mayor conocimiento de los protocolos de comunicaciones en redes locales, consultar la obra *LAN Protocol handbook*, de Mark A. Miller (Ed. Prentice-Hall y M&T books, 1990). También puede consultarse del mismo autor *InterNetWorking: a guide to network communications LAN to LAN; LAN to WAN* (Ed. Prentice-Hall y M&T books, 1991).

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS



## CAPITULO 12

### ESTRUCTURAS DINAMICAS DE DATOS

#### CONTENIDOS

- 12.1 Introducción
- 12.2 El tipo puntero
- 12.3 Estructuras dinámicas de datos lineales
- 12.4 Algoritmos de tratamiento de listas simplemente enlazadas
- 12.5 Visión recursiva de una lista
- 12.6 Otros tipos de listas: Pilas, colas, listas circulares
- 12.7 Estructuras dinámicas de datos no lineales
- 12.8 Extensiones del compilador Turbo Pascal
- 12.9 Gestión de memoria dinámica en Turbo Pascal
- 12.10 Ejercicios resueltos
- 12.11 Ejercicios propuestos
- 12.12 Ampliaciones y notas bibliográficas

## INTRODUCCION

### 12.1 INTRODUCCION

Todas las estructuras de datos estudiadas hasta ahora (con excepción de los ficheros) son *estáticas*, es decir no pueden cambiar de tamaño durante la ejecución del programa. En este tema se estudiará cómo pueden definirse y utilizarse *estructuras dinámicas de datos*.

Los tipos de Pascal *ARRAY* y *RECORD* permiten definir estructuras *estáticas* de datos. Se puede determinar el tamaño de una estructura estática examinando las declaraciones del programa, ya que el número máximo de elementos es especificado de forma directa (caso de los arrays) o indirectamente (caso de los registros). Aunque durante la ejecución del programa no se utilicen todos sus elementos, se reserva memoria para almacenar la estructura completa.

Otro inconveniente de los arrays es que sus elementos deben ocupar posiciones físicamente consecutivas de memoria.

Las *estructuras dinámicas de datos* son aquellas cuyos elementos (desde el primero) son *datos dinámicos*, que se van creando y eliminando en tiempo de ejecución. La memoria ocupada por datos dinámicos es gestionada en tiempo de ejecución, mientras que para los datos estáticos se gestiona la asignación de memoria durante la fase de compilación.

El tamaño de un elemento de una estructura dinámica en lenguaje Pascal también puede deducirse de la declaración de tipo correspondiente, según se verá más adelante. Pero las variables dinámicas no se crean mediante una declaración, sino que se crean y se destruyen durante la ejecución del programa utilizando punteros, que se estudian en la siguiente sección. El tamaño de la estructura completa no puede determinarse a partir de las declaraciones, ya que no tiene un número máximo de elementos. Estos se van añadiendo o eliminando según se necesitan en tiempo de ejecución.

Para comprobar la utilidad de las estructuras dinámicas de datos, veamos un ejemplo.

#### Ejemplo 12.1

Supongamos que se tiene un lista de elementos, que se podría representar como un *array*, con la siguiente declaración:

```
CONST n = 100;
VAR
 lista : ARRAY [1..n] OF elemento;
```

donde *elemento* puede ser un tipo simple, o un tipo estructurado.

Definir así una lista plantea varios problemas:

- ⌘ El número máximo de elementos está limitado a priori.
- ⌘ Si los elementos de la lista están ordenados siguiendo un criterio determinado, y se desea hacer una inserción de un nuevo elemento, de modo que la lista siga ordenada, hay que desplazar todos los elementos a la derecha del punto de inserción; Ejemplo: Sea una lista de números enteros ordenados de menor a mayor:

ESTRUCTURAS DINAMICAS DE DATOS

|   |    |     |      |      |     |     |
|---|----|-----|------|------|-----|-----|
| 1 | 27 | 103 | 1001 | 3321 | ... | ... |
|---|----|-----|------|------|-----|-----|

Si se desea insertar 33, primero hay que buscar la posición correspondiente y desplazar hacia la derecha (empezando por el final) todos los elementos posteriores:

|   |    |  |     |      |      |     |
|---|----|--|-----|------|------|-----|
| 1 | 27 |  | 103 | 1001 | 3321 | ... |
|---|----|--|-----|------|------|-----|

Después de la inserción, la nueva lista queda:

|   |    |    |     |      |      |     |
|---|----|----|-----|------|------|-----|
| 1 | 27 | 33 | 103 | 1001 | 3321 | ... |
|---|----|----|-----|------|------|-----|

Por otra parte si se elimina un componente de la lista y se desea que no queden huecos, se han de desplazar todos los elementos a su derecha un lugar a la izquierda.

Es evidente que las operaciones de inserción y borrado en una lista ordenada, con una estructura como la anterior, requieren un tiempo adicional para la reubicación de los elementos de la lista.

Una forma de resolver el problema anterior es mediante una *estructura dinámica de datos*, llamada **lista encadenada**, en la cual cada elemento tiene un enlace indicando quien es su sucesor, tal como se representa en la figura 12.1.

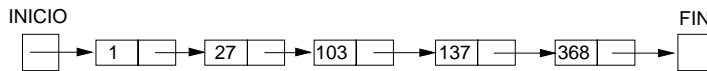


Figura 12.1 Ejemplo de lista encadenada

En la sección 12.4 se incluyen las declaraciones necesarias para manejar en Pascal una lista encadenada. Con este tipo de estructura:

- No se necesita conocer a priori el número de elementos de la lista.
- Una operación de inserción no obliga a desplazar los elementos, ni tampoco las operaciones de borrado. Para mantener los elementos ordenados basta con reajustar los enlaces, tal como se indica en la figura 12.2. Solamente se modifica el campo que señala al siguiente elemento.



## EL TIPO PUNTERO

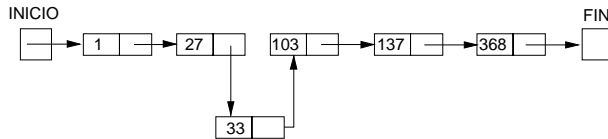


Figura 12.2 Inserción en lista encadenada

## 12.2 EL TIPO PUNTERO

La herramienta que permite la creación de estructuras dinámicas de datos son los *punteros*, también llamados *apuntadores* en algunos libros, y que corresponde a la traducción de la palabra inglesa *pointer*.

El tipo *puntero* es un tipo simple, como integer, real, char, ... Pero a diferencia de los otros tipos simples no tiene un identificador estándar. El identificador de un tipo puntero consiste en una flecha ( $\uparrow$ ) seguida del identificador del tipo al cual apunta. En la mayoría de las implementaciones se sustituye la flecha por el acento circunflejo ( $\wedge$ ), para facilitar su escritura, ya que este último símbolo se obtiene directamente pulsando una tecla.

Un *puntero* es un tipo de variable usada para almacenar la *dirección en memoria* de otra variable, en lugar de un dato convencional (números, caracteres, etc.). Mediante la variable de tipo *puntero* accedemos a esa otra variable, almacenada en la dirección de memoria que señala el *puntero*. Es decir, el valor de la variable de tipo *puntero* es una *dirección de memoria*. Se dice que el puntero *señala* o *apunta* a la variable almacenada en la dirección de memoria que contiene el puntero. Lo que nos interesa es el dato contenido en esa variable apuntada. Es fácil confundir la *variable apuntada* con el *puntero*; para evitarlo, insistiremos en diferenciarlas en todo el capítulo.

El diagrama sintáctico de la definición del tipo puntero es el representado en la figura 12.3.

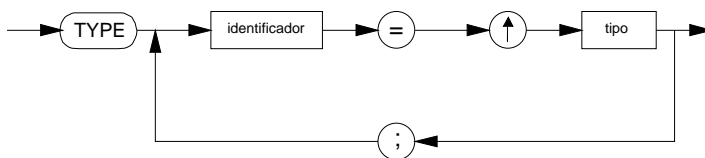


Figura 12.3 Diagrama sintáctico del tipo puntero

En notación EBNF su sintaxis es:

```
<tipo puntero> ::= \uparrow <identificador de tipo>
```

**Ejemplo 12.2**

Si se desea construir la lista encadenada, que se mostró en la introducción, se declaran los tipos:

```
TYPE
 puntero= ^nodo;
 nodo = RECORD
 info: integer;
 sig : puntero
 END;
```

Es decir, los *nodos* de la lista serán *registros* con dos campos, en uno (campo de información) se guarda el valor de un número entero, y en el otro, de tipo `puntero`, la dirección del siguiente *nodo*. Se dice que este campo *apunta al elemento siguiente*. Un nodo se suele representar como en el esquema de la figura 12.4.

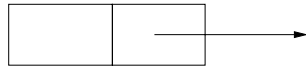


Figura 12.4 Nodo de una lista enlazada

Nótese que, como excepción de la regla general, el tipo empleado en la definición de `puntero`, puede ser utilizado antes de ser definido.

**Declaración de variables de tipo puntero**

Las variables de tipo *puntero* se declaran de forma convencional, siguiendo el esquema:

```
VAR
 identificador : tipo;
```

**Ejemplo 12.3**

Continuando con el ejemplo anterior, veamos las siguientes declaraciones:

```
PROGRAM Ejemplo (input, output)
TYPE
 puntero= ^nodo;
 nodo = RECORD
 info: integer;
 sig : puntero
 END;
VAR p, q, r: puntero ;
 ...
```

Las variables `p`, `q`, `r`, creadas mediante esta declaración, son *estáticas*, es decir convencionales, como las que hemos visto hasta ahora. Podrán apuntar a variables de tipo `nodo`. Una variable de tipo `puntero` contiene la dirección de memoria de una variable de tipo `nodo`. Las variables apuntadas se llaman *variables referenciadas*. La memoria para almacenar estas variables

## EL TIPO PUNTERO

referenciadas se reserva en *tiempo de ejecución*, por ello los punteros permiten gestionar variables y estructuras dinámicas. *Puntero* y *variable referenciada* se suelen representar gráficamente como se indica en la figura 12.5.

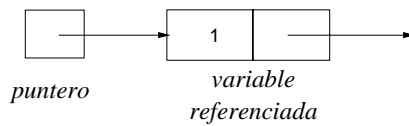


Figura 12.5 Variable referenciada

Las variables de tipo puntero, pueden apuntar a cualquier tipo de datos, tal y como se mostró en el diagrama sintáctico de la figura 12.3. También se pueden construir estructuras de datos cuyos elementos sean punteros. Por lo tanto son válidas las declaraciones siguientes:

```
PROGRAM Ejemplo_punteros (input, output)
TYPE
 Pentero= ^integer;
 Preal=^real;
 Pcharacter=^char;
 Pnodo=^Tnodo;
 Tnodo=RECORD
 info: integer;
 sig : Pnodo;
 END;
 TarrayPunteros= ARRAY [1..100] OF Pnodo;
VAR
 p:Pentero;
 q:Preal;
 r:Pcharacter;
 s:Pnodo;
 t:TarrayPunteros;
...
```

Se acostumbra a comenzar los identificadores de tipo puntero con una P, y los identificadores del resto de los tipos de datos con una T.

### Operaciones con punteros

Veamos que operaciones están permitidas con variables de tipo puntero. Recordemos que las variables de tipo *puntero* son *estáticas*, a diferencia de las *variables referenciadas* por punteros, que son las *dinámicas*.

#### • Asignación

Con variables de tipo puntero se pueden realizar sentencias de asignación.

### Ejemplo 12.4

Sean *p* y *q* dos variables de tipo *puntero* que señalan a dos nodos distintos, representadas en la figura 12.6.

## ESTRUCTURAS DINAMICAS DE DATOS

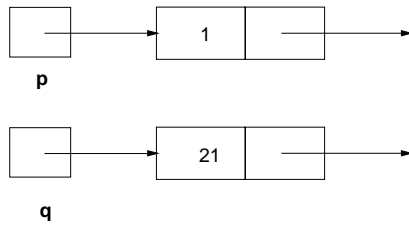


Figura 12.6 Punteros  $p$  y  $q$  antes de la asignación

Si realizamos la sentencia de asignación:

$p := q;$

entonces tenemos que  $p$  apunta al mismo nodo que  $q$ , ya que  $p$  contendrá la misma dirección de memoria que  $q$ , situación representada en la figura 12.7.

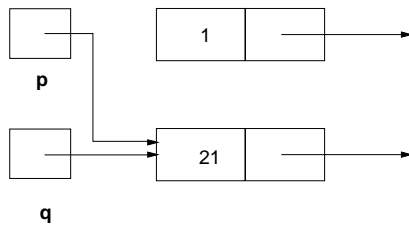


Figura 12.7 Punteros  $p$  y  $q$  después de la asignación

### • Constante NIL

Para indicar que una variable de tipo *puntero* no apunta a nada, se le asocia la constante predefinida *NIL*. Esta es una palabra reservada del lenguaje Pascal, que se utiliza para inicializar variables de tipo *puntero*, y para señalar el final de una lista.

### Ejemplo 12.5

Se puede asignar a una variable de tipo *puntero*:

$p := NIL;$

Las listas encadenadas acaban con *NIL*, como se observa en la figura 12.8.

## EL TIPO PUNTERO

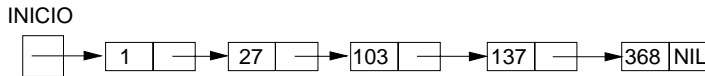


Figura 12.8 Lista encadenada acabada en NIL

En algunos libros *NIL* se representa como la toma a tierra en electricidad, o con el símbolo *lambda* mayúscula ( $\Lambda$ ) en estudios teóricos de estructuras de datos.

### • Comparaciones

Se pueden utilizar los operadores de relación = y <> para comparar variables de tipo *puntero*. Los demás operadores relacionales no tienen sentido aquí.

### Ejemplo 12.6

Veamos algunas expresiones lógicas con *punteros*:

- $p <> q$  será cierta si  $p$  y  $q$  apuntan a variables diferentes
- $p = q$  será cierta si  $p$  y  $q$  apuntan a la misma variable (es decir, contienen la misma dirección de memoria)
- $p = \text{NIL}$  será cierto si  $p$  contiene el valor *NIL*.

### Creación de variables dinámicas

*Variable dinámica* o *variable referenciada* es una variable a la que se accede a través de una variable *puntero* y no por su nombre. En notación EBNF se representa así:

```
<variable referenciada> ::= <variable puntero> ^
```

En la mayoría de los compiladores de Pascal se puede representar por el siguiente esquema:

```
<variable puntero> ^
```

Gráficamente, la relación entre la variable de tipo puntero y la variable referenciada se expresa como en la figura 12.9.

Las *variables referenciadas* son variables *dinámicas*. Una variable *dinámica* se crea y se destruye en tiempo de ejecución. Es decir las posiciones de memoria que ocupa se asignan y se desasignan durante la ejecución del programa. Sin embargo el resto de las variables estudiadas del lenguaje Pascal son *estáticas*, es decir sus posiciones de memoria relativas se asignan en tiempo de compilación, pasándose a absolutas en la fase de montaje (*link*).

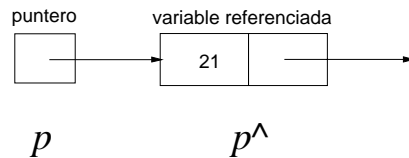


Figura 12.9 Puntero y variable referenciada

Las variables de tipo *puntero* tienen asignación *estática* de memoria. El puntero *existe* durante toda la ejecución del subprograma (o programa principal) en que se ha declarado. Esta es la diferencia entre el *puntero* y la *variable apuntada*.

Además, recordemos que las variables de tipo *puntero* declaradas **no se inicializan automáticamente**. Inicialmente, las variables de tipo *puntero* tienen un valor indeterminado, al igual que cualquier otra variable. Contendrán *basura* hasta que las inicialicemos (indicarán una dirección de memoria arbitraria).

La declaración de una variable *puntero* crea el *puntero*, pero no la variable a la que apunta.

Para crear una variable referenciada se utiliza el procedimiento estándar *New*, de la forma:

```
New(variablePuntero);
```

### Ejemplo 12.7

Para crear la variable referenciada del ejemplo anterior,  $p^$ , se haría:

```
New (p);
```

Las posiciones de memoria reservadas, en las cuales se introducirá la variable referenciada, son del tipo al cual apunta el *puntero*.

El procedimiento *New* crea el espacio necesario para almacenar la variable referenciada en la memoria, y hace que  $p$  apunte a esa variable. Es decir en  $p$  se almacena la dirección de memoria donde comienza el espacio reservado para la variable referenciada.

### Supresión de variables dinámicas

También existe un procedimiento estándar, *Dispose*, para borrar una variable referenciada y liberar la memoria que estaba utilizándose para su almacenamiento. La sintaxis del procedimiento *Dispose* es la siguiente:

```
Dispose (variable puntero);
```

**Ejemplo 12.8**

Para liberar la memoria ocupada por la variable  $p^{\wedge}$ , creada en el ejemplo 12.7, escribiríamos:

```
Dispose(p);
```

**Asignación de variables referenciadas**

Con las variables referenciadas, pueden hacerse las mismas operaciones que con las variables ordinarias de su mismo tipo, que ya estamos acostumbrados a usar. Es importante hacer notar la diferencia entre las sentencias de asignación

$$p := q \quad \text{y} \quad p^{\wedge} := q^{\wedge}$$

En la figura 12.10 se aclara esta distinción.

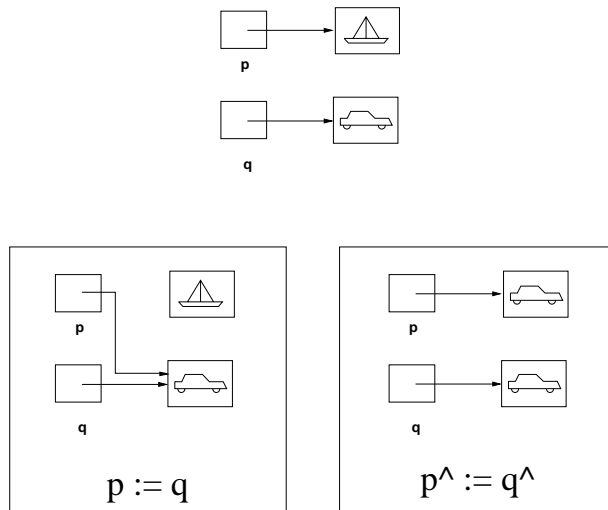


Figura 12.10 Diferencia entre  $p := q$  y  $p^{\wedge} := q^{\wedge}$

**12.3 ESTRUCTURAS DINAMICAS DE DATOS LINEALES**

Una estructura de datos es *dinámica* si sus componentes se van creando o eliminando en tiempo de ejecución, a medida que se necesitan. La *lista simplemente enlazada* o *lista lineal* es el ejemplo más sencillo de *estructura dinámica de datos*. El concepto de lista simplemente enlazada es fundamental en el desarrollo de este capítulo.

## Definición

Una *lista* es una sucesión de un *número variable* de elementos del *mismo tipo* denominados *nodos* o elementos de la lista, entre los cuales existe una acción simple que permite moverse de un elemento al siguiente, si existe.

La representación gráfica más común de una *lista* es la mostrada en la figura 12.8.

## Implementación

Veamos tres maneras de construir esta estructura:

- Realización práctica con *arrays*: no vamos a desarrollar este caso. Se puede implementar de diversas formas. Por ejemplo, se puede diseñar mediante un *array de registros*, de manera que cada elemento del ARRAY tenga uno o varios campos de información, y un campo de tipo *índice*, con la posición del siguiente elemento. Si se desea información sobre este tipo de implementación de listas, consultar la bibliografía recomendada al final del capítulo.
- Realización práctica con *ficheros*: es similar al caso anterior de *arrays*, pero con ficheros de acceso directo, y sustituyendo los índices del array por las posiciones de los registros en el fichero. Tampoco se va a desarrollar este caso.
- Realización práctica con *punteros*: Es el tipo de implementación que utilizaremos. Mediante *punteros* se van creando nodos de la lista, que se insertan en la misma reajustando los enlaces entre sus nodos.

## Tipo Abstracto de Datos lista

Para construir un *TAD lista* hay que:

- Definir los *valores* que pueden tomar los elementos de este tipo (no confundir con el tipo de los elementos de la lista).
- Construir las *operaciones* básicas para el manejo de listas.

Recordemos el concepto de *encapsulamiento de datos*: el acceso a los datos debe hacerse a través de las operaciones diseñadas. Los algoritmos que representan las operaciones básicas deben servir tanto para la realización con *arrays* o *ficheros* como con *punteros*, sin más que cambiar las declaraciones de los datos y estructuras de datos usadas.

## Puntos a destacar

- Estructura de datos *dinámica*: sus elementos pueden ser creados y suprimidos en función de las necesidades del tratamiento.
- Lista *vacía*: si no contiene ningún elemento.



## ESTRUCTURAS DINAMICAS DE DATOS LINEALES

- Debemos disponer de un mecanismo que nos permita:

- 1) Acceder al primer elemento de la lista
- 2) A partir de éste, a los restantes.
- 3) Detectar el final de la lista

Para definir y utilizar un TAD lista se necesita:

- ⊠ Un *puntero* externo a la cabecera de la lista.
- ⊠ Definir el tipo de sus elementos o *nodos*.
- ⊠ Un mecanismo para detectar el *final de la lista*.
- ⊠ Primitivas para *crear/eliminar* elementos. Otras operaciones básicas.

Examinemos más detenidamente estos cuatro elementos necesarios:

- ⊠ El *puntero externo a la cabecera* contiene la dirección del primer nodo de la lista. Debe ser accesible desde fuera del TAD. Es nuestro único medio de acceder a la lista, razón por la cual hay que tener mucho cuidado al operar con él.

La lista está *vacía* (no contiene ningún nodo) cuando el *puntero externo* a la lista tiene el valor *NIL*.

- ⊠ *Anatomía de un nodo*

En general es un registro compuesto por:

- Uno o más campos de información
- Un campo de tipo *puntero*, que contendrá la dirección del *siguiente nodo* de la lista. Se dice que *apunta al siguiente elemento*.

Gráficamente, un nodo se representa como se indica en la figura 12.11.

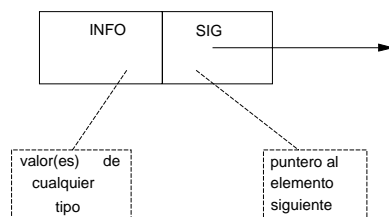


Figura 12.11 Nodo de una lista simplemente enlazada

- ⊠ *Detección de fin de lista*: El campo *SIG* del último nodo tiene el valor *NIL*.

### ⌘ Algoritmos de tratamiento de listas simplemente enlazadas

Las operaciones básicas para trabajar con listas, que se estudiarán en la siguiente sección, son las siguientes:

- Creación de una lista.
- Recorrido de una lista.
- Búsqueda de elementos.
- Inserción de nuevos elementos.
- Creación y mantenimiento de listas ordenadas.
- Supresión de elementos.

En el ejercicio resuelto 12.20 se implementa mediante una *unit* un TAD lista que incluye las operaciones básicas mencionadas.

## 12.4 ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

Para poder construir un TAD lista, mediante una *unit* de Turbo Pascal, además de incluir las definiciones de tipo necesarias para utilizar la estructura, es necesario disponer de un repertorio de operaciones suficiente, tal que sea innecesario conocer el contenido de la *unit*, y los detalles internos de la estructura.

En todos los algoritmos estudiados supondremos para los nodos la estructura de la figura 12.12.

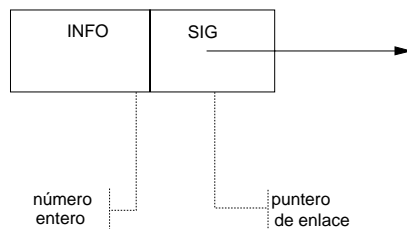


Figura 12.12 Anatomía de un nodo

### Notación algorítmica utilizada

Antes de abordar la codificación en Pascal de cada una de las operaciones sobre listas, desarrollaremos el algoritmo correspondiente utilizando la siguiente notación:

- |          |                                         |
|----------|-----------------------------------------|
| CABEZA   | puntero externo a la lista.             |
| P        | puntero auxiliar a un nodo de la lista. |
| NODO (P) | nodo apuntado por P.                    |

## ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

INFO (P) campo de información del NODO (P).  
SIG (P) campo puntero del nodo apuntado por P.  
Crear(P) crea un nuevo nodo apuntado por P.  
Liberar(P) destruye la variable apuntada por P.

### Declaración en Pascal

```
TYPE
 puntero = ↑ nodo;
 nodo = RECORD
 info: integer;
 sig: puntero;
 END;
VAR
 cabeza: puntero;
```

Utilizando la notación algorítmica que acabamos de presentar, vamos a desarrollar las operaciones básicas para el manejo de listas simplemente enlazadas, primero en lenguaje algorítmico y a continuación en lenguaje Pascal.

### CREACION DE UNA LISTA

El método más simple para crear una lista consiste en ir añadiendo elementos al principio de la misma, delante de su primer nodo.

Partimos inicialmente de una lista vacía, como la de la figura 12.13.



Figura 12.13 Lista vacía

Si llamamos VALOR a la variable de tipo entero sobre la que obtenemos los sucesivos datos, un *primer intento* nos llevaría a la situación de la figura 12.14.

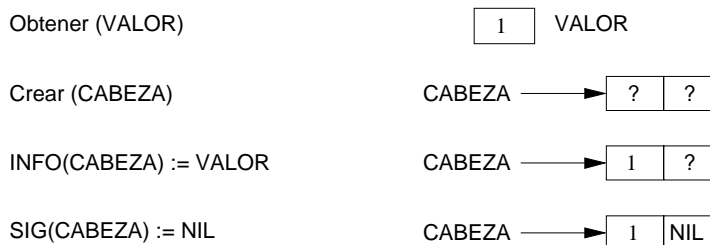


Figura 12.14 Primer intento de creación de lista enlazada

¿Cómo añadir nuevos elementos?

No podemos hacer de nuevo:



¡PERDEMOS LA LISTA ORIGINAL!

Figura 12.15 Intento de añadir otro elemento

Para resolver este problema, utilizaremos un *puntero* auxiliar P para crear los nuevos nodos, que iremos insertando en la cabecera, utilizando el siguiente algoritmo.

### Algoritmo general de inserción en la cabecera

```

ACCION CreaLista ES
CABEZA := NIL;
 {Inicialización: lista vacía}
MIENTRAS haya datos a insertar HACER
(1) Obtener(VALEN);
(2) Crear(P);
(3) INFO(P) := VALOR;
(4) SIG(P) := CABEZA;
(5) CABEZA := P;
FIN_MIENTRAS;
FIN_ACCION;

```

Obsérvese que el algoritmo funciona también para insertar el primer elemento (partimos de una lista vacía).

Suponiendo una lista ya creada con tres elementos: 3, 7 y 4. Para insertar uno nuevo hay que seguir los cinco pasos indicados, representados gráficamente en la figura 12.16.

### Observaciones

- Los elementos quedan en orden inverso al de llegada. Esta estructura se llama lista *LIFO*: *Last In, First Out* (en español: *último en entrar, primero en salir*).
- Otra alternativa consiste en la inserción al final, pero es más lento. Obtendríamos una lista *FIFO*: *First In, First Out* (en español: *primero en entrar, primero en salir*). Requiere la utilización de dos punteros auxiliares, y el reajuste de enlaces conlleva más operaciones. Se utiliza cuando nos interesa construir una lista *FIFO*.

## ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

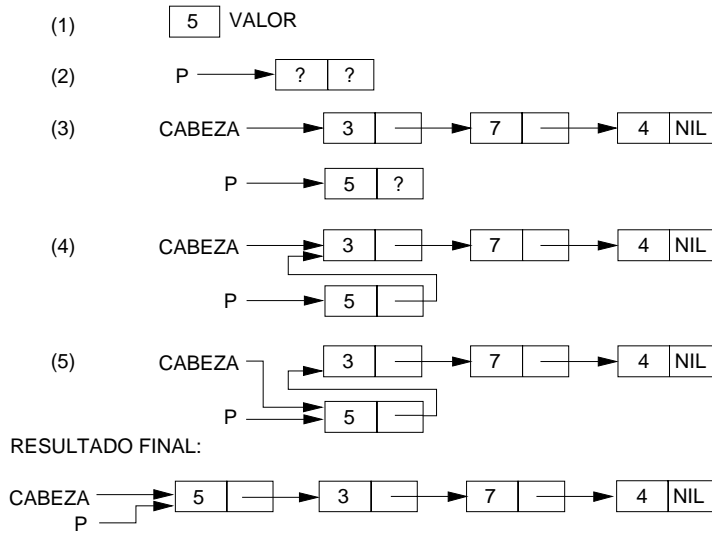


Figura 12.16 Creación de lista enlazada

### Ejemplo 12.9

Para traducir el algoritmo a lenguaje Pascal vamos a aplicarlo al caso práctico de *crear una lista con números enteros leídos desde el fichero input*.

Construiremos un procedimiento, identificado como *CrearLista*, cuya llamada sea:

```
CrearLista(inicio);
```

### Codificación en Pascal

```
PROCEDURE CrearLista(VAR cabeza: puntero);
VAR p: puntero;
 valor: integer;
BEGIN
 cabeza := NIL; {Inicialización}
 WHILE NOT Eof(input) DO { El proceso finaliza cuando se teclea }
 BEGIN { la marca de fin de fichero, Ctrl-Z }
 Read(valor);
 New(p);
 p^.info := valor;
 p^.sig := cabeza;
 cabeza := p;
 END;
END;
```

## Observaciones

- El parámetro *cabeza* se declara con comunicación por dirección, para que el argumento *inicio* sea el *puntero* externo a la lista después de la llamada.
- Hay que resaltar que el código es una traducción directa a Pascal de la notación algorítmica.

## RECORRIDO DE UNA LISTA

En muchas aplicaciones es necesario visitar todos los nodos de una lista para efectuar algún tipo de tratamiento con ellos. Esta operación se realiza utilizando un *puntero auxiliar*, *P*. No podemos movernos directamente con *CABEZA*: **¡Perderíamos la lista!**

Inicialmente el *puntero auxiliar* toma el valor de *CABEZA*, con lo que accedemos al primer nodo de la lista. Después de tratar cada elemento asignamos al *puntero auxiliar* el valor del campo *SIG* de dicho elemento, para movernos al siguiente nodo. Repetiremos esta operación hasta detectar el final de la lista, momento en el cual el *puntero auxiliar* tomará el valor *NIL*. El proceso se representa gráficamente en la figura 12.17.

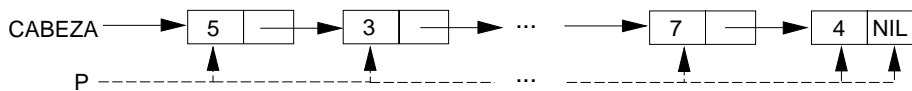


Figura 12.17 Recorrido de una lista

## Algoritmo

```

ACCION RecorrerLista ES
P := CABEZA; {Inicialización}
MIENTRAS No(Fin de Lista) HACER
 Tratar(INFO(P));
 P := SIG(P);
FIN_MIENTRAS;
FIN_ACCION;

```

## Ejemplo 12.10

Este algoritmo tan sencillo es muy utilizado en numerosas aplicaciones. Vamos a aplicarlo al caso práctico de *Imprimir la lista creada anteriormente*.

Escribiremos un *procedimiento*, identificado como *ImprimirLista*, cuya llamada sea:

```
ImprimirLista(cabeza);
```

## ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

### Codificación en Pascal

```
PROCEDURE ImprimirLista(p: puntero);
{Imprime la lista apuntada por p}
BEGIN
{La inicialización p := cabeza se realiza en la llamada}
 WHILE p <> NIL DO {¿es fin de lista?}
 BEGIN
 Writeln (p^.info); {imprime el valor}
 p:= p^.sig; {avanza al siguiente}
 END;
END;
```

### Observaciones

- La inicialización  $p := cabeza$  se efectúa directamente en la llamada.
- Al ser  $p$  un parámetro *por valor*, puedo moverme con él sin perder el *puntero externo* a la lista.

### BUSQUEDA EN UNA LISTA

La búsqueda de un elemento concreto, solo puede hacerse **secuencialmente**. Es decir, empezando desde el primer elemento, y recorriendo la lista nodo a nodo, pasando de cada elemento al siguiente. Termina cuando se cumple una de las dos condiciones siguientes:

- Se encuentra el elemento.
- Se alcanza el final de la lista.

Gráficamente, la búsqueda en una lista se representa en la figura 12.18.

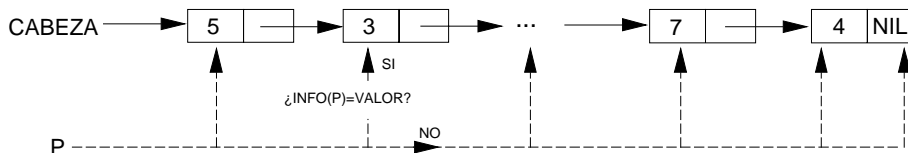


Figura 12.18 Búsqueda en una lista

Variables usadas:

- VALOR (tipo entero): valor a buscar.
- ENCONTRADO (tipo lógico): Variable auxiliar, que tomará el valor falso mientras no se encuentre el VALOR.
- P: puntero auxiliar para recorrer la lista. Al final, P apunta al elemento buscado o devuelve el valor *NIL* si no se encontró.

**Algoritmo**

```

ACCION BuscarValorEnLista ES
P := CABEZA; {Inicialización}
MIENTRAS No(Fin de Lista) Y NO(ENCONTRADO)_HACER
 SI INFO(P) = VALOR
 ENTONCES ENCONTRADO := cierto
 SI_NO P := SIG(P);
 FIN_SI;
FIN_MIENTRAS;
FIN_ACCION;

```

**Ejemplo 12.11**

Vamos a aplicarlo al caso práctico de *buscar un valor de tipo entero en la lista utilizada en los ejemplos anteriores*.

Escribiremos un subprograma *función*, identificado como *BuscarEnLista*, con dos parámetros:

- puntero externo a la lista, *p*.
- valor que se quiere buscar, *valor*.

La llamada será del tipo:

```
q := BuscarEnLista(cabeza, num);
```

siendo *q*, *cabeza* y *num* variables del tipo adecuado.

El resultado de la llamada será un *puntero* al elemento de la lista que contiene en su campo de información el número *valor*, o *NIL* si no lo encontró.

**Codificación en Pascal**

```

FUNCTION BuscarEnLista(p: puntero; valor: integer): puntero;
VAR encontrado: boolean;
BEGIN
 encontrado := false;
 WHILE (p <> NIL) AND (NOT encontrado) DO
 IF p^.info = valor
 THEN encontrado := true {Finaliza la búsqueda}
 ELSE p := p^.sig; {Avanzar al siguiente}
 BuscarEnLista := p;
END;

```

**Observaciones**

- No se puede simplificar el bucle poniendo:

```
WHILE (p <> NIL) AND (p^.info <> valor) DO p := p^.sig;
```

ya que *¡ SE PRODUCIRIA UN ERROR AL LLEGAR A FIN DE LISTA !*



## ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

Aparentemente este bucle es equivalente al anterior, pero sólo funcionaría en el caso de que el valor buscado se encontrase en la lista. En caso contrario se produciría un error de ejecución al llegar a fin de lista, pues cuando  $p$  toma el valor  $NIL$  no existe nodo apuntado por  $p$  ( $p^{\wedge}$  en Pascal). Lo mismo sucedería si la lista está inicialmente vacía.

- Con listas ordenadas, el proceso de búsqueda puede terminar antes. (Suponiendo la lista ordenada ascendentemente, basta con encontrar un nodo que contenga un valor mayor que el buscado para deducir que éste no está en la lista. Este caso se estudia más adelante).

## INSERCIÓN EN LISTAS

Se parte de una lista ya creada, y se desea insertar un nuevo nodo en medio de la lista. Existen dos casos posibles:

- Inserción *detrás* de NODO (P)
- Inserción *delante* de NODO (P)

### • Inserción *detrás*

Creamos un nuevo nodo (mediante un *puntero auxiliar*) y asignamos el valor a insertar al campo de información. Supongamos  $VALOR=33$ . A continuación reajustamos los enlaces para colocarlo en la lista, según se indica en la figura 12.19.

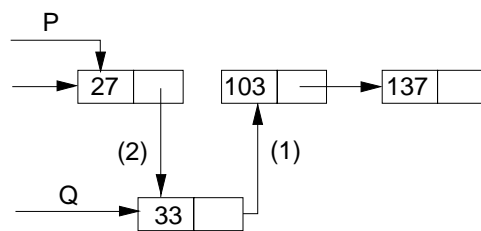


Figura 12.19 Inserción detrás

### Algoritmo

```
Crear(Q)
INFO(Q) := VALOR
(1) SIG(Q) := SIG(P)
(2) SIG(P) := Q
```

**Codificación en Pascal**

```

New (q); { q : puntero auxiliar }
q^.info := valor;
q^.sig := p^.sig;
p^.sig := q;

```

Puede comprobarse que el algoritmo sirve también si tratamos de insertar detrás del último elemento de la lista.

**• Inserción delante**

En este caso surge un problema ya que *NO PODEMOS ACCEDER AL NODO PRECEDENTE*, para modificar su *puntero* de enlace. Para solucionarlo se recurre al siguiente artificio:

Se crea el nuevo nodo (NODO(Q)), se intercambian los valores de ambos nodos, y a continuación se inserta el nuevo nodo detrás de NODO(P), como se indica en la figura 12.20.

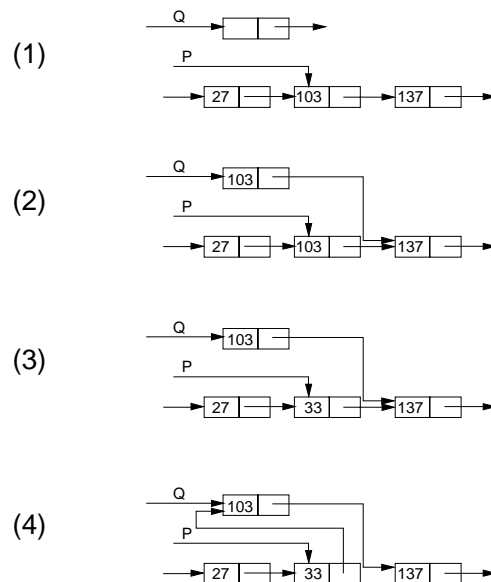


Figura 12.20 Inserción delante

## ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

### Algoritmo

```
(1) Crear (NODO (Q))
(2) NODO (Q) := NODO (P)
(3) INFO (P) := VALOR
(4) SIG (P) := Q
```

El algoritmo también sirve para insertar delante del primer elemento de la lista.

### CREACION Y MANTENIMIENTO DE LISTAS ORDENADAS

Los algoritmos anteriores se aplican para *crear* y *mantener* una lista ordenada. Se insertan los elementos en el lugar que les corresponda de manera que la lista se mantenga ordenada. Los pasos a seguir serán:

- Encontrar el lugar donde insertar el nuevo valor.
- Insertar el nuevo valor.

Supondremos que la lista está ordenada ascendentemente.

Un primer intento de construir un algoritmo nos llevaría a:

```
SI lista_vacía
 ENTONCES Insertar a la cabeza
 SI_NO P := CABEZA;
 lugar_encontrado := falso;
 MIENTRAS (NO(Fin de Lista) Y
 NO(lugar_encontrado)) HACER
 SI INFO (P) > valor
 ENTONCES
 lugar_encontrado := cierto
 SI_NO
 P := SIG (P);
 FIN_SI;
 FIN_MIENTRAS;
 SI lugar_encontrado
 ENTONCES Insertar delante de NODO (P)
 SI_NO Insertar detrás del último elemento;
 FIN_SI;
FIN_SI;
```

Pero surge un problema: en el último SI\_NO ya hemos perdido la referencia del último elemento, pues P vale NIL.

Se puede solucionar introduciendo el último condicional dentro del bucle, con lo que obtendríamos el siguiente esquema:

```
SI lista_vacía
 ENTONCES
 Insertar a la cabeza
 SI_NO
 P := CABEZA;
```

## ESTRUCTURAS DINAMICAS DE DATOS

```
lugar_encontrado := falso;
MIENTRAS (NO lugar_encontrado) HACER
 SI INFO (P) > valor
 ENTONCES
 lugar_encontrado := cierto
 Insertar delante de NODO (P)
 SI_NO
 SI SIG (P) = NIL {último elemento}
 ENTONCES
 lugar_encontrado := cierto;
 Insertar detrás de NODO(P)
 SI_NO
 P := SIG (P)
 FIN_SI;
 FIN_SI;
FIN_MIENTRAS;
FIN_SI;
```

El algoritmo es válido, pero puede ser intolerable Si la lista es larga pues:

- hay que hacer muchas comparaciones dentro del bucle.
- utiliza tres tipos de inserción diferentes, lo que se traduce en un aumento considerable del código generado.
- Se plantean casos particulares cuando:
  - ✕ la lista está vacía
  - ✕ hay que insertar al final de la lista.

### *Una posible solución*

En muchas ocasiones es posible simplificar el algoritmo anterior inicializando la lista con un elemento ficticio que sea siempre mayor que cualquier otro.

### **Ejemplo 12.12**

Tal es el caso de la ordenación alfabética de nombres. Nunca habrá un nombre que sea mayor que 'ZZZZ...ZZZ'. Obsérvese no obstante que en el caso de palabras escritas en castellano habrá que buscar otro carácter diferente a la 'Z' cuyo ordinal fuese mayor que el de las *eñes* y vocales acentuadas.

Elegiremos como elemento ficticio 'ZZZZ...ZZZ'. Será el primer elemento de la lista, el señalado por CABEZA.

Ahora ya no hay casos particulares, pues:

- la lista nunca está vacía
- nunca habrá que insertar al final.

## ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

El algoritmo se reduce entonces a:

```
P := CABEZA;
MIENTRAS INFO(P) > valor HACER
 P := SIG(P);
FIN_MIENTRAS;
Insertar delante de NODO(P)
```

Obsérvese que se ordena descendientemente, de posterior a anterior. Para ordenar alfabéticamente basta cambiar el operador > por <, e inicializar la lista con una cadena de caracteres de código ASCII menor que cualquiera de los caracteres de los nombres a ordenar.

Otros algoritmos más generales se pueden consultar en la bibliografía, quedando fuera de los objetivos de este libro. En el ejercicio resuelto 12.7 se opera con una lista de números enteros ordenada ascendientemente.

### SUPRESION DE ELEMENTOS

Consideraremos dos casos:

- Supresión del sucesor de NODO(P)
- Supresión del NODO(P)

En ambos casos utilizaremos un *puntero auxiliar* Q para liberar el nodo sobrante.

#### • Supresión del sucesor de NODO(P)

Este caso no plantea problemas. Apuntamos con Q al nodo a eliminar, reajustamos los enlaces, y liberamos la memoria ocupada por el nodo a suprimir, como se representa en la figura 12.21.

#### Algoritmo

```
(1) Q := SIG(P)
(2) SIG(P) := SIG(Q)
(3) Liberar (Q)
```

#### Codificación en Pascal

```
q := p^.sig;
p^.sig := q^.sig;
Dispose (q);
```

## ESTRUCTURAS DINAMICAS DE DATOS

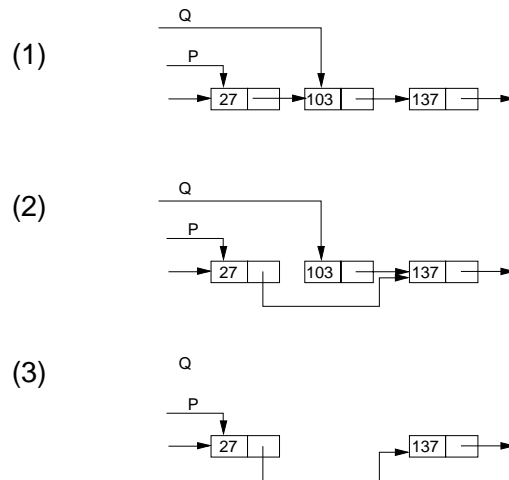


Figura 12.21 Supresión del sucesor de NODO (P)

### • *Supresion del NODO(P)*

En este segundo caso se plantea un problema similar al que nos encontramos en el método de inserción delante: no podemos retroceder al nodo precedente para modificar su *puntero* de enlace.

Para solucionarlo se recurre también a un truco: copiamos el nodo siguiente sobre el nodo actual, y a continuación eliminamos el nodo siguiente.

Gráficamente, el método usado se representa en la figura 12.22.

### Algoritmo

```
(1) Q := SIG (P)
(2) NODO (P) := NODO (Q)
(3) Liberar (Q)
```

### Codificación en Pascal

```
(1) q := p^.sig;
(2) p^ := q^;
(3) Dispose (q);
```

ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS

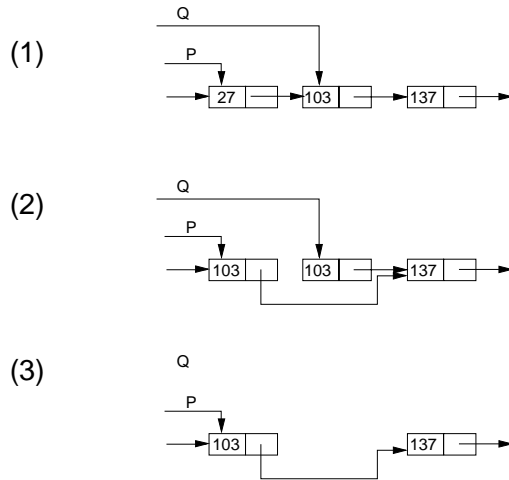


Figura 12.22 Supresión del NODO ( P )

Obsérvese que con este procedimiento *NO SE PUEDE SUPRIMIR EL ULTIMO NODO DE LA LISTA*. Para ello se necesita recorrer la lista con 2 punteros, ANTERIOR y ACTUAL, como se indica en la figura 12.23. Otra posibilidad es utilizar un procedimiento recursivo que se presenta más adelante.

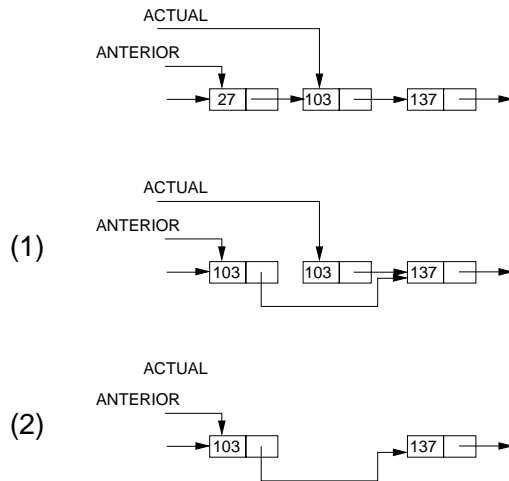


Figura 12.23 Supresión de nodos con dos punteros auxiliares

En este caso, la supresión de un nodo se haría con:

```
(1) SIG (ANTERIOR) := SIG (ACTUAL)
(2) Liberar (ACTUAL)
```

La codificación en Pascal es inmediata.

## 12.5 VISION RECURSIVA DE LA LISTA

Una lista es una estructura de datos de naturaleza *recursiva*, pues podemos definirla como:

- Lista vacía. ( puntero = *NIL* )
- Un elemento seguido de una lista.

Por tanto, la aplicación de *algoritmos recursivos* puede ser adecuada en muchos casos.

A modo meramente ilustrativo describiremos dos ejemplos sencillos, cuyo tratamiento sin utilizar la recursividad resultaría sumamente laborioso:

- Recorrido inverso de una lista.
- Supresión de un nodo de la lista.

### Ejemplo 12.13

Diseñaremos un procedimiento llamado *Imprim\_al\_reves*, para imprimir una lista en orden inverso.

#### Algoritmo recursivo

Sea P el *puntero* a la lista.

- CASO BASE: SI (lista vacía) ENTONCES no hacer nada.  
FIN\_SI;
- CASO GENERAL: Imprim\_al\_reves la lista apuntada por SIG(P)  
Después Imprimir INFO(P).

#### Codificación en Pascal

```
PROCEDURE ImprimAlReves (p: puntero);
{ Imprime en orden inverso la lista apuntada por p }
BEGIN
 IF p <> nil
 THEN
 BEGIN
 ImprimAlReves (p^.sig);
 Writeln (p^.info);
 END;
END; { ImprimAlReves }
```



### Ejemplo 12.14

Se quiere realizar un procedimiento *Suprimir*, que borre de la lista el nodo que contenga un determinado VALOR, si existe. Utilizaremos un procedimiento con dos parámetros:

- el puntero externo a la lista (P).
- el VALOR entero que se quiere suprimir, caso de que exista en la lista.

### Algoritmo

- CASO BASE: SI (lista vacía)  
 ENTONCES  
     no hacer nada  
 SI\_NO  
     SI INFO(P) = VALOR  
         ENTONCES  
             *Suprimir NODO(P)*;  
         FIN\_SI;  
 FIN\_SI;
- CASO GENERAL: *Suprimir* en la lista apuntada por SIG(P)

El subproblema *suprimir NODO(P)* consiste simplemente en:

```
Q := P
P := SIG (P)
Liberar (Q)
```

### Codificación en Pascal

```
PROCEDURE Suprimir (VAR p: puntero; valor: integer);
{ Suprime de la lista apuntada por p el nodo que }
{ contenga el valor, si existe. }

VAR
 q: puntero; { puntero auxiliar }

BEGIN
 IF p <> NIL
 THEN IF p^.info = valor { la lista no está vacía }
 THEN
 BEGIN { Suprimir el nodo p^ }
 q := p;
 p := p^.sig;
 Dispose (q);
 END
 ELSE Suprimir (p^.sig, valor);
END; { Suprimir }
```

No se plantean casos excepcionales para suprimir el primero o el último nodo de la lista.

En el ejercicio resuelto 12.2 se utilizan diversos algoritmos recursivos con una lista de números enteros leída de un fichero de texto.

## 12.6 OTROS TIPOS DE LISTAS: PILAS, COLAS, LISTAS CIRCULARES

Las listas simplemente enlazadas se pueden clasificar según la manera de introducir y retirar los nodos en ellas. Se estudiarán los tipos: pilas, colas, y listas circulares.

### • Pilas

Una caso particular de lista encadenada es la que se le denomina *pila*, *stack*, o almacenamiento *LIFO* (*Last In First Out*), porque el último elemento introducido queda el primero de la lista, según se explicó en el apartado *Creación de una lista*, después del algoritmo general de inserción a la cabecera.

Es decir una pila es una estructura en la que se introducen y retiran los elementos por un sólo extremo.

Se pueden mostrar varios ejemplos de estructura de *pila* en casos reales:

- pilas de platos
- estuches de monedas
- vías muertas de ferrocarril

La operación de agregar un nodo a la lista se llama *Meter* (*push*), y su algoritmo es el explicado en la sección 4.1, *Creación de una lista*. La operación de retirar un nodo se llama *Sacar* (*pop*), y su algoritmo se obtiene a partir del utilizado en el apartado *Supresión del NODO (P)*, considerando el caso particular de suprimir el primer nodo de la lista. En el ejercicio resuelto 12.12 se incluyen los códigos en Pascal de *Meter* y *Sacar*.

Otros ejemplos de utilización de estructuras de tipo *pila*, se pueden consultar en los ejercicios resueltos 12.1, 12.3, 12.4, 12.5 y 12.6.

### • Colas

Una *cola* es una lista *FIFO* (*First In, First Out*: primero en entrar, primero en salir), en la cual los elementos se añaden por un extremo (el final) y se eliminan por el otro (el frente). Puede implementarse mediante una lista simplemente enlazada con dos punteros externos, uno para cada nodo de los extremos. Esto facilita la operación de insertar elementos por el final de la *cola*. Ambos punteros pueden unirse en un registro. Podemos construir una cola en lenguaje Pascal mediante las siguientes declaraciones:

```

TYPE puntero = ^nodo;
nodo = RECORD
 info: tipoInfo; {puede ser un tipo simple o estructurado}
 sig : puntero;
END;
tipoCola = RECORD
 frente, final: puntero;
END;

VAR p: tipoCola

```

## OTROS TIPOS DE LISTAS: PILAS, COLAS, LISTAS CIRCULARES

La figura 12.24 es una representación gráfica de una estructura de tipo `tipoCola`. Los valores  $x_1, x_2, x_3, \dots, x_{n-1}, x_n$ , son datos de tipo `tipoInfo`. Además de las declaraciones anteriores, para que una lista enlazada sea considerada una *cola* la inserción de elementos debe realizarse detrás del nodo señalado por el puntero `p^.final`, y el nodo a eliminar deberá ser el señalado por `p^.frente`. El ejercicio 12.10 simula mediante una cola una lista de espera de pacientes de un médico.

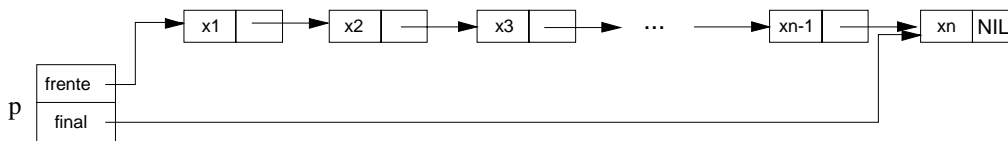


Figura 12.24 Ejemplo de *cola*

### • Listas circulares

Una *lista circular* se caracteriza porque el último elemento no señala a *NIL*, sino al primer elemento de la lista, según se indica en la figura 12.25.

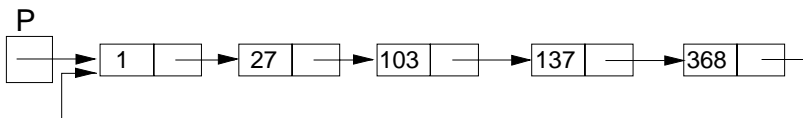


Figura 12.25 Lista circular

Si llamamos *cabeza* al puntero externo a la lista, y *p* a un puntero auxiliar para recorrerla, en este caso la condición de *fin de lista* no es `p=NIL`, sino `p=cabeza`. Las declaraciones necesarias para manejar una lista circular son idénticas a las utilizadas con listas lineales.

Se puede convertir una lista lineal en circular, cambiando el campo de enlace del último elemento, haciendo que apunte al primero.

En el ejercicio resuelto 12.11 se utiliza una lista circular simplemente enlazada, y en el 12.16 se simula el juego de la ruleta mediante una lista circular doblemente enlazada. Se explica en qué consiste una lista doblemente enlazada en la siguiente sección.

## 12.7 ESTRUCTURAS DINAMICAS DE DATOS NO LINEALES

Se llaman estructuras dinámicas de datos *no lineales* a aquellas que tienen más de un enlace por nodo. El estudio exhaustivo de este tipo de estructuras queda fuera de los objetivos de esta obra. Las estructuras de este tipo más usadas son las *listas doblemente enlazadas*, *grafos* y *árboles*.

### Listas doblemente enlazadas

En algunas aplicaciones se utilizan listas lineales *doblemente enlazadas*, en las cuales cada nodo tiene dos enlaces: uno apunta al elemento siguiente (para el último nodo este enlace apunta a *NIL*) y otro al elemento anterior (para el primer nodo este enlace apunta a *NIL*). En la figura 12.26 se representa una lista lineal doblemente enlazada, cuyos elementos son registros con tres campos: un campo de información ( de tipo cadena de caracteres), y dos campos de tipo puntero, según se acaba de explicar. Como ejemplo de utilización de este tipo de listas, pueden consultarse los ejercicios resueltos 12.16 y 12.17.

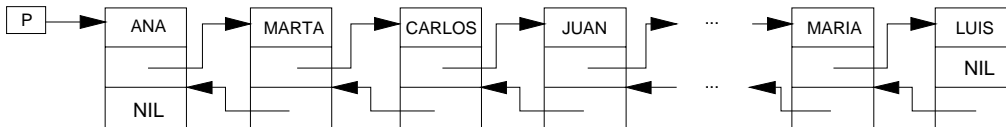


Figura 12.26 Lista doblemente enlazada

### Grafos

Un *grafo* o *gráfica dirigida* es una estructura matemática compuesta por una serie de puntos, llamados *vértices* o *nodos*, unidos por líneas llamadas *aristas*, *lados*, o *arcos*. En Programación los *vértices* suelen ser *registros* enlazados mediante *punteros*. Cada registro puede tener varios campos de tipo puntero que señalan a otros registros. Es decir, los vértices o nodos están representados por registros, y los lados por punteros. En la figura 12.27 se representa un ejemplo sencillo de *grafo*.

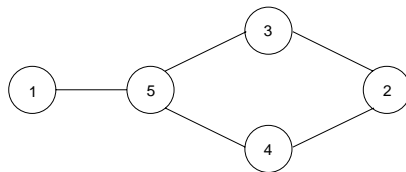


Figura 12.27 Ejemplo de grafo

## ESTRUCTURAS DINAMICAS DE DATOS NO LINEALES

Los grafos se utilizan en Informática, Matemáticas, Ingeniería y muchas otras ciencias, cuando es necesario representar relaciones arbitrarias entre datos.

En Pascal, podemos construir un grafo mediante la siguientes declaraciones:

```
TYPE
 arco = ^vertice;
 vertice = RECORD
 info: tipoInfo;
 a1, a2, a3, a4, a5: arco;
 END;
VAR
 grafo: arco;
```

Un *árbol*, estructura estudiada a continuación, puede considerarse una clase particular de *grafo*, que cumple dos propiedades:

- Las subestructuras enlazadas con cualquier nodo están desarticuladas.
- Existe un nodo, llamado *raíz*, desde el cual puede alcanzarse cada nodo del árbol recorriendo un número finito de lados.

### Arboles

La estructura dinámica no lineal más utilizada es el *árbol*. A continuación se introducen el concepto de *árbol* y los algoritmos de tratamiento de *árboles binarios*, los árboles más sencillos, sin profundizar en el estudio de estructuras complejas.

Se puede definir un *árbol* como una estructura dinámica de datos en la cual, cada nodo o elemento puede tener varios campos de tipo *puntero*, es decir, varios descendientes colgando de él. Piénsese por ejemplo en un árbol genealógico: si nos fijamos en un nodo genérico del mismo, veremos que depende directamente de otro nodo al que denominamos *padre* de aquel. A su vez, puede tener varios nodos descendientes directos de él denominados *hijos*. Por analogía con los árboles genealógicos, esta misma terminología se aplica a los árboles utilizados en Informática. También, y por analogía con un árbol real invertido, el primer nodo de un árbol se denomina *nodo raíz*, y los últimos (es decir, aquellos que no tienen ningún hijo) se denominan *hojas* del árbol.

Los árboles que acabamos de describir se denominan *árboles generales*, ya que cada nodo puede tener un número arbitrario de hijos. Su estudio queda fuera del alcance de este libro. Nos centraremos en un tipo especial de árboles denominados *árboles binarios*, en los cuales cada nodo puede tener como máximo dos hijos, denominados *hijo izquierdo* e *hijo derecho*. En la figura 12.28 se representa gráficamente un árbol binario. La anatomía de cada uno de estos nodos se representa por el siguiente esquema:



Se accede al *nodo raíz* del árbol a través de un *puntero* externo.

Un ejemplo de declaración en Pascal para manejar un árbol puede ser:

```

TYPE
 arbol = ^nodo;
 nodo = RECORD
 info:integer; (* u otro tipo *)
 izq, der: arbol;
 END;
VAR
 p:arbol;

```

Los *árboles binarios* son de extraordinaria importancia en programación. De hecho, los árboles generales se implementan a base de árboles binarios y listas.

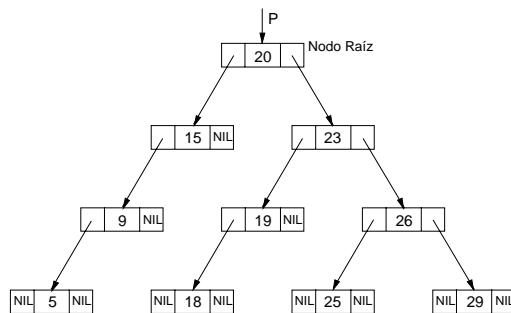


Figura 12.28 Ejemplo de árbol binario

### Arboles binarios de búsqueda

Se dice que un árbol binario es de búsqueda si, para cada nodo, se cumple que:

- La información almacenada en su subárbol *izquierdo* es *menor* que la suya propia.
- La información almacenada en su subárbol *derecho* es *mayor* que la suya propia.

La importancia de este tipo de árboles radica en la mayor rapidez con que se puede localizar una determinada información en el mismo. La construcción del árbol de acuerdo con las reglas anteriores, hace que se minimice el número de comparaciones a efectuar en el proceso de búsqueda.

### Algoritmos de tratamiento de árboles binarios

Un árbol es, por su propia definición, una estructura de datos de naturaleza *recursiva*, ya que los descendientes de cada nodo tienen a su vez estructura de árbol. Por lo tanto se utilizarán *algoritmos recursivos* para su tratamiento.

## ESTRUCTURAS DINAMICAS DE DATOS NO LINEALES

### • Creación de árboles

Supongamos que `valor` es una variable entera que contendrá los sucesivos valores que se van a insertar en el árbol. El algoritmo recursivo a utilizar sería:

NIVEL 0

- CASO BASE:      SI (árbol vacío)  
                  ENTONCES  
                  *Insertar valor en árbol*  
                  FIN\_SI;
  
- CASO GENERAL: SI `valor < valor-del-nodo`  
                  ENTONCES  
                  *Insertar valor a la izquierda*  
                  {en el subárbol izquierdo}  
                  SI\_NO  
                  SI `valor > valor-del-nodo`  
                  ENTONCES  
                  *Insertar valor a la derecha*  
                  {en el subárbol derecho}  
                  SI\_NO {elemento duplicado}  
                  no hacer nada.  
                  FIN\_SI;  
                  FIN\_SI;

NIVEL 1

```
Insertar valor en árbol:
INICIO
 Crear(p);
 Info(p) := valor;
 Izq(p) := NIL;
 Der(p) := NIL;
FIN
```

En el caso particular de que los valores a insertar ya estén ordenados, el árbol binario *degenera* en una lista simplemente enlazada.

Para la traducción a Pascal del algoritmo asumiremos las definiciones de tipo presentadas anteriormente.

### Codificación en Pascal

```
PROCEDURE Insertar (valor:integer; VAR p:arbol);
{Creación de un árbol binario de búsqueda}
BEGIN
IF p=NIL {caso base}
THEN
 BEGIN {Insertamos}
 New(p)
 p^.info:= valor;
 p^.izq := NIL;
 p^.der := NIL;
```

## ESTRUCTURAS DINAMICAS DE DATOS

```
END
ELSE
 IF valor < p^.info
 THEN Insertar (valor, p^.izq)
 ELSE
 IF valor > p^.info
 THEN Insertar (valor, p^.der)
 { ELSE valor duplicado: no hacer nada }
END; {Insertar}
```

Obsérvese que el *puntero* al árbol, *p*, debe transmitirse por dirección ya que de otro modo no se produciría la inserción en el árbol.

### • Recorrido de un árbol

Existen tres formas diferentes de recorrer un árbol binario: *preorden*, *inorden*, y *postorden*. Sus nombres provienen del momento en el cual se visita cada nodo para procesar su información. Veámoslo:

#### • *Preorden*:

- 1.-Procesar nodo
- 2.-Recorrer subárbol izquierdo
- 3.-Recorrer subárbol derecho

#### • *Inorden*:

- 1.-Recorrer subárbol izquierdo
- 2.-Procesar nodo
- 3.-Recorrer subárbol derecho

#### • *Postorden*:

- 1.-Recorrer subárbol izquierdo
- 2.-Recorrer subárbol derecho.
- 3.-Procesar nodo.

Obsérvese que en los tres casos siempre se recorre antes el subárbol izquierdo que el derecho.

Como ejemplo, si queremos escribir el contenido de un árbol en *inorden*, el procedimiento en Pascal sería:

```
PROCEDURE Recorrer (p:arbol);
{Escribe el árbol en inorden}
BEGIN
 IF arbol <> NIL
 THEN
 BEGIN
 Recorrer (p^.izq);
 Write (p^.info);
 Recorrer (p^.der);
 END;
END; {Recorrer}
```



## EXTENSIONES DEL COMPILADOR TURBO PASCAL

Para escribirlo en *preorden* o *postorden* bastaría con cambiar de posición la sentencia *Write*, según los esquemas anteriores.

### • Búsqueda en un árbol

Vamos a diseñar una función que nos permita encontrar un determinado valor en el *árbol*. Queremos que dicha función nos devuelva un *puntero* al nodo que contienen el valor buscado, caso de que exista en el árbol; si el valor buscado no se encuentra en el árbol, debe devolver el valor *NIL*.

- CASO BASE: SI árbol-vacío  
    ENTONCES devolver NIL  
    SI\_NO  
    SI valor-buscado = valor-nodo  
    ENTONCES Devolver puntero a ese nodo.  
    FIN\_SI;  
FIN\_SI;
- CASO GENERAL: SI valor-buscado < valor-nodo  
    ENTONCES *Buscar a la izq.*  
    SI\_NO *Buscar a la der.*  
FIN\_SI;

### Codificación en Pascal

```
FUNCTION Buscar (valor:integer; p:arbol):arbol;
{Busca el valor en el árbol}
BEGIN
 IF p=NIL
 THEN Buscar:=NIL
 ELSE
 IF valor = p^.info
 THEN Buscar:=p
 ELSE
 IF valor < p^.info
 THEN Buscar := Buscar(valor,p^.izq)
 ELSE Buscar := Buscar(valor,p^.der);
END; {Buscar}
```

En el ejercicio resuelto 12.19, se muestra un ejemplo de utilización de árboles binarios.

## 12.8 EXTENSIONES DEL COMPILADOR TURBO PASCAL

### OTROS TIPOS DE PUNTEROS

En Turbo Pascal existe el tipo predefinido *pointer*, para referirse a *punteros* que no apuntan a variables de un tipo determinado, es decir son punteros *genéricos*. Como la constante *NIL*, los valores de este tipo son compatibles con cualquier tipo *puntero*.

Otro tipo puntero especial de Turbo Pascal es el tipo *pChar*. Se utiliza generalmente apuntando a un string terminado en caracter nulo (que se estudia más adelante en esta misma sección). Una variable de tipo *pChar* es un puntero a un caracter. La definición del tipo *pChar* está en la unit *System*, y es la siguiente:

```
TYPE
 pChar = ^char;
```

## OPERADOR @

En Pascal estandar para referirnos a la variable referenciada por un *puntero* se antepone el signo  $\uparrow$  al identificador de la variable puntero. En muchos compiladores comerciales se permite usar el signo  $\wedge$  o el signo @. En Turbo Pascal el signo @ tiene otro significado, es un operador unario.

En Turbo Pascal, @ es un operador unario, que *aplicado a una variable* devuelve su dirección, es decir devuelve un puntero que apunta a la dirección de memoria donde está almacenada dicha variable. El tipo del puntero devuelto depende del estado de la directiva de compilación *\$T*:

{*\$T-*} Si no está activa *{*\$T-*}* (caso por defecto), el tipo de resultado es un puntero genérico *pointer*, es decir es un puntero sin tipo, que es compatible con otros tipos de punteros.

{*\$T+*} Si está activa *{*\$T+*}* el tipo de resultado es un puntero que señala al tipo de la variable a la cual se ha aplicado el operador @.

El operador @ también puede aplicarse al *nombre de una función, un procedimiento o un método*<sup>21</sup> devolviendo en este caso un puntero genérico *pointer*, que contiene la dirección al punto de comienzo de dicho subprograma. El tipo de resultado en este caso es independiente del estado de la directiva de compilación *\$T*.

El operador @ ya se aplicó en el ejemplo 11.12 del capítulo once, para el uso de los procedimientos *bloquea* y *desbloquea* de ficheros en redes, que tenían como parámetro un puntero genérico a una variable de tipo fichero.

### Ejemplo 12.15

Partiendo de las declaraciones:

```
TYPE pareja = ARRAY[0..1] OF char;
VAR entero: integer;
 punPareja: ^pareja;
```

Mediante la sentencia:

```
punPareja := @ entero;
```

---

<sup>21</sup> Los métodos son los subprogramas definidos dentro de un tipo objeto. Se estudian en el capítulo 13.

## EXTENSIONES DEL COMPILADOR TURBO PASCAL

conseguimos que `punPareja` apunte a `entero`. Nos podemos referir al valor almacenado en la variable `entero` mediante `punPareja^`.

## OTROS SUBPROGRAMAS PARA MANEJO DE MEMORIA DINAMICA

Además de los procedimientos estandar *New* y *Dispose*, Turbo Pascal incorpora los siguientes procedimientos y funciones:

- *PROCEDURE GetMem*(*VAR p:pointer; tamagno:word*);. Procedimiento que asigna un bloque de memoria de tamaño dado en bytes (*tamagno*) a una variable referenciada  $p^{\wedge}$  por un puntero genérico *p* de tipo *pointer*. Si no hay espacio libre en la memoria *heap*<sup>22</sup> se produce un error en tiempo de ejecución, para prevenirlo usar las funciones *MemAvail* y *MaxAvail*, que se explican a continuación. Para determinar el *tamagno* se suele usar la función estándar *SizeOf*. Este es el método habitual para reservar memoria dinámica para los punteros genéricos *pointer* y los punteros *pChar*.
- *PROCEDURE FreeMem*(*VAR p:pointer; tamagno:word*);. Si *p* es una variable de cualquier tipo puntero, a la cual se le ha asignado memoria mediante el procedimiento *GetMem*, entonces *FreeMem* libera la memoria ocupada por una variable referenciada dinámica  $p^{\wedge}$  de un tamaño dado en bytes (*tamagno*).
- *FUNCTION MemAvail:LongInt*. Devuelve la suma de los tamaños de todos los bloques libres de la memoria *heap*. Téngase en cuenta que un bloque contiguo de almacenamiento con el tamaño de toda la memoria *heap* es poco probable que exista, dado que las sucesivas llamadas a los procedimientos de asignación y liberación de memoria dejan huecos en la memoria *heap*, produciendo su *fragmentación*. Para determinar el bloque libre de mayor tamaño en la memoria *heap*, usar la función *MaxAvail*, que se explica a continuación. En el apartado 12.9 de este capítulo se explica el mecanismo de asignación y liberación de bloques de memoria *heap* en Turbo Pascal.
- *FUNCTION MaxAvail:LongInt*. Devuelve el tamaño del mayor bloque contiguo libre de la memoria *heap*, indicando el tamaño de la mayor variable referenciada dinámica que puede ser asignada usando *New* o *GetMem*.

Los procedimientos *FreeMem* y *GetMem* se utilizan con *strings* terminados en caracter nulo, que se estudiarán en esta misma sección. En el capítulo 13 pueden consultarse varios ejemplos en que se utilizan con punteros genéricos de tipo *pointer*. Respecto a la utilización de las funciones *MaxAvail* y *MemAvail*, puede consultarse el ejercicio resuelto 12.18.

---

<sup>22</sup> La memoria *heap* o montón (traducción literal al castellano) es la parte de la memoria dejada por el compilador para su gestión en tiempo de ejecución. Su funcionamiento para el compilador Turbo Pascal se explica con profundidad en el apartado 12.9 de este capítulo.

**OTRAS FUNCIONES DE MANEJO DE PUNTEROS Y DIRECCIONES**

- *FUNCTION Addr (x):pointer*. Devuelve la dirección como puntero genérico del elemento especificado *x*. Donde *x* es cualquier variable, o nombre de procedimiento o función.
- *FUNCTION Assigned (VAR p): boolean*. Comprueba si un puntero genérico *pointer* o variable procedural *p* vale *NIL*.
- *FUNCTION Cseg:word*. Devuelve el valor actual del registro CS. El resultado de tipo *word* es la dirección de segmento del segmento de código dentro del cual se llamó a *Cseg*.
- *FUNCTION Dseg:word*. Devuelve el valor actual del registro DS. El resultado de tipo *word* es la dirección de segmento del segmento de código dentro del cual se llamó a *Dseg*.
- *FUNCTION Ofs(x):word*. Devuelve el desplazamiento (*offset*) del elemento especificado *x*. Donde *x* es cualquier variable, o nombre de procedimiento o función.
- *FUNCTION Ptr(Seg, Ofs:word): pointer*. Convierte un segmento base y una dirección de desplazamiento (*offset*) en un valor de tipo puntero.
- *FUNCTION Seg(x):word*. Devuelve el segmento de la dirección de un elemento especificado *x*. Donde *x* es cualquier variable, o nombre de procedimiento o función.
- *FUNCTION SPtr:word*. Devuelve el valor actual del registro SP (puntero a *stack*).
- *FUNCTION SSeg:word*. Devuelve el valor actual del registro SS (segmento de *stack*).

**CONVERSIONES DE TIPO**

En Turbo Pascal es posible cambiar el tipo del resultado de una expresión, para poder asignárselo a una variable del tipo deseado. En particular, esta característica se utiliza para poder convertir punteros sin tipo (*pointer*) a punteros del tipo que nos interese en un momento dado. La sintaxis de la conversión en notación EBNF es:

```
<conversión tipo expresión> ::= <identificador de tipo> (<expresión>)
```

El tipo de <expresión> y el <identificador de tipo> deben ser ambos tipos ordinales o tipos puntero. Si el tamaño del tipo especificado es diferente del de la expresión, el valor original puede ser truncado o extendido, conservándose siempre el signo del valor. Debe tenerse mucho **cuidado cuando se producen truncamientos**, dado que los resultados pueden ser inesperados.

En particular, <expresión> puede ser una referencia a una variable, y la conversión de tipos nos sirve en este caso para poder hacer una asignación a una variable de otro tipo. Esta conversión de tipos sigue las pautas del lenguaje C.

## EXTENSIONES DEL COMPILADOR TURBO PASCAL

Las conversiones de tipo se utilizará en los capítulos siguientes para adaptar tipos abstractos de datos genéricos a estructuras de datos concretas. En particular para realizar conversiones de tipo cuando las variables referenciadas por punteros genéricos *pointer* se usen con tipos de datos concretos. Véase apartado *genericidad* en el capítulo trece.

### Ejemplo 12.16

Suponiendo las siguientes declaraciones:

```
TYPE
 puntero = ^real;
VAR
 p1: pointer;
 p2: puntero;
```

podríamos hacer la siguiente conversión de tipos:

```
p2 := puntero(p1);
```

Después de esta sentencia, la variable `p2` contendrá la misma dirección de memoria que `p1`. Nótese que no se puede hacer la asignación directa `p2 := p1`, por ser las variables de distintos tipos punteros.

## CADENAS TERMINADAS EN CARACTER NULO

Son un tipo especial de cadenas de caracteres de utilidad especial en ciertos casos, gracias a la sintaxis extendida de Turbo Pascal, y a los procedimientos y funciones incorporados en la *unit Strings*.

Se diferencian de los *strings* estudiados en el capítulo ocho en que no tienen el byte de longitud en la posición cero de la cadena, y que consisten en una secuencia de caracteres no nulos acabada de un caracter nulo (carácter 0 de la tabla ASCII, también representado por *NULL* o #0). Turbo Pascal no pone límites a su longitud, pero la arquitectura de 16 bits del sistema operativo DOS impone un límite de 65.535 caracteres.

Turbo Pascal incorpora la *unit Strings* con las funciones siguientes para el manejo de *cadena terminadas en caracter nulo*:

- *FUNCTION StrNew(s:pChar):pChar*. Asigna espacio para una cadena en la memoria *heap*.
- *FUNCTION StrDispose(s:pChar)*. Libera la memoria *heap* previamente asignada a una cadena.
- *FUNCTION StrCat(dest,fuent:pChar):pChar*. Añade una cadena fuente al final de una cadena destino y devuelve un puntero a la cadena destino.
- *FUNCTION StrComp(s1,s2:pChar):integer*. Compara dos cadenas, *s1* y *s2*, devolviendo un valor menor que cero si  $s1 < s2$ , cero si  $s1 = s2$ , y mayor que cero si  $s1 > s2$ .

## ESTRUCTURAS DINAMICAS DE DATOS

- *FUNCTION StrCopy(dest,fuent:pChar):pChar*. Copia una cadena fuente en una cadena destino y devuelve un puntero a la cadena destino.
- *FUNCTION StrECopy(dest,fuent:pChar):pChar*. Copia una cadena fuente en una cadena destino y devuelve un puntero al final de la cadena destino.
- *FUNCTION StrEnd(s:pChar):pChar*. Devuelve un puntero al final de una cadena (un puntero al caracter nulo final).
- *FUNCTION StrIComp(s1,s2:pChar): integer*. Compara dos cadenas sin distinguir mayúsculas de minúsculas.
- *FUNCTION StrLCat(dest,fuente:pChar; lonMax:word):pChar*. Añade una cadena fuente al final de una cadena destino, comprobando que la longitud de la cadena resultante no excede de un máximo *lonMax*, y devuelve un puntero a la cadena destino.
- *FUNCTION StrLComp(s1,s2:pChar;lonMax:word):integer*. Compara dos cadenas hasta una longitud máxima determinada.
- *FUNCTION StrLCopy(dest, fuente:pChar; lonMax:word): pChar*. Copia, hasta un número determinado de caracteres, una cadena fuente en una cadena destino y devuelve un puntero a la cadena destino.
- *FUNCTION StrLen(s:pChar):word*. Devuelve la longitud de una cadena, sin contar el carácter nulo.
- *FUNCTION StrLIComp(s1,s2:pChar;lonMax:word):integer*. Compara dos cadenas hasta una longitud máxima determinada, sin distinguir mayúsculas de minúsculas.
- *FUNCTION StrLower(s:pChar):pChar*. Convierte una cadena a minúsculas y devuelve un puntero a la cadena.
- *FUNCTION StrMove(dest,fuente:pChar;n:word):pChar*. Mueve un bloque de *n* caracteres desde una cadena fuente a una cadena destino y devuelve un puntero a la cadena destino. Las dos cadenas pueden solaparse.
- *FUNCTION StrPas(s:pChar):STRING*. Convierte una cadena terminado en caracter nulo en una cadena de tipo string.
- *FUNCTION StrPCopy(dest:pChar; fuente:STRING):pChar*. Copia una cadena de tipo string en una cadena terminada en caracter nulo, y devuelve un puntero a esta última.
- *FUNCTION StrPos(s1,s2:pChar):pChar*. Devuelve un puntero a la primera aparición de una subcadena *s2* dentro de la cadena *s1*, o *NIL* si la subcadena no está en la cadena.

## EXTENSIONES DEL COMPILADOR TURBO PASCAL

- *FUNCTION StrRScan(s:pChar;c:char):pChar*. Devuelve un puntero a la última aparición de un caracter *c* dentro de una cadena *s*, o *NIL* si el caracter no se encuentra en la cadena.
- *FUNCTION StrScan(s:pChar;c:char):pChar*. Devuelve un puntero a la primera aparición de un caracter *c* dentro de una cadena *s*, o *NIL* si el caracter no se encuentra en la cadena.
- *FUNCTION StrUpper(s:pChar):pChar*. Convierte una cadena a mayúsculas y devuelve un puntero a la cadena.

### Uso de cadenas terminadas en caracter nulo

Los cadenas terminadas en nulo se almacenan como *arrays de caracteres con base 0*, que son de la forma:

```
ARRAY [0..n] OF char;
```

siendo *n* un número entero positivo distinto de cero. La principal diferencia con el tipo *string* de Turbo Pascal, como ya se ha citado, es que no tienen el byte de longitud en la posición cero. El final de la cadena se marca almacenando el caracter nulo (representado por *NULL* o *#0*) en la posición siguiente a la última utilizada. Se manipulan mediante punteros, con un conjunto de *reglas de sintaxis extendida*. Para activar la sintaxis extendida, hay que usar la directiva de compilación *\$X+*.

### Compatibilidad pChar / literales string

Con la sintaxis extendida activada, una variable de tipo *pChar* es compatible con un literal string (expresión constante de tipo string). Como consecuencia, están permitidas las siguientes operaciones:

- Asignación de un literal string a una variable de tipo *pChar*.
- Paso de un literal string como argumento para un parámetro de tipo *pChar*.
- Inicialización de constantes con tipo de tipo *pChar* con literales string. Se puede aplicar también a tipos estructurados, como *arrays* cuyos elementos sean de tipo *pChar* o *registros* con campos *pChar*.

### Ejemplo 12.17

Veamos con un ejemplo estas tres operaciones derivadas de la compatibilidad de literales string con variables *pChar*.

## ESTRUCTURAS DINAMICAS DE DATOS

```
CONST
 literal: ARRAY[0..20] OF char = 'Esto es un ejemplo'#0;
{4} aviso: pChar = 'CERRADO POR REFORMA';
{5} diaSemana: ARRAY[0..6] OF pChar = ('domingo', 'lunes', 'martes',
 'miércoles', 'jueves', 'viernes',
 'sábado');

VAR
 p, q: pChar;
 ...

PROCEDURE ImprCadena(cad: pChar);
 ...

END; (* ImprCadena *)

BEGIN
{1} p := 'Esto es un ejemplo';
{2} q := @literal;
{3} ImprCadena('Esto es otro ejemplo');
 ...
```

El efecto de estas sentencias es el siguiente:

- En la dirección de memoria apuntada por *p* se almacena una copia del literal asignado con la sentencia {1}. El efecto de esta sentencia es similar al de la asignación al puntero *q* (sentencia {2}).
- Tras la llamada al procedimiento *ImprCadena* (sentencia {3}), el parámetro *cad* contiene una dirección de memoria en la que se almacena una copia del literal usado como argumento.
- El identificador *aviso* representa una constante con tipo de tipo *pChar*, que se inicializa con un literal string en la sentencia {4}. Por último, *diaSemana* representa a una constante con tipo, de tipo estructurado *array*, cuyos elementos son de tipo *pChar* y se inicializan con literales string en la sentencia {5}.

### Ejemplo 12.18

```
CONST prueba = 'Esto es otro ejemplo';
VAR
 p: pChar;

BEGIN
 p := prueba;
{1} Writeln(p);
{2} WHILE p^<>#0 DO
 BEGIN
 Write(p^);
 p := p+1;
 END;
END.
```



## EXTENSIONES DEL COMPILADOR TURBO PASCAL

En este ejemplo, la sentencia {1} y el bucle WHILE {2} hacen exactamente lo mismo, escribir en pantalla la cadena 'Esto es otro ejemplo'.

### Compatibilidad pChar / arrays de caracteres con base 0

Con la sintaxis extendida activada, una variable de tipo *pChar* es compatible con un array de caracteres con base 0. Es decir, donde se espera *pChar* se puede usar en su lugar un array de caracteres con base 0. El compilador convierte el array de caracteres en una constante puntero, que contiene la dirección de memoria del primer elemento del array.

#### Ejemplo 12.19

```
VAR
 a: ARRAY[0..50] OF char;
 p: pChar;
 ...
BEGIN
 ...
 p := a;
 ImprCadena(p);
 ImprCadena(a);
```

La sentencia `p := a` es correcta, debido a la compatibilidad de tipos descrita. Por la misma causa, son válidas ambas llamadas al procedimiento *ImprCadena*, declarado en el ejemplo 12.17, y su efecto es el mismo.

Una constante con tipo, de tipo array de caracteres con base 0, puede inicializarse con un literal string de longitud menor que el número de elementos del array. En los elementos sobrantes se almacena el carácter *NULL* (#0), con lo cual el array contiene un string terminado en carácter nulo. El ejemplo 12.20 aclara esta situación:

#### Ejemplo 12.20

```
TYPE
 TnombreFich = ARRAY[0..79] OF char;
CONST
 NomBufFic: TnombreFich = 'PRUEBA.PAS';
 NomPtrFic: pChar = NomBufFic;
```

La constante con tipo `NomBufFic`, de tipo `TnombreFich`, es un array de caracteres con base 0. Los elementos de índices 11 a 79 contienen el carácter *NULL* (#0). Puede utilizarse como una cadena terminada en carácter nulo, y ser asignado a valores de tipo *pChar*, como la constante `NomPtrFic`.

## Indexado de punteros a carácter

Como consecuencia de la compatibilidad `pChar` / arrays de caracteres con base 0 se puede indexar un puntero a carácter, como si fuera un array de caracteres con base 0. Los subíndices, aplicados a `pChar` tienen un significado especial, ya que representan un desplazamiento (*offset*) que se aplica a la dirección de memoria que contiene la variable de tipo `pChar`.

### Ejemplo 12.21

```
VAR
 cadena: ARRAY[0..79] OF char;
 p: pChar;
 ch: char;
 ...
BEGIN
 p := cadena;
 ch := cadena[5];
 ch := p[5];
 ...
```

Las dos últimas sentencias son equivalentes, y asignan a `ch` el valor del sexto elemento del array `cadena`.

En el ejemplo 12.21 `p[0]` equivale a `p^`, y contiene la dirección de memoria del primer carácter. `p[1]` apunta al carácter siguiente, etc. En cuanto a indexación, el tipo `pChar` equivale a la siguiente declaración:

```
TYPE
 arrayCar = ARRAY[0..65535] OF char;
 pChar = ^arrayCar;
```

Cuando se indexa un puntero a carácter, hay que tener en cuenta que el compilador no hace comprobaciones de rango, siendo el programa el que debe hacerlas.

## Cadenas terminadas en carácter nulo y procedimientos estándar

Con la sintaxis extendida activada, los procedimientos estándar `Read`, `Readln`, `Str` y `Val` pueden utilizarse con arrays de caracteres con base 0. Además, se pueden usar los procedimientos estándar `Write`, `Writeln`, `Val`, `Assign` y `Rename` con arrays de caracteres con base 0 y con punteros a carácter (`pChar`), según se muestra en el siguiente ejemplo.

### Ejemplo 12.22

```
PROGRAM PruebaPChar(input, output);
```

## EXTENSIONES DEL COMPILADOR TURBO PASCAL

```
TYPE
 cadena = ARRAY[0..80] OF char;
VAR
 c: cadena;
 p,q: pChar;
 x: real;
 cod: integer;

BEGIN
 Write('Introduzca un número con varias cifras...');
{1} Readln(c);
 p := c;
{2} Writeln('Convertiremos la cadena ', p, ' a un valor numérico: ');
{3} Val(p, x, cod);
 IF cod <> 0
 THEN
 Writeln('Error en la posición: ', cod)
 ELSE
 Writeln('Valor en formato exponencial = ', x);
 Readln;
END.
```

Con la sentencia {1} leemos directamente el array de caracteres *c*, operación no permitida en Pascal estándar. La sentencia {2} escribe la cadena apuntada por *p*, de tipo *pChar*. En la sentencia {3} utilizamos *p*, de tipo *pChar*, como si fuese de tipo *string*, y convertimos la cadena apuntada por *p* a un valor real, almacenado en *x*.

En resumen, se han estudiado cuatro tipos de estructuras de datos para el manejo de *cadena de caracteres*, cada una de ellas con ventajas e inconvenientes respecto a las demás:

- *Arrays de caracteres*. Utilizan memoria estática. No tienen tope de longitud.
- *Arrays empaquetados de caracteres*. Utilizan memoria estática. No tienen tope de longitud. Es la estructura incorporada por el lenguaje Pascal estándar para manejo de *cadena de caracteres*. El lenguaje permite con ellos ciertas operaciones prohibidas con los arrays de caracteres sin empaquetar, por ejemplo: asignación, lectura y escritura con una sola sentencia. Pero el empaquetamiento supone algunas restricciones (sus elementos no pueden utilizarse como parámetros actuales a un procedimiento o función).
- *Strings*. Utilizan memoria estática. Su tope de longitud es de 255 caracteres en Turbo Pascal. En las aplicaciones vistas hasta ahora esta era la estructura cuya utilización resultaba más ventajosa. En la programación del entorno Windows no se permite el uso del tipo *string*, para el manejo de las funciones API (*Applications Programming Interface*) de Windows.
- *Cadenas terminados en caracter nulo*. Pueden utilizar memoria estática (por medio de *arrays*) o dinámica (por medio del tipo *pChar*). No tienen tope de longitud. En los capítulos siguientes se utilizarán en la programación con entornos como *Turbo Vision* o *Windows*. También se utilizan en ocasiones por compatibilidad con el lenguaje C. En concreto, son indispensables si se quiere utilizar la *Interfaz de Programación de Aplicaciones* de Windows (API), como se verá en el capítulo quince.

## ACCESO DIRECTO A POSICIONES DE MEMORIA

El manejo de posiciones de memoria directamente por los programas debe de tener en cuenta la existencia de dos tipos de expresiones:

- *Las expresiones reubicables* contienen valores que necesitan reubicación (*relocation*) en tiempo de enlace (*link*). Una expresión que contiene etiquetas, variables o subprogramas es siempre reubicable. La reubicación es el proceso por el cual el enlazador (*linker*) asigna direcciones absolutas a los distintos símbolos del lenguaje, ya que en tiempo de compilación, el compilador desconoce la dirección final que ocupará una etiqueta, variable o subprograma.
- *Las expresiones absolutas* contienen valores que no necesitan reubicación. Una expresión que sólo opera con constantes es absoluta.

Es posible declarar variables con una dirección específica de memoria, que en Turbo Pascal se denominan *variables absolutas*. La declaración de tales variables debe incluir detrás del nombre del tipo la cláusula *absolute* junto con la dirección en la cual va a residir la variable, especificada por el segmento y el desplazamiento (*offset*). Por ejemplo:

```
VAR
 ModoCrt : byte ABSOLUTE $0040:$0049;
```

La cláusula *absolute* no puede utilizarse en la programación de aplicaciones en el entorno Windows.

También se puede utilizar la cláusula *absolute* para declarar dos variables en la misma posición de memoria, aunque no es aconsejable dado que va en contra de todos los principios de la programación estructurada. En el ejemplo siguiente *cadena* y *longCadena* comparten la misma posición de memoria de comienzo. Dado que en las cadenas de tipo *string* se almacena su longitud en la primera posición, *lonCadena* contendrá siempre la longitud de *cadena*.

```
VAR
 cadena : STRING[25];
 longCadena : byte ABSOLUTE cadena;
```

Para acceder directamente a posiciones de memoria, Turbo Pascal dispone de tres arrays predefinidos llamados *Mem*, *MemW* y *MemL*. Los elementos de *Mem* son de tipo *byte*, los de *MemW* son de tipo *word*, y los de *MemL* son de tipo *longint*. Para acceder a los elementos de estos *arrays*, los subíndices se construyen de manera especial: cada subíndice se forma por dos expresiones de tipo *word*, separadas por dos puntos (:), que representan respectivamente la base y el desplazamiento del segmento de la posición de memoria a acceder. Ejemplo:

```
Mem[$0040:$0049] := 7;
data := MemW[Seg(v):Ofs(v)];
memLong := MemL[64:3*12];
```

## EXTENSIONES DEL COMPILADOR TURBO PASCAL

La primera sentencia almacena el valor 7 en la posición de memoria \$0040:\$0049. El efecto de la segunda sentencia es copiar el valor de tipo *word*, almacenado en los dos primeros bytes de la variable *v*, en la variable *data*. Por último, la tercera sentencia copia en la variable *memLong* el valor de tipo *longint* almacenado en la posición 64:12, que en hexadecimal es \$0040:\$000C.

### Ejemplo 12.23

Se presenta un programa que indica: si la tecla de bloquea mayúsculas está pulsada o no; y si el teclado numérico está activo o no. Las instrucciones alternativas están dirigidas por una expresión booleana que trabaja a nivel de bits, véase en este mismo epígrafe el subapartado posterior al actual titulado *Operadores lógicos de manejo de bits*.

```
PROGRAM Teclas (Output);
VAR
 teclado:word;
BEGIN
 teclado:=Mem[$0040:$0017];
 IF (teclado AND 64) <> 0
 THEN Writeln ('Mayúsculas activas')
 ELSE Writeln ('Mayúsculas NO activas');
 IF (teclado AND 32) <> 0
 THEN Writeln ('Teclado numérico activo')
 ELSE Writeln ('Teclado numérico desactivado');
END.
```

## ACCESO DIRECTO A LOS PUERTOS

Para acceder a los puertos de datos de la CPU 80x86, el compilador Turbo Pascal implementa dos *pseudoarrays* predefinidos: *Port* y *PortW*. Ambos se comportan como arrays unidimensionales, teniendo un índice de tipo *word* que se corresponde con la dirección de un puerto hardware de entrada/salida. Los componentes de *Port* son *bytes*, y los de *PortW* son *words*. Por ejemplo si se desea escribir el dato de tipo *byte* almacenado en la variable *DatoByte* en la puerta serie COM1, se realiza la siguiente asignación:

```
Port[$3F8] := DatoByte;
```

Si se desea leer de COM1, se realizaría la asignación al revés:

```
DatoByte := Port[$3F8];
```

De igual forma se haría con *PortW*, pero con variables de tipo *word*.

*Port* y *PortW* no son realmente *arrays*, por lo tanto su uso está restringido sólo a asignaciones y referencias en expresiones.

## OPERADORES LOGICOS DE MANEJO DE BITS

Realizan operaciones lógicas con los bits de un dato de tipo *integer*. Los operadores se resumen en la tabla 12.1.

## ESTRUCTURAS DINAMICAS DE DATOS

| <b>Operador</b> | <b>Operación</b>              |
|-----------------|-------------------------------|
| NOT             | negación bit a bit            |
| AND             | conjunción Y, bit a bit       |
| OR              | unión O, bit a bit            |
| XOR             | O exclusivo bit a bit         |
| Shl             | desplazamiento a la izquierda |
| Shr             | desplazamiento a la derecha   |

Tabla 12.1 Operadores lógicos de bits

Estos operadores son binarios, excepto el operador *NOT*, que es unario. Para todos ellos los operandos tienen que ser de tipo *integer*, y el tipo del resultado también es *integer*. Los operadores lógicos se aplican a los bits de los operandos, uno a uno, de acuerdo con las tablas de verdad 12.2, 12.3, 12.4 y 12.5. En el ejemplo 11.12 del capítulo once, y en el ejemplo 12.23 del capítulo actual, ya han sido utilizados operadores lógicos de *bits*.

| <b>NOT</b> |   |
|------------|---|
| 0          | 1 |
| 1          | 0 |

Tabla 12.2 Tabla de verdad del operador NOT bit a bit

| <b>AND</b> | 0 | 1 |
|------------|---|---|
| 0          | 0 | 0 |
| 1          | 0 | 1 |

Tabla 12.3 Tabla de verdad del operador AND bit a bit

| <b>OR</b> | 0 | 1 |
|-----------|---|---|
| 0         | 0 | 1 |
| 1         | 1 | 1 |

Tabla 12.4 Tabla de verdad del operador OR bit a bit

## EXTENSIONES DEL COMPILADOR TURBO PASCAL

| <b>XOR</b> | 0 | 1 |
|------------|---|---|
| 0          | 0 | 1 |
| 1          | 1 | 0 |

Tabla 12.5 Tabla de verdad del operador XOR bit a bit

### Ejemplo 12.24

Suponiendo de tipo *integer* las variables *m* y *n*, a continuación se ilustra el funcionamiento de estos operadores. Representaremos el valor de los datos en base 2.

```
m 0000000011011011
NOT m 1111111100100100
n 0000000000000000
m AND n 0000000000000000
m OR n 0000000011011011
m XOR n 0000000011011011
```

### Ejemplo 12.25

Las operaciones de desplazamiento (*Shl* y *Shr*) trasladan el contenido de cada bit del primer operando hacia la izquierda o la derecha, el número de bits indicados por el segundo operando, como se observa a continuación. Los bits entrantes a la izquierda o derecha respectivamente se rellenan con ceros.

```
m 0000000011011011
m Shl 1 0000000110110110
m Shr 1 000000001101101
```

En este ejemplo el segundo operando es una constante, pero puede ser también una variable de tipo entero.

Los operadores de desplazamiento a izquierda y derecha pueden utilizarse respectivamente para realizar operaciones de multiplicación y división por potencias de 2.

### Ejemplo 12.26

Veamos como se utilizarían estos operadores dentro de un programa.

```
PROGRAM EjemploBits (input, output);
VAR
 c: integer;
 (*****
 PROCEDURE ImprBits(v: integer);
 (* Escribe la representación de los bits del parámetro v *)
 VAR
 i, mascara, tamagno: integer;
 BEGIN
 tamagno := 8 * SizeOf(integer);
 mascara := 1;
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

mascara := mascara shl (tamagno - 1);
FOR i := 1 TO tamagno DO
 BEGIN
 IF (v AND mascara = 0) THEN Write('0')
 ELSE Write('1');
 v := v shl 1;
 END;
 Writeln;
END;
(*****
BEGIN
 Write('Introduzca un entero c: ');
 Readln(c);
 Writeln('c en base 10: ', c);
 Write('c en base 2: ');
 ImprBits(c);
 Write('c AND c en base 2: ');
 ImprBits(c AND c);
 Write('c OR c en base 2: ');
 ImprBits(c OR c);
 Write('c XOR c en base 2: ');
 ImprBits(c XOR c);
 Write('NOT c en base 2: ');
 ImprBits(NOT c);
 Write('c Shl 1 en base 2: ');
 ImprBits(c Shl 1);
 Write('c Shr 1 en base 2: ');
 ImprBits(c Shr 1);
 Write('Pulsa <Intro> para volver al editor...');
 Readln;
END.

```

Este programa representa en pantalla el resultado de los operadores lógicos de bits, mediante el procedimiento *ImprBits*. A su vez este procedimiento también utiliza dichos operadores para acceder y representar el contenido de cada bit del parámetro *v*.

### 12.9 GESTION DE MEMORIA DINAMICA EN TURBO PASCAL

Se ha hablado al principio de este capítulo de dos modos de asignación de memoria: asignación estática (en tiempo de compilación) y dinámica (en tiempo de ejecución).

Las variables *globales* declaradas en el programa principal, fuera de todos los procedimientos y funciones residen en el *segmento de datos*, y tienen *asignación estática de memoria*.

Las variables *locales* declaradas dentro de los procedimientos y funciones residen en el *segmento stack*, y tienen asignación de *memoria dinámica stack*, denominada así porque tiene una estructura de tipo *stack* (*pila* o lista *LIFO*) y porque se realiza en tiempo de ejecución. Cuando se llama a un procedimiento o función en tiempo de ejecución se asignan direcciones de memoria en el *stack* para sus parámetros por valor y sus variables y constantes locales. Al finalizar la ejecución del subprograma se libera el espacio anteriormente ocupado en el *stack*. Si el subprograma es recursivo los valores de cada llamada se almacenan también en el *stack*. La directiva de compilación */\$S+* comprueba si se ha desbordado el *stack* al comienzo de cada procedimiento o función. EN el compilador Turbo Pascal se define el tamaño máximo del segmento *stack* con la directiva de compilación *\$M*.



## GESTION DE MEMORIA DINAMICA EN TURBO PASCAL

Se utiliza la *asignación dinámica heap* para las variables referenciadas mediante punteros, tal y como hemos estudiado en este capítulo. Las variables estáticas se crean mediante una *declaración*. Las dinámicas se crean mediante una llamada a los procedimientos *New* o *GetMem*. La *memoria dinámica heap* es la parte de la memoria reservada a las variables dinámicas, y se llama *Heap* (montón). También es una estructura de tipo pila, pero crece hacia arriba, en sentido contrario al *Stack*, según puede verse en la figura 12.29, que representa el mapa de ocupación de memoria por un programa creado con Turbo Pascal. Esta figura es muy similar a la figura 10.8 del capítulo diez, donde se utilizó para explicar el funcionamiento de los registros internos del microprocesador.

Los tamaños en bytes de los tres tipos de memoria se pueden definir mediante la directiva de compilación *\$M*, que define el tamaño de los tres segmentos (datos, stack y heap). Su forma de uso es la siguiente:

```
{ $M memoriaEstática, memoriaStack, memoriaHeap }
```

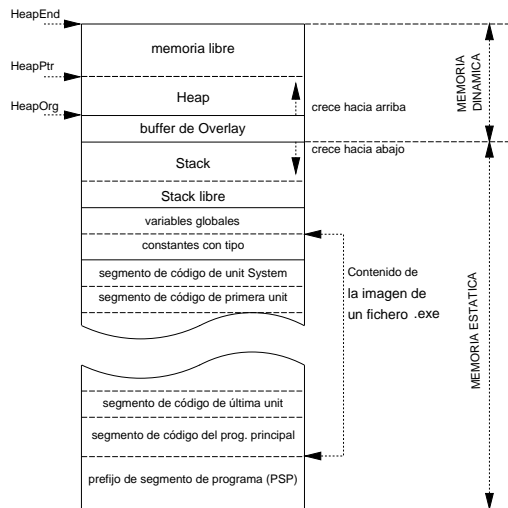


Figura 12.29 Mapa de memoria de Turbo Pascal

La memoria *heap* ocupa toda o parte de la memoria libre que deja un programa cuando se ejecuta. Por defecto su tamaño mínimo es de 0 bytes, y el máximo de 640 Kb; esto significa que, por defecto, la *heap* ocupa toda la memoria baja libre. Estos valores máximo y mínimo pueden cambiarse mediante la directiva de compilación *\$M*. Si no hay suficiente memoria libre (menos que el tamaño mínimo definido para la *heap*) el programa no se ejecuta.

## ESTRUCTURAS DINAMICAS DE DATOS

La dirección del principio del *Heap* se almacena en la variable *HeapOrg*, y la del final (que coincide con el principio de la memoria disponible) en la variable *HeapPtr*. Cada vez que se utilizan los procedimientos *New* o *GetMem*, se asigna espacio a una variable dinámica en el *Heap*, moviendo *HeapPtr* hacia arriba. Las variables dinámicas se van apilando unas encima de otras en el *Heap*.

Supongamos ya declaradas cuatro variables de tipo puntero: *p1*, *p2*, *p3*, y *p4*. Gráficamente, al ejecutarse las sentencias:

```
New(p1);
New(p2);
New(p3);
New(p4);
```

creamos cuatro variables dinámicas, cuyo contenido será almacenado en el *Heap*. El estado del *Heap* será el mostrado en la figura 12.30.

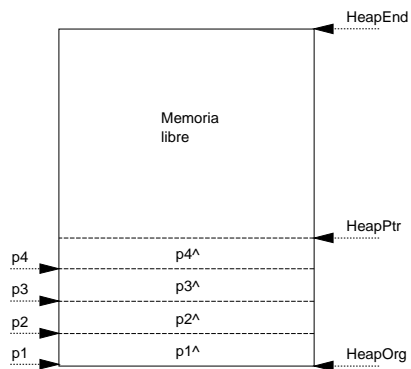


Figura 12.30 Almacenamiento de variables dinámicas en el Heap

Para dejar otra vez disponible la memoria ocupada cuando ya no es necesaria una variable dinámica, se utilizan los procedimientos *Dispose* y *FreeMem*. Por ejemplo, si ya no necesitamos *p4^*, ejecutaremos:

```
Dispose(p4);
```

y el estado resultante del *Heap* será el mostrado en la figura 12.31.

## GESTION DE MEMORIA DINAMICA EN TURBO PASCAL

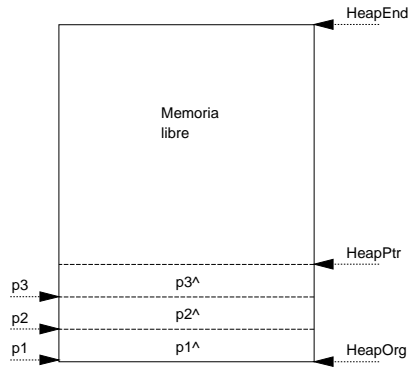


Figura 12.31 Liberación de variables dinámicas en el Heap

La variable *HeapPtr* se desplaza hacia abajo, dejando otra vez libre el espacio anteriormente reservado a  $p4^$ . Cabe la posibilidad de que las variables dinámicas no sean eliminadas en el orden en que han sido creadas. ¿Qué sucede si ahora destruimos la variable  $p2^$ ? Si ejecutamos la sentencia:

```
Dispose(p2);
```

el resultado es la aparición de un *agujero* en medio del *Heap*, situación representada en la figura 12.32.

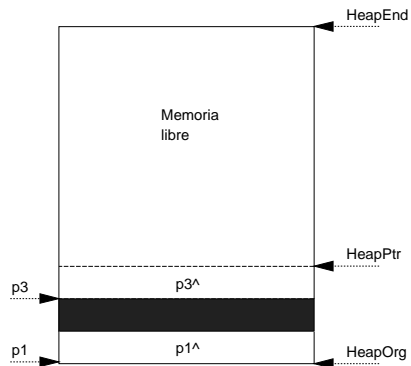


Figura 12.32 Aparición de un agujero en el Heap

Si en estas condiciones volvemos a crear la variable dinámica  $p4^$ , ejecutando de nuevo la sentencia:

```
New(p4);
```

la dirección de memoria contenida en  $p_4$ , y el espacio reservado a  $p_4^{\wedge}$  serán los mismos de antes. Aparentemente, estamos desaprovechando el espacio libre en memoria, al no reutilizar el bloque libre (agujero) dejado por  $p_2^{\wedge}$ .

En esta situación, si ahora se ejecuta la sentencia:

```
Dispose(p3);
```

inicialmente se crea un bloque libre mayor, pero a continuación la posición de *HeapPtr* baja hasta el principio de la memoria disponible, desapareciendo el agujero. El resultado puede verse en la figura 12.33.

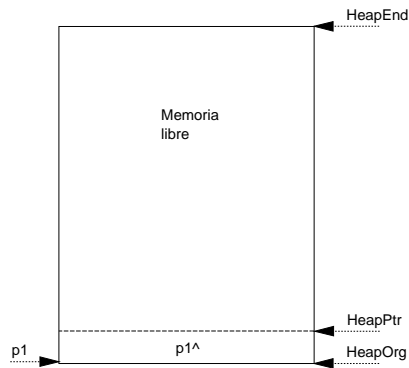


Figura 12.33 Aumento y liberación del bloque libre en el Heap

Las direcciones y tamaños de los bloques libres generados por las operaciones *Dispose* y *FreeMem* se guardan en una *lista de libres*. Cada vez que hay que asignar espacio a una variable dinámica se recorre primero la lista de libres, antes de desplazar *HeapPtr*. Si hay un bloque libre de tamaño suficiente es reutilizado, y no se mueve *HeapPtr*. La variable *FreeList*, de la *unit System* apunta al primer bloque libre del *Heap*. Este bloque contiene un puntero al siguiente bloque libre, el cual contiene un puntero al siguiente bloque libre, y así sucesivamente se forma la *lista de libres*. El puntero del último bloque libre no apunta a *NIL*, sino a la posición apuntada por *HeapPtr*. De esta manera, si la lista de libres está vacía, *FreeList* será igual a *HeapPtr*.

Cuando se hace una llamada a *New* o *GetMem* y no existe suficiente memoria disponible en el *Heap*, se produce un *error de ejecución*. Ocurre cuando no hay suficiente espacio entre *HeapPtr* y *HeapEnd* y a la vez no existe un bloque libre suficientemente grande. Siempre que se utilizan *New* y/o *GetMem*, se llama a una función de error del *Heap*, que detecta este problema y provoca un error de ejecución y el final del programa. El resultado de la llamada a esta función se almacena en la variable *HeapError*. Si sospechamos que puede agotarse el *Heap* durante la ejecución de un programa, podemos evitar el error de ejecución mediante una llamada a esta función de error, antes de la creación de cada nueva variable dinámica. También pueden utilizarse las funciones *MemAvail*

## EJERCICIOS RESUELTOS

y *MaxAvail*, descritas en la sección 12.8, *Extensiones del compilador Turbo Pascal*. En el ejercicio resuelto 12.18 se analiza la memoria dinámica disponible en el *Heap*, antes de introducir nuevos nodos en una lista simplemente enlazada, utilizando las funciones *MemAvail* y *MaxAvail*.

### 12.10 EJERCICIOS RESUELTOS

- 12.1** Realizar un programa que cree una lista encadenada de números enteros a partir de un fichero de texto. El programa debe de escribir la lista encadenada en orden inverso a su lectura.

#### Análisis

Con fines didácticos, se desarrolla el proceso gráficamente, describiendo paso a paso el efecto de cada acción ejecutada.

Dada una entrada de enteros 7, 4, 3, 15, 18, 9, ... , se trata de construir la lista encadenada de la figura 12.34.

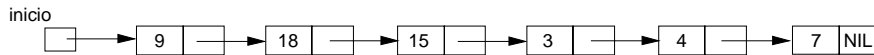


Figura 12.34 Lista encadenada de números enteros

En un principio la lista estará vacía, es decir:

```
inicio := NIL;
```

Con el procedimiento *New* se crea un nuevo componente de la lista:

```
New (p);
```

El efecto de esta sentencia es el mostrado en la figura 12.35.

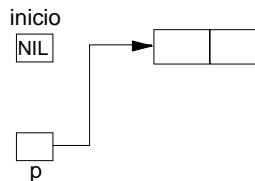


Figura 12.35 Creación del primer nodo

Con la sentencia:

```
Read (texto, p^.datos);
```

## ESTRUCTURAS DINAMICAS DE DATOS

se asigna un valor al campo `datos` de la variable referenciada `p`, por ejemplo el 4, quedando la lista según el esquema de la figura 12.36.

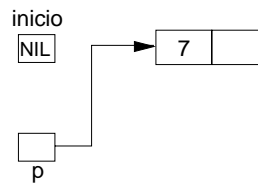


Figura 12.36 Introducción de datos en el nuevo nodo

Con la sentencia

```
p^.siguiente := inicio;
```

se asigna al campo `puntero` el valor de `inicio` que en este caso es `NIL`, según se muestra en la figura 12.37.

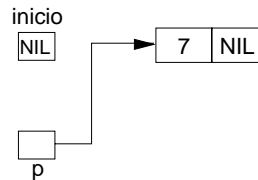


Figura 12.37 Reajuste de enlaces

Con la sentencia:

```
inicio := p;
```

Se coloca el primer elemento en la lista, como puede verse en la figura 12.38.

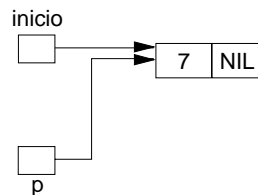


Figura 12.38 Inserción del primer nodo

## EJERCICIOS RESUELTOS

Para situar otro elemento en la lista, hay que crear un nuevo nodo auxiliar (ver figura 12.39):

```
New(p) ;
```

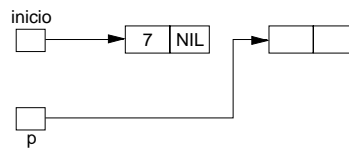


Figura 12.39 Otro nuevo nodo

Con las sentencias:

```
Read (texto, p^.datos) ;
p^.siguiente := inicio ;
```

se alcanza la situación de la figura 12.40.

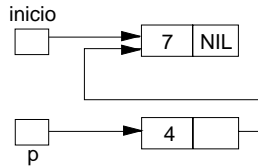


Figura 12.40 Reajuste de enlaces (2º nodo)

Con la sentencia:

```
inicio := p ;
```

se logra insertar el segundo nodo, llegando al estado de la figura 12.41.

## ESTRUCTURAS DINAMICAS DE DATOS

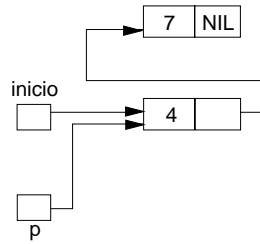


Figura 12.41 Inserción del segundo nodo

Realizando los pasos anteriores sucesivas veces se llega a una estructura como la representada en la figura 12.42.

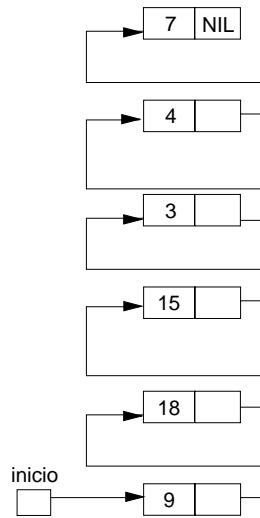


Figura 12.42 Lista creada

Esta estructura también se puede representar como en la figura 12.43.



## EJERCICIOS RESUELTOS

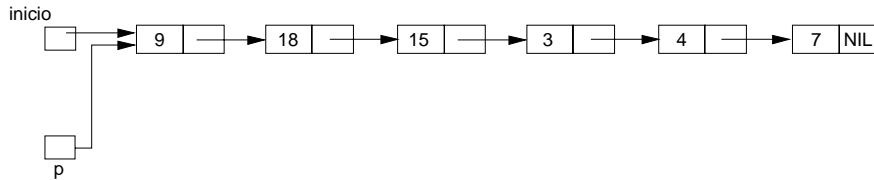


Figura 12.43 Otra representación de la lista creada

Para escribir los componentes de la lista, se usan las siguientes sentencias:

```
p := inicio;
```

Esta sentencia sitúa el puntero auxiliar de recorrido de la lista al principio de la misma (ver figura 12.44).

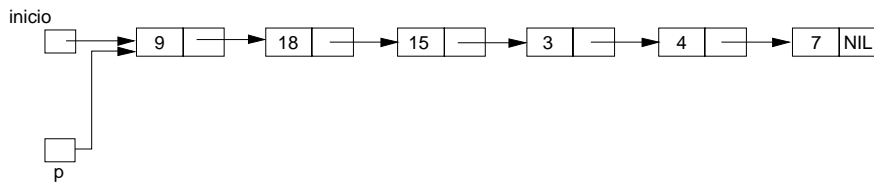


Figura 12.44 p se sitúa al principio de la lista

a continuación se escribe el campo `datos` del registro al que apunta `p`, y con la siguiente sentencia, se logra que `p` apunte al segundo nodo (ver figura 12.45):

```
p := p^.siguiente;
```

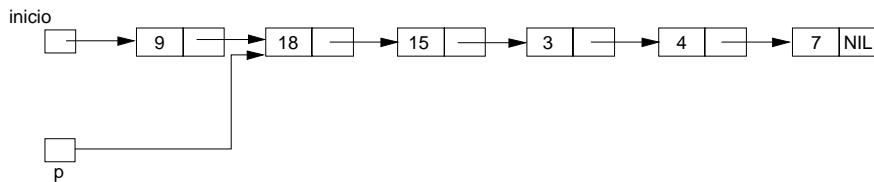


Figura 12.45 p avanza al siguiente nodo. Ejercicio 12.1

Así sucesivamente se escriben todos los elementos de la lista, hasta que `p` apunte a `NIL`, como en la figura 12.46.

## ESTRUCTURAS DINAMICAS DE DATOS

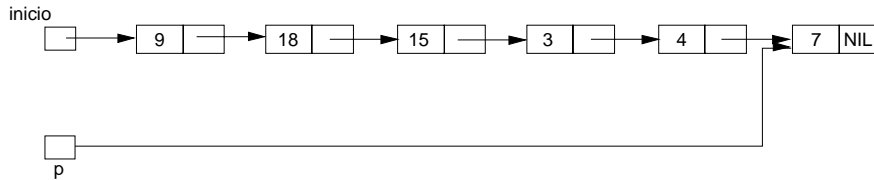


Figura 12.46 p ha llegado al final de la lista. Ejercicio 12.1

### Codificación en pascal

```

PROGRAM InvierteLista (texto,output);
(* Este programa crea una lista encadenada a partir de los datos
leídos de un fichero de texto y escribe dicha lista en orden
inverso a su lectura. *)
TYPE
 enlace=^elemento;
 elemento=RECORD
 siguiente:enlace;
 datos:integer
 END;
VAR
 inicio,p:enlace;
 texto:text;

BEGIN
 (* Inicializaciones *)
 Assign(texto,'INVIER.DAT');
 Reset(texto);

 (* Creación de la lista encadenada *)
 inicio:=NIL;
 WHILE NOT Eof(texto) DO
 BEGIN
 WHILE NOT Eoln(texto) DO
 BEGIN
 New(p);
 Read(texto,p^.datos);
 p^.siguiente:=inicio;
 inicio:=p
 END;
 Readln(texto);
 END;

 (* Escritura de la lista en orden inverso a su lectura *)
 Writeln('La lista invertida es la siguiente : ');
 p:=inicio;
 WHILE p<>NIL DO
 BEGIN
 Writeln;
 Write(p^.datos);
 p:=p^.siguiente
 END
 END.

```

## EJERCICIOS RESUELTOS

- 12.2** Hacer un programa que, utilizando procedimientos recursivos, cree una lista de enteros leídos de un fichero de texto, y a continuación los escriba en el mismo orden en que han sido introducidos en la lista, y en orden inverso. Observe que si se crea la lista añadiendo elementos a la cabecera, habrá que recorrer la lista en orden inverso para escribir los datos en el orden en que han sido introducidos. El programa permitirá además al usuario eliminar opcionalmente un nodo de la lista creada.

### Solución

Este ejercicio es un ejemplo de como utilizar los algoritmos presentados en la sección 12.5 de este capítulo, *Visión recursiva de una lista*.

```
PROGRAM ListaRecursiva (listaEnteros, output);
(* Ejemplo de utilización de procedimientos recursivos. Crea una
 lista enlazada a partir de los datos del fichero 'Copia.dat'.
 La escribe en el orden de lectura y en orden inverso. *)
TYPE
 enlace=^elemento;
 elemento=RECORD
 siguiente:enlace;
 datos:integer
 END;

VAR
 base:enlace;
 listaEnteros:text;
 ValorFuera : integer;
 opcion : char;

(*****)

PROCEDURE agnade (VAR p:enlace);
BEGIN
 IF p = NIL
 THEN
 BEGIN
 New(p);
 p^.siguiente:=NIL;
 Read(listaEnteros,p^.datos)
 END
 ELSE
 agnade(p^.siguiente) (* Llamada recursiva *)
 END;

(*****)

PROCEDURE escribeLista (p:enlace);
BEGIN
 IF p <> NIL
 THEN
 BEGIN
 Writeln(p^.datos);
 escribeLista(p^.siguiente) (* Llamada recursiva *)
 END
 END;

(*****)

PROCEDURE EscribeAlReves (p:enlace);
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

BEGIN
 IF p <> NIL
 THEN
 BEGIN
 EscribeAlReves (p^.siguiente);(* Llamada recursiva *)
 Writeln(p^.datos);
 END
 END;

(*****

PROCEDURE Suprimir (VAR p:enlace; valor:integer);
VAR
 q: enlace;
BEGIN
 IF p <> NIL
 THEN
 IF p^.datos = valor
 THEN
 BEGIN
 q := p;
 p := p^.siguiente;
 Dispose (q);
 END
 ELSE Suprimir (p^.siguiente, valor);
 END; (* Suprimir *)

(*****

(* Programa principal *)

BEGIN
 Assign(listaEnteros,'COPIA.DAT');
 Reset(listaEnteros);
 base:=NIL;
 (* Lee la lista del fichero de entrada *)

 WHILE NOT Eof(listaEnteros) DO
 BEGIN
 WHILE NOT Eoln(listaEnteros) DO
 agnade(base);
 Readln(listaEnteros)
 END;

(*****

 escribeLista(base);
 (* Escribe la lista en el mismo orden de entrada *)
 Writeln ('Pulse <Return> para continuar');
 Readln;
 Writeln ('La Lista invertida es:');
 EscribeAlReves(base);
 (* Escribe la lista en orden inverso al de entrada *)

(*****

 (* Supresión de un nodo de la lista *)
 REPEAT
 Write ('¿Desea suprimir algún nodo? (S/N) ');
 Readln (opcion);
 opcion := Uppcase(opcion)
 UNTIL opcion IN ['S','N'];
 WHILE opcion = 'S' DO
 BEGIN
 Write ('Introduzca el valor que desea suprimir : ');
 Readln (valorfuera);
 Suprimir (base, valorfuera);

```

## EJERCICIOS RESUELTOS

```
 EscribeLista (base);
 REPEAT
 Write ('¿Desea suprimir algún nodo? (S/N) ');
 Readln (opcion);
 opcion := Uppcase(opcion)
 UNTIL opcion IN ['S','N']
 END;
 Writeln ('Pulse <Return> para volver al Editor');
 Readln;
END.
```

- 12.3** Construir un programa que permita crear una lista cuyos componentes sean nombres de personas leídas de teclado (input). La entrada de datos finalizará con la palabra *fin*. Asimismo, deberá permitir añadir y borrar elementos de la misma una vez que esta haya sido creada. Todas estas operaciones deberán poder seleccionarse desde un menú.

### Solución

```
PROGRAM ListaEnlazada(input,output);
TYPE
 linea = PACKED ARRAY[1..40] OF char;
 enlace = ^personal;
 personal = RECORD
 nombre:linea;
 siguiente:enlace
 END;
VAR
 p,cabeza :enlace;
 opcion :1..4;
 cuenta:0..40; (* contador del array de caracteres *)
 nombreitem:linea;

 (*****)

FUNCTION Buscar(q:enlace;nomBuscado:linea):enlace;
(* Devuelve un puntero al elemento de campo nombre igual a
 nomBuscado, o NIL si no lo encuentra *)
VAR
 encontrado:boolean;
BEGIN
 encontrado:=FALSE;
 WHILE NOT (encontrado) AND (q<>NIL) DO
 IF q^.nombre = nomBuscado
 THEN encontrado:=true
 ELSE q:=q^.siguiente;
 Buscar := q;
END;

 (*****)

PROCEDURE LeerNombre(VAR nombre:linea);
VAR
 ch:char;
(* Este procedimiento lee un nombre del teclado *)
BEGIN
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

(* Llenado de blancos del array de caracteres *)
FOR cuenta:=1 TO 40
 DO nombre[cuenta]:=' ';
(* Puesta del contador a cero *)
cuenta:=0;
(* Lectura de un nombre por teclado *)
WHILE NOT Eoln DO
 BEGIN
 cuenta:=cuenta+1;
 Read(ch); (* No se pueden leer directamente los *)
 nombre[cuenta]:=ch; (* elementos de un array empaquetado *)
 (* de caracteres *)
 END;
 Readln;
 Writeln
END;

(*****

PROCEDURE visualizar(p:enlace);
(* Este procedimiento visualiza la lista completa *)
BEGIN
 Writeln;
 Write('Lista: ');
 IF p=NIL
 THEN Writeln(' vacía')
 ELSE Writeln;
 WHILE p<>NIL DO
 BEGIN
 Writeln(p^.nombre);
 p:=p^.siguiente
 END
 END;
END;

(*****

PROCEDURE crear(VAR cabeza:enlace);
(* Este procedimiento crea una lista encadenada, insertando
elementos en cabeza. Cuando se teclea FIN se deja de
introducir elementos. La palabra FIN no se inserta. *)
BEGIN
 cabeza:=NIL; (* Inicialización *)
 New(p); (* Creación nuevo nodo *)
 Write('Introduzca el nuevo elemento : ');
 leerNombre(p^.nombre); (* campo de información *)
 WHILE NOT ((p^.nombre[1] IN ['F','f']) AND
 (p^.nombre[2] IN ['I','i']) AND
 (p^.nombre[3] IN ['N','n'])) AND
 (p^.nombre[4] = ' ') DO
 BEGIN
 p^.siguiente:=cabeza; (* reajuste de enlaces *)
 cabeza:=p;
 New(p);
 Write('Introduzca el nuevo elemento: ');
 leerNombre(p^.nombre);
 END;
 visualizar(cabeza);
END;

(*****

PROCEDURE agnadir(VAR cabeza:enlace);
VAR
 p,q:enlace;

```

## EJERCICIOS RESUELTOS

```

BEGIN
 New(p);
 Writeln;
 Write('Introduzca nuevo elemento : ');
 leerNombre(p^.nombre);
 Write('Colocar detrás de: (pulsar <INTRO> si es el primero)');
 leerNombre(nombreitem);
 IF nombreitem[1]=' '
 THEN
 BEGIN (* Inserta al principio de la lista *)
 p^.siguiente:=cabeza;
 cabeza:=p
 END
 ELSE
 BEGIN
 q:=Buscar(cabeza,nombreitem);
 IF q <> NIL (* si no se encuentra el lugar, *)
 THEN (* no se realiza la inserción *)
 BEGIN
 p^.siguiente:=q^.siguiente; (* Caso general *)
 q^.siguiente:=p;
 END;
 END;
 visualizar(cabeza);
 END;

 (*****)

PROCEDURE suprimir(VAR p:enlace);
VAR
 q, r:enlace;
(* Este código se simplifica notablemente usando recursividad *)
BEGIN
 Writeln;
 Write('Introduzca el elemento a suprimir : ');
 leerNombre(nombreitem);
 q := Buscar(p,nombreitem); (* q apunta al elemento a suprimir *)
 IF q = NIL
 THEN
 Writeln('El elemento no está en la lista')
 ELSE
 IF q^.siguiente = NIL (* Hay que eliminar el último *)
 THEN
 IF q=p
 THEN
 p:=NIL (* Si era el único, la lista queda vacía *)
 ELSE (* si no, situamos otro puntero auxiliar *)
 BEGIN (* en el nodo anterior *)
 r:=p;
 WHILE r^.siguiente <> q DO
 r:= r^.siguiente;
 r^.siguiente := NIL;
 dispose(q);
 END
 ELSE (* Caso general. Truco: copiamos el siguiente *)
 BEGIN (* sobre el actual y eliminamos el siguiente *)
 r:=q^.siguiente;
 q^:=r^; (* Además del nombre se copia el campo enlace *)
 Dispose(r);
 END;
 visualizar(p);
 END;

 (*****)

PROCEDURE menu;

```

## ESTRUCTURAS DINAMICAS DE DATOS

```
BEGIN
 Writeln;
 Writeln('***** MENU PRINCIPAL *****');
 Writeln;
 Writeln(' 1 - Crear lista encadenada ');
 Writeln;
 Writeln(' 2 - Añadir un componente ');
 Writeln;
 Writeln(' 3 - Suprimir un componente ');
 Writeln;
 Writeln(' 4 - F I N ');
 Writeln;
 Write('Introduzca su opción : ');
 Readln(opcion);
 Writeln;
END;

(*****

(* Programa principal *)
BEGIN
 REPEAT
 menu;
 CASE opcion OF
 1 : crear(cabeza);
 2 : agnadir(cabeza);
 3 : suprimir(cabeza);
 4 :
 END
 UNTIL opcion=4
END.
```

- 12.4** Escribir un subprograma, para ser utilizado por el programa anterior, que saque una relación de todas las palabras de la lista que empiecen por vocal, por el fichero *output*.

### Solución

```
PROCEDURE ListaVocal(q:enlace);
BEGIN
 Writeln('Listado de nombres que empiezan por vocal:');
 WHILE q<>NIL DO
 BEGIN
 IF q^.nombre[1] IN ['a','e','i','o','u','A','E','I','O','U']
 THEN Writeln(q^.nombre);
 q:=q^.siguiente;
 END;
 END;
END;
```

Para utilizarlo, podríamos añadir una opción al menú, que deberá completarse con una etiqueta de la estructura multialternativa *CASE*. Al elegir esta nueva opción, deberá ejecutarse la llamada al procedimiento:

```
ListaVocal(cabeza);
```

- 12.5** En Informática una notación utilizada frecuentemente es la *RPN* (notación polaca inversa) o notación postfija, basada en la utilización de una *pila LIFO* (último en entrar, primero en salir) y en la que los operandos preceden al operador así:



## EJERCICIOS RESUELTOS

| <i>Notación algebraica</i> | <i>Notación RPN</i> |
|----------------------------|---------------------|
| a + b                      | a b +               |
| a - b                      | a b -               |
| a * b                      | a b *               |
| a / b                      | a b /               |

Se trata de hacer un programa que funcione como una calculadora RPN que sólo opera con números reales y con los operadores +, -, \*, / y C (borrado). Para lograrlo, lee un carácter y hace lo siguiente:

- Si es una **E** significa que después pedirá un número real, que será introducido en la pila.
- Si es un signo + sumar los dos últimos elementos introducidos en la pila, eliminarlos de la pila y meter el resultado en la última posición de la pila, y escribirlo en pantalla. Si la pila está vacía o sólo contiene un elemento, dar mensaje de error.
- Si es una **C** eliminar el último elemento introducido en la pila.
- Si es un -, \*, / actuar de la misma forma que con el signo +, pero con la resta, multiplicación y división.
- Si no es uno de estos operadores dar un mensaje de error.

*Nota:* La pila no tiene tope, y permite no usar paréntesis.

### Solución

```
PROGRAM Rpn (input,output);
TYPE
 puntero=^registro;
 registro=RECORD
 numero:real;
 siguiente:puntero;
 END;
VAR
 opcion:char;
 p,inicio:puntero;
 aux:real;

BEGIN
 inicio:=NIL;
 REPEAT
 REPEAT
 Write('Deme un símbolo:');
 Write(' E(entrada de número),+,-,*,/,C(borrado),F(fin): ');
 Readln(opcion)
 UNTIL opcion IN ['E','+','-','*','/','C','F'];
 Writeln;
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

CASE opcion OF
 'E': BEGIN
 New(p);
 Write('Introduzca un número :');
 Readln(p^.numero);
 p^.siguiente:=inicio;
 inicio:=p
 END;
 '+', '-', '/', '*': BEGIN
 p:=inicio;
 IF (p^.siguiente=NIL) OR (p=NIL)
 THEN
 Write('ERROR: No hay suficientes ');
 Writeln('elementos en la pila')
 ELSE
 BEGIN
 CASE opcion OF
 '+':aux:=(p^.numero)+(p^.siguiente^.numero);
 '-':aux:=(p^.siguiente^.numero) - (p^.numero);
 '*':aux:=(p^.numero)*(p^.siguiente^.numero);
 '/':aux:=(p^.siguiente^.numero) / (p^.numero)
 END; (* CASE *)
 Writeln(aux:9:2);
 p^.siguiente^.numero:=aux;
 inicio:=p^.siguiente;
 Dispose(p)
 END
 END;
 'C': BEGIN
 p:=inicio;
 IF p=NIL
 THEN
 Writeln('ERROR: pila vacia')
 ELSE
 BEGIN
 inicio:=p^.siguiente;
 Dispose(p)
 END
 END;
 END;
END (* Fin de la sentencia CASE *)
UNTIL opcion='F'
END.

```

- 12.6 a)** Escribir un programa que cree una lista encadenada de enteros (claves) leídos desde *input*. La entrada de datos finalizará al introducir el número cero.
- b)** Escribir una función para el programa anterior que nos dé como resultado el número de claves comprendidas entre 6 y 99.

### Solución

**a)** Programa que crea la lista pedida:

```

PROGRAM Lista (input, output);
CONST
 claveFinal=0;
TYPE
 puntero = ^elemento;
 elemento = RECORD
 clave:integer;

```

## EJERCICIOS RESUELTOS

```

 sucesor:puntero;
 END;
VAR
 cabecera,p:puntero; (* cabecera de la lista *)
 n:integer; (* p=puntero auxiliar *)

(* 1 *) (* ver apartado b *)

BEGIN (* Programa principal *)
 Write ('Introduzca clave (', claveFinal, ' para terminar:');
 Readln(n); (* Leemos la clave *)
 cabecera:=NIL; (* inicialmente:lista vacía *)
 WHILE n <>claveFinal DO
 BEGIN
 New(p); (* insertamos al principio de *)
 p^.clave:=n; (* la lista *)
 p^.sucesor:= cabecera;
 cabecera:=p;
 Write ('Introduzca clave (', claveFinal, ' para terminar:');
 Readln(n); (* leemos clave siguiente *)
 END;
 (* Visualización de la lista creada *)
 Writeln('La lista creada es:');
 p:=cabecera;
 WHILE p<>NIL DO
 BEGIN
 Writeln(p^.clave);
 p:=p^.sucesor;
 END;
 (* 2 *) (* ver apartado b *)

 Write('Pulse <INTRO> para volver al editor...');
 Readln;
END.

```

**b)** Utilizaremos una función, (a la que pasaremos como parámetro el puntero cabecera de lista) que nos devolverá el número de claves tales que:  $5 < \text{clave} < 100$ .

La función, simplemente, recorre la lista desde la cabecera hasta el final, incrementando un contador cuando la expresión `clave IN [6..99]` tome el valor *true*, lo que equivale a decir que:  $(\text{clave} > 5)$  y  $(\text{clave} < 100)$ .

```

FUNCTION Contar (p:puntero):integer;
VAR
 cont:integer;
BEGIN
 cont:=0;
 WHILE p<>NIL DO
 BEGIN
 IF (p^.clave>5) AND (p^.clave<100) THEN cont:=cont+1;
 p:= p^.sucesor;
 END;
 Contar:= cont;
END; (* Contar *)

```

## ESTRUCTURAS DINAMICAS DE DATOS

El código de esta función deberá ir en el lugar señalado con (\* 1 \*) en el programa principal del apartado a). Además, en el lugar señalado con (\* 2 \*) se podría introducir la sentencia:

```
WriteLn('Claves comprendidas entre 6 y 99 =', Contar(cabecera));
```

**12.7** Se tiene una lista encadenada ordenada según claves crecientes, cuyo primer elemento es el 1 y el último el 1000. Escribir un segmento de programa, con las declaraciones de tipos y variables necesarias, capaz de insertar claves leídas desde input.

El programa deberá comprobar que las nuevas claves introducidas se encuentren en el rango 1..1000, antes de proceder a su inserción en la lista. En el caso de que una clave ya exista en la lista, (clave repetida) no se insertará de nuevo.

### Solución

El programa utilizará los siguientes subprogramas:

*FUNCION ClaveValida*: comprueba si la clave está comprendida entre 1 y 1000, antes de insertarla.

*PROCEDIMIENTO InsertarClave*: recibe como parámetros la clave a insertar y la cabecera de la lista (*puntero*). Utiliza la siguiente función local.

*FUNCION Lugar*: recorre la lista ordenada hasta llegar al lugar donde debe insertarse la clave; devolviendo como resultado un *puntero* que señala al elemento de la lista, delante del cual debemos insertar la nueva clave.

Si la clave buscada ya existía en la lista, esta función devuelve el valor *NIL*.

Apoyándose en esta función, *InsertarClave* se limita a insertar el nuevo elemento mediante el método de inserción delante visto anteriormente; o bien no hace nada, si el valor devuelto por la función es *NIL* (pues ya existía esa clave).

*p*: es el puntero a la cabecera de la lista en el programa principal.

### Solución

```
PROGRAM ClavesOrdenadas (input, output);
TYPE
 puntero= ^elem;
 elem = RECORD
 clave:integer;
 enlace:puntero;
 END;
VAR
 p,aux:puntero; (* puntero a la cabecera de lista *)
 ch:char; (* en el programa principal *)
 i:integer; (* para leer claves desde teclado *)
```

## EJERCICIOS RESUELTOS

```

{-----}

FUNCTION ClaveValida (cv:integer):boolean;
BEGIN
 ClaveValida:= (cv >= 1) AND (cv <= 1000);
END;
(* Esta función también podía haberse hecho con una sentencia
 IF - THEN - ELSE.
 También, si nuestro ordenador admitiese conjuntos de hasta
 1000 elementos, o más, podríamos poner:
 ClaveValida:= cv IN [1..1000];
 pero en general, los compiladores usuales limitan el número
 de elementos de un conjunto a 256 ó menos *)

{-----}

PROCEDURE InsertarClave (cabecera:puntero; cv:integer);
VAR
 q, aux:puntero;

{-----}
FUNCTION Lugar (p:puntero; cv:integer):puntero;
BEGIN
 WHILE p^.clave < cv DO
 p:= p^.enlace; (* avanza al siguiente *)
 IF p^.clave=cv
 THEN Lugar:=NIL (* ya existe *)
 ELSE Lugar:=p;
 END; (* Lugar *)
{----- }

BEGIN (* InsertarClave *)
 q:= Lugar(cabecera, cv);
 IF q <> NIL
 THEN
 BEGIN
 New(aux); (* inserción delante *)
 aux^ :=q^;
 q^.clave:=cv;
 q^.enlace:=aux;
 END
 END; (*InsertarClave *)

{-----}

BEGIN (* Programa Principal *)
(* Aquí estaría el programa que creó la lista ya existente. Lo
sustituimos por un fragmento de código que crea una lista con dos
elementos (con claves 1 y 1000 para ajustarse al enunciado) *)

p:=NIL;
New(aux);
aux^.clave:=1000;
aux^.enlace:=p;
p:=aux;
New(aux);
aux^.clave:=1;
aux^.enlace:=p;
p:=aux;

REPEAT
 Write ('¿Quiere insertar otra clave? (s/n)');
 Readln(ch);
 IF (ch='s') OR (ch='S')
 THEN
 BEGIN
 Write('Teclee la clave:');

```

## ESTRUCTURAS DINAMICAS DE DATOS

```
 Readln(i);
 IF ClaveValida(i)
 THEN InsertarClave(p,i);
 END;
 (* Fragmento de código para visualizar la lista *)
 Writeln('Situación actual de la lista:');
 aux:=p;
 WHILE aux<>NIL DO
 BEGIN
 Writeln(aux^.clave);
 aux:=aux^.enlace;
 END;
 UNTIL ch IN ['n','N'];
END. (* Programa Principal *)
```

**12.8** Escribir un programa que lea claves enteras desde *input* y vaya formando una lista encadenada de registros cuya estructura sea:

```
RECORD
 clave:integer;
 contador:integer;
 enlace:puntero;
END;
```

Cada vez que se introduzca una nueva clave, se insertará un nuevo registro en la lista, inicializando el campo `contador` a 1. Si posteriormente se introdujese una clave ya existente en la lista, se incrementará el campo `contador` de la misma.

El proceso de inserción de claves finaliza al introducir la clave cero, que no se introducirá en la lista. En ese momento, el programa deberá listar las diferentes claves introducidas así como el número de veces que aparecieron.

*Nota:* El orden de las claves es irrelevante.

### Solución

*FUNCION buscar:* recorre la lista buscando una determinada clave. Si la encuentra, devuelve un *puntero* al elemento de la lista que contiene dicha clave; si no, llega hasta el final de la lista y devuelve el valor *NIL*.

*PROCEDIMIENTO listar:* imprime todas las claves que hay en la lista y el número de veces que aparecen.

*Programa Principal:* Realiza las siguientes tareas:

1º) Inicializa la lista a cero elementos:

```
cabecera:=NIL;
```

2º) Cada vez que lee una clave, mira si ya está en la lista (Utilizando la función *Buscar*). Si está, incrementa el `contador` de dicha clave. Si no, inserta la nueva clave al principio de la lista con su contador a 1.

## EJERCICIOS RESUELTOS

3º) Al recibir la clave 0, sale del bucle anterior y llama al procedimiento `listar`.

### Solución

```
PROGRAM ClavesRepetidas (input, output);
CONST
 ClaveFinal=0;
TYPE
 puntero= ^reg;
 reg = RECORD
 clave:integer;
 contador:integer;
 enlace:puntero;
 END;
VAR
 cabecera, p:puntero; (* p=puntero auxiliar *)
 n:integer; (* para leer claves *)

 (*****)

PROCEDURE Listar (q:puntero);
BEGIN
 WHILE q <> NIL DO
 BEGIN
 Writeln(q^.clave:10, q^.contador:10);
 q:=q^.enlace; (* avanza al siguiente *)
 END;
 END; (* Listar *)

 (*****)

FUNCTION Buscar (cve:integer; p:puntero):puntero;
BEGIN
 WHILE (p <> NIL) AND (p^.clave <> cve) DO
 p:=p^.enlace;
 Buscar:=p;
 END; (* Buscar *)

BEGIN (* Principal *)
 cabecera:= NIL; (*inicializa lista vacía *)
 Write ('Teclee clave (', claveFinal, ' para terminar:');
 Readln(n);
 WHILE n <> claveFinal DO
 BEGIN
 p:= Buscar(n,cabecera);
 IF p=NIL
 THEN
 BEGIN (* insertar al principio *)
 New(p);
 p^.clave:=n;
 p^.contador:=1;
 p^.enlace:=cabecera;
 cabecera:=p;
 END
 ELSE p^.contador:=p^.contador+1;
 Write ('Teclee clave (', claveFinal, ' para terminar:');
 Readln(n); (* Leer clave siguiente *)
 END; (* de While *)
 Writeln(' CLAVE VECES');
 Listar (cabecera);
 Readln; (* Para retener el listado en pantalla *)
 END. (* Principal *)
```

**12.9** Hacer una *FUNCION* para el programa anterior que devuelva *true* si la suma de todas las claves de la lista es mayor que 150, y *false* en caso contrario. Debe recibir como parámetro un *puntero* a la cabecera de la lista. Las claves son números enteros positivos.

### Solución

```

FUNCTION Mayor (p:puntero):boolean;
VAR
 suma:integer;

BEGIN
 suma:=0;
 WHILE (p <> NIL) AND (suma <= 150) DO
 BEGIN
 suma:=suma + p^.clave;
 p:=p^.enlace;
 END;
 Mayor:=(suma > 150);
END; (* Function Mayor *)

```

Para utilizar la función, antes del final del programa principal podríamos poner:

```

IF mayor(cabecera)
 THEN Writeln('La suma de las claves es mayor que 150')
 ELSE Writeln('La suma de las claves no es mayor que 150');

```

**12.10** Diseñar una estructura de tipo *cola* para el mantenimiento de la lista de espera de pacientes en la consulta de un médico. Cada nodo de la estructura contendrá simplemente el nombre del paciente y el campo de enlace. Construir un subprograma en Pascal para añadir elementos a la cola y otro para eliminarlos. Este último deberá devolver al punto de llamada, en una variable del tipo adecuado, el campo de información del nodo eliminado. Construir un programa para mantenimiento de la lista de espera, utilizando los subprogramas anteriores.

*Notas:* La cola puede estar vacía, y en cualquier caso debe ser utilizable después de las llamadas a ambos subprogramas. Elegir adecuadamente parámetros y variables locales.

En la sección 12.6 de este capítulo se explica en qué consiste una estructura de tipo *cola*, y en la figura 12.24 puede verse una representación gráfica general.

### Solución

```

PROGRAM COLA(input, output);
Uses crt;

```



## EJERCICIOS RESUELTOS

```

TYPE
 cadena = string[20];
 puntero = ^paciente;
 paciente = RECORD
 info: cadena;
 sig : puntero;
 END;
 tipoCola = RECORD
 frente,final: puntero;
 END;
VAR
 valor : cadena;
 c : tipoCola;
 p : puntero;
 opcion: char;
 salir : boolean;

{-----}

PROCEDURE Insertar(VAR c:tipoCola; valor:cadena);
VAR
 p:puntero;
BEGIN
 New(p);
 p^.info := valor;
 p^.sig :=NIL;
 IF c.frente=NIL (* ó c.final=NIL, cola vacía *)
 THEN c.frente:= p
 ELSE c.final^.sig:=p;
 c.final:=p;
END;

{-----}

PROCEDURE Eliminar(VAR c:tipoCola; VAR valor:cadena);
VAR
 p:puntero;
BEGIN
 IF c.frente = NIL (* cola vacía *)
 THEN Writeln('Cola vacía')
 ELSE
 BEGIN
 p:=c.frente;
 valor:=p^.info;
 c.frente:=c.frente^.sig;
 IF c.frente=NIL THEN c.final:=NIL;
 dispose(p);
 END;
END;

{-----}

PROCEDURE Escribir(c:tipoCola);
VAR
 p:puntero;
BEGIN
 p:=c.frente;
 Writeln('ESTADO ACTUAL DE LA COLA:');
 Write('FRENTE->');
 WHILE p<>NIL DO
 BEGIN
 Write(p^.info:10,'->');
 p:=p^.sig;
 END;
 Writeln('NIL');
 (* Comprobación del puntero final *)
 IF c.final<>NIL (* cola no vacía*)

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

 THEN Writeln('FINAL->',c.final^.info:10)
 ELSE Writeln('FINAL->NIL');
Write('Pulse <INTRO> para continuar...');
Readln;
END;

{-----}

BEGIN (* P.P. *)
ClrScr;
salir := false;
(* Inicializar cola *)
c.frente := NIL;
c.final := NIL;
REPEAT
Writeln('MANTENIMIENTO DE LA LISTA DE ESPERA:');
Writeln('1.- Añadir elemento');
Writeln('2.- Extraer elemento');
Writeln('3.- Salir');
Write('Elija opción...'); Readln(opcion);
CASE opcion OF
 '1': BEGIN
 Write('¿Nombre del paciente? ');
 Readln(valor);
 Insertar(c,valor);
 Escribir(c);
 END;
 '2': BEGIN
 Eliminar(c,valor);
 Writeln('EL ELEMENTO ELIMINADO ES: ',valor);
 Escribir(c);
 END;
 '3': salir:=true;
END; (* CASE *)
UNTIL salir;

END.
```

**12.11** Considere una *lista circular* simplemente enlazada, cuyos nodos son de la forma:

```

TYPE
 enlace = ^elemento;
 elemento = RECORD
 siguiente: enlace;
 datos: integer
 END;
```

Diseñe un procedimiento llamado por  $\text{Retroceder}(p, n, q)$ ; donde  $q$  es el *puntero* externo a la lista,  $p$  es un *puntero* que recorre la lista, y  $n$  una variable o expresión entera; que al ser llamado haga retroceder al *puntero*  $p$  el número de nodos indicado por  $n$ .

### Algoritmo

Una lista circular es aquella en que el último elemento apunta al primero, en lugar de apuntar a NIL, según puede observarse en la figura 12.47.

## EJERCICIOS RESUELTOS

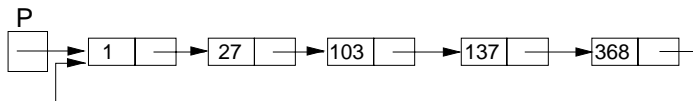


Figura 12.47 Lista circular

Siendo  $q$  el *puntero* externo a la lista y  $aux$  un *puntero* que la recorre, la condición de fin de lista en este caso no es  $aux = NIL$ , sino  $aux = q$ . Para crear una lista circular, cambiaremos el campo *siguiente* del último elemento, y le haremos apuntar al primer elemento.

Para diseñar el subprograma seguiremos los siguientes pasos:

- 1.- Contar el n° de nodos ( $m$ ) de la lista:  
     $m := 0$ ;  $aux := q$ ;  
    MIENTRAS  $aux^{sig} \neq q$  HACER  
         $m := m + 1$ ;  
         $aux := aux^{sig}$ ;  
    FIN\_MIENTRAS;
- 2.- Quitar a  $n$  las vueltas completas:  
    ( $n =$  n° de nodos a retroceder)  
     $n := n \text{ MOD } m$ ;
- 3.- Retroceder  $n =$  Avanzar  $m - n$ :  
    PARA  $i := 1$  HASTA  $m - n$  HACER  
         $p := p^{sig}$ ;  
    FIN\_PARA;

### Solución

```
PROCEDURE Retroceder (VAR p: enlace; q: enlace; n: integer);
VAR
 m, i: integer;
 aux: enlace;

BEGIN (* Retroceder *)
 m := 1;
 aux := q;

 WHILE aux^{siguiente} <> q DO
 (* Contamos el n° de nodos de la lista *)
 BEGIN
 aux := aux^{siguiente};
 m := m + 1;
 END;

 n := n MOD m;
 FOR i := 1 TO (m - n) DO p := p^{siguiente};

END; (* Retroceder *)
```

## ESTRUCTURAS DINAMICAS DE DATOS

A continuación se muestra con un fragmento de código un ejemplo de utilización de *Retroceder*. Se supone construido un subprograma *Escribir*, que incluye la visualización de los elementos de la lista apuntada por *inicio*, e indica el nodo de dicha lista apuntado por *p*.

```
Escribir (p, inicio);
REPEAT
 REPEAT
 Write ('Introduzca el número de nodos a retroceder (0 para
acabar): ');
 Readln (num)
 UNTIL num IN [0..100];
 Retroceder (p, inicio, num);
 Escribir (p, inicio);
 UNTIL num = 0;
```

- 12.12** Escribir un programa para crear una estructura de tipo pila de caracteres, leyendo una cadena de caracteres de teclado. A continuación deberá imprimirla en orden inverso. Utilizar subprogramas *Meter* y *Sacar* para introducir y extraer elementos de la pila.

### Solución

Este ejercicio ha sido utilizado para la construcción del TAD lista del ejercicio resuelto 12.20.

```
PROGRAM InvertirLinea(input,output);
(* Lee una línea de caracteres y la imprime en orden inverso *)
TYPE
 puntero = ^nodo;
 nodo = RECORD
 info:char;
 sig:puntero;
 END;
VAR
 caracter:char;
 pila:puntero;

(*****)

PROCEDURE Sacar(VAR pila:puntero;VAR componente:char);
{Elimina el primer elemento, suponiendo que la pila no está vacía}
{pila = puntero a la cabeza de la pila}
{componente = elemento quitado de la pila}
VAR
 p:puntero;
BEGIN
 p:=pila;
 componente:=pila^.info;
 pila:=pila^.sig;
 dispose(p);
END;

(*****)
```

## EJERCICIOS RESUELTOS

```
PROCEDURE Meter(VAR pila:puntero;VAR componente:char);
{Inserta un componente como primer elemento de la pila, que se
supone inicializada a NIL}
{pila = puntero a la cabeza de la pila}
{componente = elemento a insertar en la pila}
VAR
 p:puntero;
BEGIN
 new(p);
 p^.info:=componente;
 p^.sig:=pila;
 pila:=p;
END;

(*****)

PROCEDURE Inicializar(VAR pila:puntero);
BEGIN
 pila := NIL;
END;

(*****)

FUNCTION Vacia(Pila:puntero):boolean;
BEGIN
 Vacia:=(pila=NIL);
END;

(*****)

BEGIN
 Writeln('Introduzca una secuencia de caracteres...');
 Inicializar(pila);

 WHILE NOT Eoln DO {Lee y guarda caracteres}
 BEGIN
 read(caracter);
 Meter(pila,caracter);
 END;
 Readln;

 Writeln('La secuencia invertida es:');
 WHILE NOT Vacia(pila) DO {Imprime caracteres en orden inverso}
 BEGIN
 Sacar(pila,caracter);
 Write(caracter);
 END;
 Writeln;
 Write('Pulse <Intro> para acabar...');
 Readln;
END.
```

**12.13** Se dispone de un fichero de texto *precio.dat*, que contiene los precios de determinados productos. Así mismo, se dispone de otro fichero de texto *calidad.dat*, que contiene las calidades de esos productos. Precios y calidades están relacionados por una de las siguientes expresiones:

$$P = K \cdot C^2$$

$$P = K \cdot (C + C^2)$$

donde  $P$  es el precio,  $C$  la calidad, y  $K$  una constante de valor 150.

## ESTRUCTURAS DINAMICAS DE DATOS

Se pide desarrollar un programa que, a partir de la lectura de los ficheros anteriores, construya dos listas simplemente enlazadas, cuyos nodos contendrán dos campos claves, precio y calidad. En cada una de las listas las claves precio y calidad estarán relacionadas por una de las fórmulas anteriores.

*Nota:* Los dos ficheros no contienen necesariamente el mismo número de datos, y en el fichero de precios puede haber datos que no correspondan a ninguna calidad.

### Solución

```
PROGRAM PrecioCalidad(input, output, FichPrecio, FichCalidad);
TYPE
 puntero=^nodo;
 nodo=RECORD
 prec,cal:integer;
 sig:puntero;
 END;
VAR
 inicio1, inicio2, p:puntero;
 FichPrecio, FichCalidad:text;
 precio, calidad:integer;
 enc1, enc2:boolean;

(*****)

FUNCTION Precio1(c:integer):integer;
CONST
 k=150;
BEGIN
 Precio1:=k*Sqr(c);
END;

(*****)

FUNCTION Precio2(c:integer):integer;
CONST
 k=150;
BEGIN
 Precio2:=k*(c+Sqr(c));
END;

(*****)

PROCEDURE EscribeLista(p:puntero);
BEGIN
 Write('Inicio-> ');
 WHILE p<>NIL DO
 BEGIN
 Write(p^.prec, '|', p^.cal, ' -> ');
 p:=p^.sig;
 END;
 Writeln('NIL');
END;

(*****)

BEGIN
 inicio1:=NIL; inicio2:=NIL;
 Assign (FichPrecio, 'precio.dat');
 Assign (fichCalidad, 'calidad.dat');
```

## EJERCICIOS RESUELTOS

```
Reset(FichPrecio);
WHILE NOT Eof(FichPrecio) DO
 BEGIN
 WHILE NOT Eoln(FichPrecio) DO
 BEGIN
 Read(FichPrecio,precio);
 Reset(FichCalidad);
 enc1:=false;enc2:=false;
 WHILE NOT Eof(FichCalidad) AND(NOT enc1 OR NOT enc2) DO
 BEGIN
 WHILE NOT Eoln(FichCalidad) AND(NOT enc1 OR NOT enc2) DO
 BEGIN
 Read(FichCalidad,calidad);
 IF (NOT enc1) AND (precio=precio1(calidad))
 THEN
 BEGIN
 New(p);
 enc1:=true;
 p^.prec:=precio;
 p^.cal:=calidad;
 p^.sig:=inicio1;
 inicio1:=p;
 END;
 IF NOT enc2 AND (precio=precio2(calidad))
 THEN
 BEGIN
 New(p);
 enc2:=true;
 p^.prec:=precio;
 p^.cal:=calidad;
 p^.sig:=inicio2;
 inicio2:=p;
 END;
 END;
 Readln(FichCalidad);
 END;
 Readln(FichPrecio);
 END;
 END;
 BEGIN
 Writeln('Contenido de la primera lista:');
 EscribeLista(inicio1);
 Writeln('Contenido de la segunda lista:');
 EscribeLista(inicio2);
 readln;
 END.
```

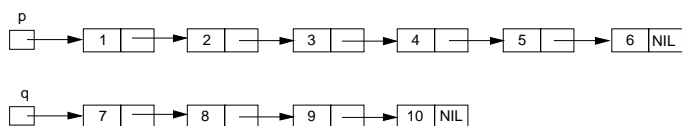
- 12.14** Diseñar un procedimiento que reciba como parámetros dos *punteros*  $p$  y  $q$  y realice el *entrelazado* de las listas correspondientes según se muestra en la figura 12.48. Razonar convenientemente si la transmisión de parámetros debe realizarse por valor o por dirección.

Observe que:

- 1) Los *punteros*  $p$  y  $q$  no sufren ninguna modificación. Como consecuencia, si alguna de las listas estuviese inicialmente vacía, el procedimiento no debe realizar ninguna acción.
- 2) El proceso de intercambio de *punteros* entre elementos homólogos de ambas listas, concluye una vez alcanzado el final de la lista más corta.

## ESTRUCTURAS DINAMICAS DE DATOS

Listas de entrada:



Listas de salida:

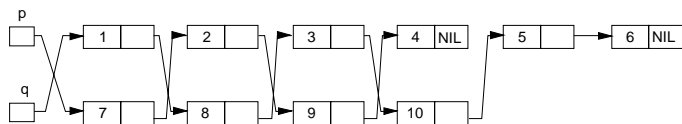


Figura 12.48 Entrelazado de dos listas

### Solución

Se incluye el procedimiento *Cruzar*, solución del ejercicio, dentro de un programa con las operaciones necesarias para ilustrar el uso de *Cruzar*. La sentencia de llamada es:

Cruzar(inicio1, inicio2)

donde los argumentos *inicio1* e *inicio2* dan valor a los parámetros *p* y *q*. Obsérvese que no es necesario el paso de los parámetros por dirección.

```

PROGRAM CruzaListas (input, output);
TYPE
 puntero = ^registro;
 registro = RECORD
 nombre : string[10];
 sig : puntero;
 END;
VAR
 inicio1, inicio2 : puntero;

 (*****)

PROCEDURE Cruzar (p,q : puntero);
VAR
 aux : puntero;
BEGIN
 WHILE (p <> NIL) AND (q <> NIL) DO
 BEGIN
 aux := p^.sig;
 p^.sig := q^.sig;
 q^.sig := aux;
 p := p^.sig;
 q := q^.sig;
 END;
 END;
END;

```



## EJERCICIOS RESUELTOS

```
(*****)

PROCEDURE CrearLista (VAR inicio : puntero);
VAR
 r : puntero;
BEGIN
 inicio := NIL;
 new (r);
 r^.nombre := '';
 Write ('Introduzca el primer nombre de la lista: ');
 Readln (r^.nombre);
 WHILE r^.nombre <> '' DO
 BEGIN
 r^.sig := inicio;
 inicio := r;
 new (r);
 Write ('Introduzca el siguiente nombre de la lista: ');
 Readln (r^.nombre);
 END;
 Dispose (r);
END;

(*****)

PROCEDURE Escribir (inicio : puntero);
VAR
 r : puntero;
BEGIN
 r := inicio;
 Write ('Inicio -->');
 WHILE r <> NIL DO
 BEGIN
 Write (r^.nombre: 10);
 Write (' --> ');
 r := r^.sig;
 END;
 Writeln ('NIL');
END;

(*****
 (* PROGRAMA PRINCIPAL *)

BEGIN
 Writeln ('Creando la 1ª lista...');
 CrearLista (inicio1);
 Writeln ('Creando la 2ª lista...');
 CrearLista (inicio2);

 Writeln ('La 1ª lista es:');
 Escribir (inicio1);
 Writeln;
 Writeln ('La 2ª lista es:');
 Escribir (inicio2);
 Writeln;

 Cruzar (inicio1, inicio2);

 Writeln ('La 1ª lista queda así:');
 Escribir (inicio1);
 Writeln;
 Writeln ('La 2ª lista queda así:');
 Escribir (inicio2);
 Writeln;
 Writeln ('Pulse <Return> para volver al Editor.');
```

```
Readln;
END.
```

## ESTRUCTURAS DINAMICAS DE DATOS

**12.15** Escribir un programa que simule un *Cajero Automático* simplificado, con solo dos opciones:

- 1.- Consulta de saldo
- 2.- Reintegro (sacar dinero)

Se supone ya creado un fichero de clientes del banco con tarjeta del Cajero, cuyos elementos tienen la siguiente estructura:

```
TYPE tipoCliente = RECORD
 cuenta: string[4];
 saldo: real;
END;
```

La tarjeta de cada cliente se simula mediante un fichero de texto con dos líneas:

- 1ª línea: N° secreto de la tarjeta (string[4]);
- 2ª línea: N° de cuenta del cliente. Coincide con uno de los números de cuenta del fichero de clientes.

El programa pedirá al cliente por teclado el nombre de su fichero tarjeta y su n° secreto, y lo comparará con el leído de la tarjeta, denegando el acceso al Cajero si no coinciden.

Cada vez que se realice una operación de reintegro se actualizará el saldo del cliente y el fondo del Cajero (inicializado a un millón de pts.). Si la cantidad que el cliente desea retirar supera el fondo o su saldo, no se permitirá la operación.

### Solución

```
PROGRAM CajeroAutomatico(input, output, clientes, tarjeta);
CONST
 FondoInicial=1E6;
TYPE
 cadena = string[4];
 tipoClientes = RECORD
 cuenta: cadena;
 saldo: real;
 END;
 puntero = ^nodo;
 nodo = RECORD
 cuenta: cadena;
 saldo: real;
 sig: puntero;
 END;
 fichClientes = FILE OF tipoClientes;
VAR
 clientes: fichClientes;
 tarjeta: text;
 cantidad: real;
 nct, nst, ns: cadena;
 salir: boolean;
 p, inicio: puntero;
 fondo: real;
 respu: char;
 nomTar:string[40];
```

## EJERCICIOS RESUELTOS

```
(*****)
PROCEDURE Menu(VAR opcion: char);
BEGIN
 REPEAT
 Writeln(' MENU:');
 Writeln('1. Consulta de saldo');
 Writeln('2. Sacar dinero');
 Writeln('3. Salir');
 Readln(opcion);
 UNTIL opcion IN ['1', '2', '3'];
END;

(*****)
PROCEDURE LeeTarjeta(VAR tarjeta:text; VAR nst, nct: cadena;
 VAR fuera: boolean);
BEGIN
 fuera := false;
 Reset(tarjeta);
 IF NOT Eof(tarjeta)
 THEN
 Readln(tarjeta, nst)
 ELSE
 BEGIN
 Writeln('ERROR: Tarjeta fuera de servicio');
 Writeln('Consulte con su banco');
 fuera := TRUE;
 END;
 IF NOT Eof(tarjeta)
 THEN
 Readln(tarjeta, nct)
 ELSE
 BEGIN
 Writeln('ERROR: Tarjeta fuera de servicio');
 Writeln('Consulte con su banco');
 fuera := TRUE;
 END;
END;

(*****)
FUNCTION BuscaCliente(nct: cadena; p: puntero): puntero;
VAR
 encontrado: boolean;
BEGIN
 encontrado := false;
 WHILE (p<>NIL) AND NOT(encontrado) DO
 IF p^.cuenta = nct
 THEN encontrado := true
 ELSE p := p^.sig;
 BuscaCliente := p;
END;

(*****)
PROCEDURE LeeFichero (VAR clientes: fichClientes;
 VAR inicio: puntero);
VAR
 p: puntero;
 regAux: tipoClientes;
BEGIN
 Reset(clientes);
 inicio := NIL;
 WHILE NOT Eof(clientes) DO
 BEGIN
 New(p);
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

 Read(clientes, regAux);
 p^.cuenta := regAux.cuenta;
 p^.saldo := regAux.saldo;
 p^.sig := inicio;
 inicio := p;
 END;
END;

(*****)

PROCEDURE ActualizaFichero (VAR clientes: fichClientes; p: puntero);
VAR
 regAux: tipoClientes;
BEGIN
 Rewrite(clientes);
 WHILE p<> NIL DO
 BEGIN
 regAux.cuenta := p^.cuenta;
 regAux.saldo := p^.saldo;
 Write(clientes, regAux);
 p := p^.sig;
 END;
 END;

(*****)

BEGIN (* PROGRAMA PRINCIPAL *)
 fondo := fondoInicial;
 salir := false;
 Assign(clientes, 'clientes.dat');
 Writeln('BIENVENIDOS AL SIMULADOR DE CAJERO AUTOMATICO');
 Write('Introduzca su tarjeta pulsando <Intro>');
 Readln;
 Write('¿Nombre del fichero tarjeta?');
 Readln(NomTar);
 Assign(tarjeta, NomTar);
 LeeTarjeta(tarjeta, nct, nst, salir);
 Write('¿Número secreto?');
 Readln(ns);
 IF ns <> nst
 THEN
 BEGIN
 Writeln('ACCESO DENEGADO');
 salir := true;
 END;
 WHILE NOT(salir) DO
 BEGIN
 LeeFichero(clientes, inicio);
 p := BuscaCliente(nct, inicio);
 IF p = NIL
 THEN
 BEGIN
 Writeln('ERROR: Tarjeta fuera de servicio');
 Writeln('Consulte con su banco');
 salir := TRUE;
 END
 ELSE
 BEGIN
 Menu(respu);
 CASE respu OF
 '1': Writeln('Su saldo es...', p^.saldo:8:0);
 '2': BEGIN
 REPEAT
 Write('¿Qué cantidad desea retirar? ');
 Readln(cantidad);
 IF cantidad > fondo
 THEN

```

## EJERCICIOS RESUELTOS

```
 Write('FONDO AGOTADO.');
```

```
 Writeln(' DISPONIBLE...', fondo:10:0);
```

```
 IF cantidad > p^.saldo
```

```
 THEN
```

```
 Writeln('EXCEDE SU SALDO: ', p^.saldo:8:0);
```

```
 UNTIL (cantidad <= fondo) AND
```

```
 (cantidad <= p^.saldo) AND
```

```
 (cantidad >=0);
```

```
 Write('Recoja su dinero...', cantidad:8:0, 'pts.');
```

```
 Readln;
```

```
 p^.saldo := p^.saldo-cantidad;
```

```
 fondo := fondo - cantidad;
```

```
 ActualizaFichero(clientes, inicio);
```

```
 END;
```

```
 '3': salir := true;
```

```
 END; (* CASE *)
```

```
 END; (* ELSE *)
```

```
 END; (* WHILE *)
```

```
 Write('Retire su tarjeta pulsando <Intro>...');
```

```
 Readln;
```

```
END.
```

**12.16** Escribir un programa para simular una ruleta con 37 números (del 0 al 36). Para asegurar la aleatoriedad de cada jugada, se realizan dos tiradas: una para determinar el sentido de giro (0 o 1), y otra que indica cuantos valores (de 0 a 100) se avanzan.

*Nota:* La ruleta debe ser una lista circular doblemente enlazada para asegurar el movimiento en ambos sentidos.

### Solución

```
PROGRAM Ruleta(input, output);
```

```
Uses crt;
```

```
TYPE
```

```
 puntero = ^nodo;
```

```
 nodo = RECORD
```

```
 num: 0..36;
```

```
 sig, ant: puntero;
```

```
 END;
```

```
VAR
```

```
 inicio, fin, p: puntero;
```

```
 a,b: word;
```

```
 i: integer;
```

```
{-----}
```

```
PROCEDURE CrearRuleta (VAR p, f: puntero);
```

```
VAR
```

```
 q: puntero;
```

```
 i: integer;
```

```
BEGIN
```

```
 New(p);
```

```
 p^.num := 36;
```

```
 p^.ant := NIL;
```

```
 p^.sig := NIL;
```

```
 f := p;
```

```
 FOR i:=35 DOWNTO 0 DO
```

```
 BEGIN
```

```
 New(q);
```

```
 q^.num := i;
```

```
 q^.sig := p;
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

 p^.ant := q;
 q^.ant := NIL;
 p := q;
 END;
 (* Conversión de la lista en circular *)
 p^.ant := f;
 p^.sig := p;
END;

{-----}

PROCEDURE Jugada (VAR a, b: word);
BEGIN
 Randomize;
 a := Random(2);
 b := Random(101);
END;

{-----}

BEGIN (* PROGRAMA PRINCIPAL *)
 CrearRuleta(inicio, fin);

 REPEAT
 Writeln;
 Jugada(a, b);

 CASE a OF
 0: BEGIN (* Avanza de dcha. a izda. *)
 p := inicio;
 FOR i := 1 TO b DO
 p := p^.sig;
 END;
 1: BEGIN (* Avanza de izda. a dcha. *)
 p := fin;
 FOR i := 1 TO b DO
 p := p^.ant;
 END;
 END; (* CASE *)

 (* Identificar la posición *)
 Writeln ('Posición de la ruleta -> ', p^.num);
 Write('¿Otra tirada (s/n)?');
 UNTIL Uppcase(Readkey)='N';
 END.

```

- 12.17** Escribir un programa para crear una lista doblemente enlazada con nombres de personas leídos de teclado, manteniéndola en orden alfabético. La lista debe ser similar a la representada en la figura 12.26.

### Solución

```

PROGRAM DosEnlaces (Input,output);
TYPE
 puntero = ^registro;
 registro = RECORD
 nombre : String[20];
 ant,sig : puntero;
 END;
VAR
 cabecera, fin : puntero;
 nuevo : registro;

```

## EJERCICIOS RESUELTOS

```

(*****)

PROCEDURE Insertar (VAR cabecera, fin: puntero; nuevo: registro);
(* Procedimiento que inserta nuevos registros en la lista
manteniendo un orden alfabético. *)
VAR
 busca, aux: puntero;
 hallado: boolean;
BEGIN
 New (aux); (* Se crea un registro nuevo. *)
 aux^ := nuevo; (* Se introduce la información. *)
 aux^.sig := NIL;
 aux^.ant := NIL;
 busca := cabecera;

IF busca = NIL (* Si la lista está vacía, el nuevo registro *)
 THEN (* será el único. *)
 BEGIN
 Writeln ('Insertando el primer registro...');
 cabecera := aux;
 fin := aux;
 END
ELSE
 BEGIN (* Busca el lugar donde insertar el registro nuevo *)
 hallado := FALSE;
 WHILE (busca <> NIL) AND NOT hallado DO
 BEGIN
 Writeln ('Buscando lugar donde insertar...');
 IF busca^.nombre < nuevo.nombre
 THEN
 busca := busca^.sig
 ELSE
 BEGIN
 hallado := TRUE;
 Writeln ('Hallado.')
 END;
 END;
 END;
 IF busca = cabecera (* El registro nuevo es el primero *)
 THEN
 BEGIN
 Writeln ('Insertando en cabeza...');
 cabecera := aux;
 busca^.ant := aux;
 aux^.sig := busca;
 END
 ELSE IF busca = NIL (* El registro nuevo será el último *)
 THEN
 BEGIN
 Writeln ('Insertando en cola...');
 aux^.ant := fin;
 fin^.sig := aux;
 fin := aux
 END
 ELSE
 BEGIN (* El registro nuevo va a estar en medio *)
 Writeln ('Insertando en medio...');
 aux^.ant := busca^.ant;
 aux^.sig := busca;
 busca^.ant^.sig := aux;
 busca^.ant := aux;
 END;
 END; (* Del primer ELSE *)
 END; (* Del PROCEDURE *)

(*****)

PROCEDURE escribeLista (p:puntero); (* Procedimiento recursivo. *)

```

## ESTRUCTURAS DINAMICAS DE DATOS

```
BEGIN
 IF p <> NIL
 THEN
 BEGIN
 Writeln(p^.nombre);
 escribeLista(p^.sig) (* Llamada recursiva *)
 END
 END;
END;

(*****
 (* PROGRAMA PRINCIPAL *)
BEGIN
 cabecera := NIL;
 Nuevo.nombre := '';
 Write ('Introduzca el nombre a insertar (<Return> para acabar): ');
 Readln (Nuevo.nombre);
 WHILE nuevo.nombre <> '' DO
 BEGIN
 nuevo.ant := NIL;
 nuevo.sig := NIL;
 Insertar (cabecera, fin, nuevo);
 Writeln;
 Writeln ('Estado actual de la Lista:');
 EscribeLista (cabecera);
 Writeln;
 Write ('Introduzca el nombre a insertar (<Return> para acabar):');
 Readln (Nuevo.nombre);
 END;
 Writeln ('Pulse <Return> para volver al Editor');
 Readln;
END.
```

**12.18** El ejemplo clásico de objeto fractal es la *curva de Von Koch*, también llamada *copo de nieve*. Su construcción se realiza a partir de un triángulo equilátero de lado unidad, como el de la figura 12.49.

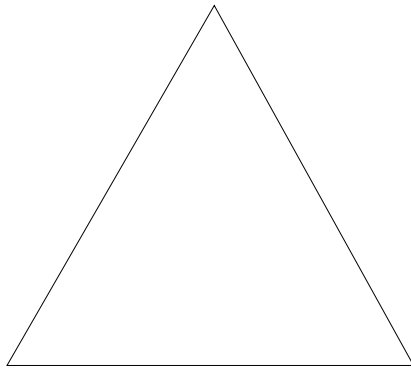


Figura 12.49 Curva de Von Koch. Estado 0



## EJERCICIOS RESUELTOS

A continuación, en el tercio central de cada uno de los tres lados, se dispone un saliente en forma de triángulo equilátero de lado igual a un tercio. Se obtiene así la *estrella de David* de la figura 12.50.

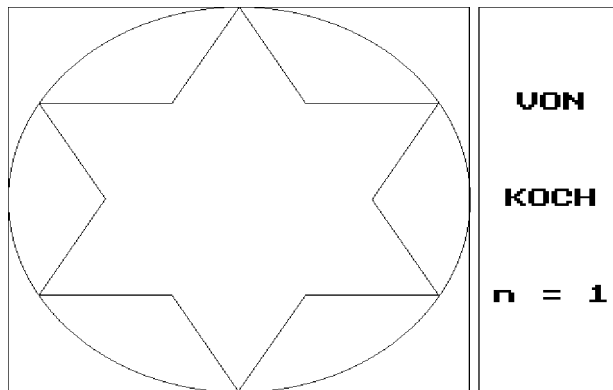


Figura 12.50 Curva de Von Koch. Estado 1

Se realiza la misma operación con los doce lados de la estrella de David, y se obtiene la figura 12.51.

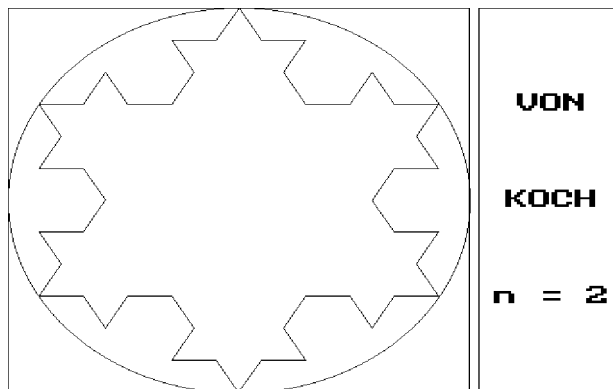


Figura 12.51 Curva de Von Koch. Estado 2

Repetiendo esta operación sucesivas veces se alcanza el estado  $n$ , correspondiente a la curva *copo de nieve*. Esta curva, cuando  $n$  tiende a infinito es de longitud infinita y no rectificable. Se representa en la figura 12.52 la curva correspondiente al estado 5.

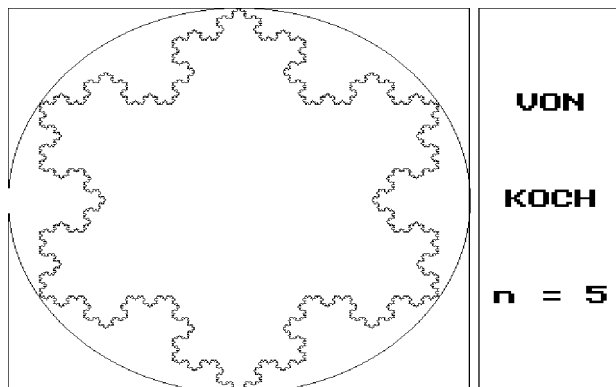


Figura 12.52 Curva de Von Koch. Estado 5

Realizar un programa que almacene en una lista encadenada circular las coordenadas que constituyen los vértices de la curva de Von Koch de estado  $n$ , siendo  $n$  un número entero leído por teclado. Las coordenadas se almacenarán consecutivamente según el sentido de las agujas del reloj. Diseñar primero el algoritmo correspondiente en pseudocódigo. Se recomienda seguir los siguientes pasos:

- a) Inicializar la lista con los valores de las coordenadas de los vértices del triángulo equilátero, según el esquema de la figura 12.53.

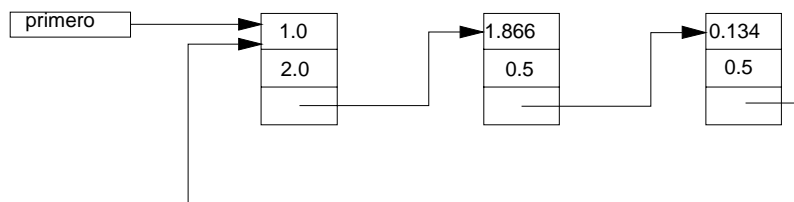


Figura 12.53 Lista de Koch. Estado 0

- b) Obtener los sucesivos estados, utilizando el procedimiento listado a continuación. Las fórmulas utilizadas se deducen de la figura 12.54.

EJERCICIOS RESUELTOS

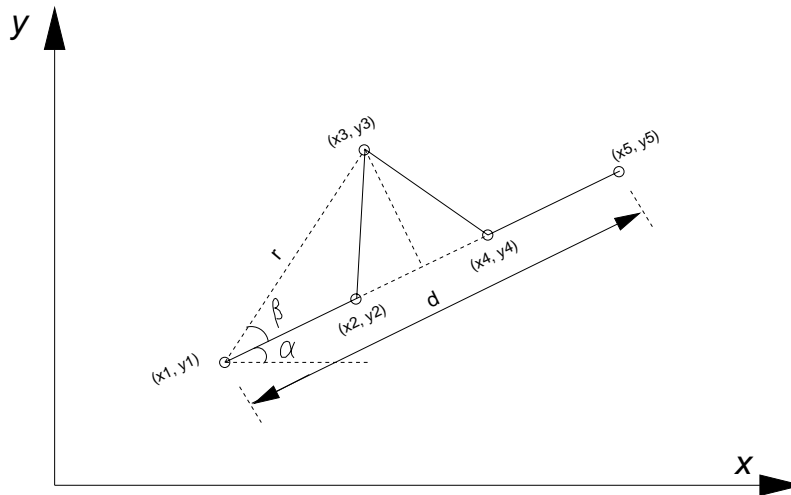


Figura 12.54 Coordenadas de tres nuevos vértices

```

PROCEDURE koch (x1,y1,x5,y5:real;VAR x2,y2,x3,y3,x4,y4:real);
VAR
 d,alfa,r,beta:real;
BEGIN
 d:=Sqrt(Sqr(x5-x1)+Sqr(y5-y1));
 alfa:=arcTan2((y5-y1),(x5-x1));
 beta:=ArcTan(2*0.866/3);
 r:=d*Sqrt(Sqr(0.5)+Sqr(0.866/3));
 x2:=x1+(d/3)*Cos(alfa);
 y2:=y1+(d/3)*Sin(alfa);
 x3:=x1+r*cos(alfa+beta);
 y3:=y1+r*sin(alfa+beta);
 x4:=x1+(2*d/3)*Cos(alfa);
 y4:=y1+(2*d/3)*Sin(alfa);
END;

```

donde  $(x_1, y_1)$  y  $(x_5, y_5)$  son las coordenadas de los extremos de un segmento, representado en la figura 12.55.



Figura 12.55 Segmento original en el estado n

y  $(x_2, y_2)$ ,  $(x_3, y_3)$ ,  $(x_4, y_4)$  son las coordenadas de los nuevos vértices que se generan a partir de dicho segmento, según se representa en la figura 12.56.

## ESTRUCTURAS DINAMICAS DE DATOS

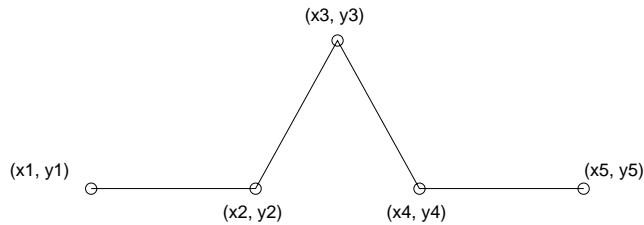


Figura 12.56 Segmento anterior en el estado n+1

Aplicando este procedimiento entre dos nodos consecutivos de la lista, se obtienen tres nuevos nodos que hay que insertar entre los dos que los generaron, como se indica en la figura 12.57.

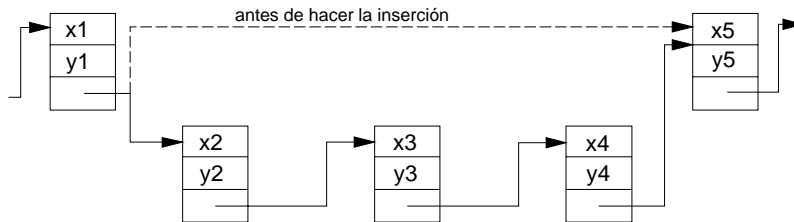


Figura 12.57 Lista de Koch. Inserción de tres nuevos nodos

El programa resuelto a continuación incluye, además de la creación de la lista circular con las coordenadas de los vértices de la curva, un subprograma que escribe las coordenadas de los nodos de la lista, y otro que dibuja la curva resultante en pantalla. Se han incorporado dos diagnósticos de error: Agotamiento de la memoria dinámica (heap) y error en dispositivo gráfico. Además se utiliza una función para el cálculo de la arcotangente, que elimina la indeterminación que produce la utilización de la función estándar de Turbo Pascal  $ArcTan(x)$  para algunos casos. Si al intentar ejecutarlo se produce un error en dispositivo gráfico, lo más posible es que haya que cambiar el directorio en el que el compilador buscará los procedimientos y funciones de manejo de gráficos, establecido mediante una llamada a *InitGraph*.

### Algoritmo en pseudocódigo

- 1.-Inicializar la lista circular con los tres elementos
  - Creación del primer elemento de la lista
  - Introducción de las coordenadas

## EJERCICIOS RESUELTOS

- El puntero se señala a si mismo
- Creación del segundo elemento
  - Introducción de las coordenadas
  - El puntero señala al primer elemento
  - El puntero del primer elemento señala al segundo
- Creación del tercer elemento
  - Introducción de las coordenadas
  - El puntero señala al primer elemento
  - El puntero del segundo elemento señala al tercero
- 2.-Leer el estado que se desea alcanzar
- 3.-Repetir desde 1 a n
  - Siguiente estado
    - Repetir desde el primer elemento
      - hasta volver otra vez al primero
    - \* Crear tres vértices
    - \* Introducir coordenadas con el procedimiento koch
    - \* Colocar los campos puntero
- 4.-Escribir la lista encadenada
- 5.-Pintar la lista

### Codificación en Pascal

```
PROGRAM Fractal_de_Von_Koch (input,output);
USES graph,Crt;
CONST
 (* Coordenadas de los vértices del triángulo equilátero *)
 a=1.0; b=2.0;
 c=1.866; d=0.5;
 e=0.134; f=0.5;
TYPE
 puntero=^vertice;
 vertice=RECORD
 x,y:real;
 siguiente:puntero
 END;
VAR
 inicio:puntero;
 i,n:integer;
 t:char;

 (*****)

PROCEDURE koch (x1,y1,x5,y5:real;VAR x2,y2,x3,y3,x4,y4:real);
VAR
 d,alfa,r,beta:real;

 (*-----*)

FUNCTION arcTan2(x,y:real):real;
(* Resuelve el problema de la indeterminación de la función arcoTan-
gente *)
CONST
 pi=3.141592;
BEGIN
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

IF (x=0.0) AND (y=0.0)
THEN
 BEGIN
 Writeln('Error: Indeterminación en ArcTangente');
 Halt;
 END
ELSE IF y>0 THEN arcTan2:=ArcTan(x/y)
ELSE IF (y=0)AND(x>0) THEN arcTan2:=pi/2
ELSE IF (y=0)AND(x<0) THEN arcTan2:=-pi/2
ELSE IF (y<0)AND(x>=0) THEN arcTan2:=pi+ArcTan(x/y)
ELSE IF (y<0)AND(x<0) THEN arcTan2:=-pi+ArcTan(x/y);
END;

(*-----*)

BEGIN
 d:=Sqrt(Sqr(x5-x1)+Sqr(y5-y1));
 alfa:=arcTan2((y5-y1),(x5-x1));
 beta:=ArcTan(2*0.866/3);
 r:=d*Sqrt(Sqr(0.5)+Sqr(0.866/3));
 x2:=x1+(d/3)*Cos(alfa);
 y2:=y1+(d/3)*Sin(alfa);
 x3:=x1+r*cos(alfa+beta);
 y3:=y1+r*sin(alfa+beta);
 x4:=x1+(2*d/3)*Cos(alfa);
 y4:=y1+(2*d/3)*Sin(alfa);
END;

(******)

PROCEDURE pinta(primero:puntero);
VAR
 tarjeta:integer;
 modo:integer;
 error:integer; (* Código de error devuelto al manejar
 InitGraph y GraphResult *)
 xAspecto,yAspecto:word;
 xp,yp:integer;
 xq,yq:integer;
 p:puntero;
 estado: string[6];

(*-----*)

PROCEDURE transforma (xa,ya:real;VAR xn,yn:integer);
(* Transforma las coordenadas iniciales reales, al modo pantalla *)
CONST
 maximaLongitud=2.0;
BEGIN
 GetAspectRatio(xAspecto,yAspecto);
 xn:=Round(xa*(yAspecto/xAspecto*GetMaxY)/maximaLongitud);
 yn:=Round(GetMaxY*(1-ya/maximaLongitud));
END;

(*-----*)

BEGIN
 DetectGraph(tarjeta,modo); (* Determina tarjeta instalada *)
 InitGraph(tarjeta,modo,'c:\compil\tp\bgi');
 error:=GraphResult;
 IF error<>0 THEN
 BEGIN
 Write('Error en manejo de gráficos, ');
 Writeln(' quizá no tenga tarjeta gráfica');
 Write(' También es posible que no encuentre ');
 Writeln('el controlador BGI correspondiente');
 Halt; (* Halt devuelve el control al sistema operativo *)
 END
 END

```

## EJERCICIOS RESUELTOS

```

 END;
 GetAspectRatio(xAspecto,yAspecto);
 Rectangle(0,0,Round(Yaspecto/xaspecto*GetMaxY),GetMaxY);
 Rectangle(Round(Yaspecto/xaspecto*GetMaxY)+10,0,GetMaxX,GetMaxY);
 SetTextJustify(CenterText,CenterText);
 SetTextStyle(DefaultFont,HorizDir,3);
 OutTextXY(((Round(Yaspecto/xaspecto*GetMaxY)+10)+GetMaxX) DIV 2,
 GetMaxY DIV 2, 'KOCH');
 OutTextXY(((Round(Yaspecto/xaspecto*GetMaxY)+10)+GetMaxX) DIV 2,
 GetMaxY DIV 4, 'VON');
 Str(n, estado);
 estado := 'n = ' + estado;
 OutTextXY(((Round(Yaspecto/xaspecto*GetMaxY)+10)+GetMaxX) DIV 2,
 GetMaxY*3 DIV 4, estado);
 transforma(1.0,1.0,xp,yp);
 Circle(xp,yp,Round(GetMaxY/2*yAspecto/xAspecto));
 p:=primero;
 REPEAT
 transforma(p^.x,p^.y,xp,yp);
 transforma(p^.siguiente^.x,p^.siguiente^.y,xq,yq);
 Line(xp,yp,xq,yq);
 p:=p^.siguiente
 UNTIL (p=primero)AND(KeyPressed);
 CloseGraph;
END;

```

(\*\*\*\*\*)

```

PROCEDURE Escribe(primero:puntero);
VAR

```

```

 p:puntero;
 i:integer;
BEGIN
 p:=primero;
 i:=1;
 Writeln;
 Writeln('Lista de vértices');
 REPEAT
 Writeln('Vértice ', i, ' Coordenadas: x= ',p^.x:6:4,
 ' y= ',p^.y:6:4);
 p:=p^.siguiente;
 i:=i+1;
 UNTIL p=primero;
END;

```

(\*\*\*\*\*)

```

PROCEDURE estado_0(VAR primero:puntero);
(* Creación del ESTADO 0 *)
(* Creación de una lista con los tres vértices del triángulo *)
VAR

```

```

 p:puntero;
BEGIN
 New(primero); (* Creación del primer vértice *)
 WITH primero^ DO
 BEGIN
 x:=a;
 y:=b;
 siguiente:=primero
 END;
 New(p); (* segundo vértice *)
 primero^.siguiente:=p;
 WITH p^ DO
 BEGIN
 x:=c;
 y:=d;
 siguiente:=primero;

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

 END;
 New(p); (* tercer vértice *)
 primero^.siguiente^.siguiente:=p;
 WITH p^ DO
 BEGIN
 x:=e;
 y:=f;
 siguiente:=primero;
 END;
END;

(*****)

PROCEDURE siguiente_Estado(VAR primero:puntero);
VAR
 p,q,r,s:puntero;
BEGIN
 p:=primero;
 GotoXY(10,3);
 Write('Heap: bytes disponibles:');
 REPEAT
 GotoXY(35,3);
 ClrEol;
 Writeln(MemAvail);
 IF MaxAvail < 3 * SizeOf(vertice)
 THEN
 BEGIN
 GotoXY(10,10);
 Writeln('Se ha agotado la memoria heap');
 Halt;
 END
 ELSE
 BEGIN
 New(q);New(r);New(s);
 END;
 koch(p^.x,p^.y,p^.siguiente^.x,p^.siguiente^.y,
 q^.x,q^.y,r^.x,r^.y,s^.x,s^.y);
 q^.siguiente:=r;
 r^.siguiente:=s;
 s^.siguiente:=p^.siguiente;
 p^.siguiente:=q;
 p:=s^.siguiente
 UNTIL p=primero;
END;

(*****)

 (* PROGRAMA PRINCIPAL *)
BEGIN
 ClrScr;
 estado_0(inicio);

 REPEAT
 Write('Introduzca el estado que desea visualizar: ');
 Readln(n);
 UNTIL n>=1;

 FOR i:=1 TO n DO siguiente_Estado(inicio);
 escribe(inicio);

 Writeln;
 Writeln('Pulse una tecla para visualizar el gráfico');
 REPEAT UNTIL KeyPressed;
 t:=ReadKey;
 pinta(inicio);
END.

```



## EJERCICIOS RESUELTOS

**12.19** Escribir un programa que realice la gestión (borrar, insertar o escribir) de una estructura dinámica de datos de tipo *árbol de búsqueda*.

### Solución

Se utilizan los algoritmos de tratamiento de árboles binarios descritos en la sección 12.7. La escritura del árbol creado puede ser imprimida de cuatro formas distintas, según señala el menú presentado por el procedimiento *Inicio2*. En la figura 12.58 puede verse un ejemplo de ejecución del programa.

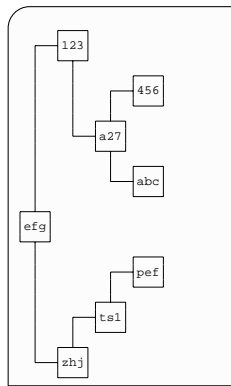


Figura 12.58 Ejemplo de ejecución. Ejercicio 12.19

### Codificación en Pascal

```
PROGRAM Arbol_Busqueda(input,output);
Uses Crt, Printer;
TYPE
 no = String[3];
 linea = String[80];
 ref = ^nodo;
 nodo = RECORD
 clave: no;
 contador: integer;
 izquierdo, derecho: ref;
 END;
VAR
 nombre: no;
 raiz, q: ref;
 c1, c2: char; (* Elección de opciones en los menús *)
 line: linea; (* Línea de enlace de las ramas del árbol *)

(* ***** *)

PROCEDURE Inicio1;
(* presenta el menú inicial de programa *)
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

BEGIN
 ClrScr;
 Gotoxy(10,6);
 Write('OPERACION :');
 Gotoxy(16,8);
 Write('1.* INSERTAR ELEMENTOS .');
 Gotoxy(16,10);
 Write('2.* BORRAR UN ELEMENTO .');
 Gotoxy(16,12);
 Write('3.* ESCRIBIR EL ARBOL .');
 Gotoxy(16,14);
 Write('4.* FIN .');
 Gotoxy(7,22);
 Write('OPCION ELEGIDA » » » ');
 c1 := Readkey;
 IF NOT (c1 IN ['1'..'4'])
 THEN
 BEGIN
 ClrScr;
 Gotoxy(10,10);
 Write('OPCION INCORRECTA . REPITE .');
 Delay(2000);
 Inicio1
 END
 END;

 (* ***** *)

PROCEDURE Inicio2;
(* Presenta un menú con diferentes formas de representar el árbol *)
BEGIN
 ClrScr;
 Gotoxy(14,8);
 Write('FORMAS DE ESCRIBIR EL ARBOL :');
 Gotoxy(18,10);
 Write('1.* PREORDEN .');
 Gotoxy(18,12);
 Write('2.* ORDEN CENTRAL .');
 Gotoxy(18,14);
 Write('3.* POSTORDEN .');
 Gotoxy(18,16);
 Write('4.* FORMATO DE ARBOL .');
 Gotoxy(7,22);
 Write('OPCION ELEGIDA » » » ');
 c2 := Readkey;
 IF NOT (c2 IN ['1'..'4'])
 THEN
 BEGIN
 ClrScr;
 Gotoxy(10,10);
 Write('OPCION NO CONTEMPLADA. REPITE .');
 Delay(2000);
 Inicio2
 END
 END;

END;

(* ***** *)

PROCEDURE Preorden(a:ref);
(* Lista los nodos del árbol por preorden *)
BEGIN
 IF a<>NIL
 THEN
 BEGIN
 Writeln('CLAVE : ',a^.clave:10,' CONTADOR : ',a^.contador);
 Write(lst,'CLAVE : ',a^.clave:10);
 Writeln(lst,' CONTADOR : ',a^.contador);
 END
 END;

```

## EJERCICIOS RESUELTOS

```

 Preorden(a^.izquierdo);
 Preorden(a^.derecho)
 END
END;

(* ***** *)

PROCEDURE Orden_Central(a:ref);
 (* Lista los nodos del árbol por orden central *)
 BEGIN
 IF a<>NIL
 THEN
 BEGIN
 Orden_Central(a^.izquierdo);
 Writeln('CLAVE: ',a^.clave:10,' CONTADOR: ',a^.contador);
 Write(lst,' CLAVE : ',a^.clave:10);
 Writeln(lst,' CONTADOR : ',a^.contador);
 Orden_Central(a^.derecho)
 END
 END;
 END;

 (* ***** *)

PROCEDURE Postorden(a:ref);
 (* Lista los nodos del árbol por postorden *)
 BEGIN
 IF a<>NIL
 THEN
 BEGIN
 Postorden(a^.izquierdo);
 Postorden(a^.derecho);
 Writeln('CLAVE: ',a^.clave:10,'CONTADOR: ',a^.contador);
 Write(lst,' CLAVE : ',a^.clave:10);
 Writeln(lst,' CONTADOR : ',a^.contador);
 END
 END;
 END;

 (* ***** *)

PROCEDURE Buscar(x:no;VAR p:ref);
 (* Encuentra la posición de un elemento de clave "x" en el árbol
 incrementando en 1 el contador de apariciones de dicho elemento
 y si no existe lo crea en el lugar que le corresponde *)
 BEGIN
 ClrScr;
 IF p=NIL
 THEN
 BEGIN
 New(p);
 p^.clave := x;
 p^.izquierdo := NIL;
 p^.derecho := NIL;
 p^.contador := 1
 END
 ELSE
 IF x<p^.clave
 THEN
 Buscar(x,p^.izquierdo)
 ELSE
 IF x>p^.clave
 THEN
 Buscar(x,p^.derecho)
 ELSE
 p^.contador := p^.contador+1
 END
 END;
 END;

 (* ***** *)

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

PROCEDURE Insertar;
 (* Crea nuevos nodos en el árbol hasta que
 se indica el final de inserción *)
BEGIN
 ClrScr;
 Gotoxy(10,10);
 Write('CUANDO NO SE QUIERAN INSERTAR MAS ELEMENTOS ');
 Gotoxy(10,12);
 Write('INTRODUCIR LA CLAVE "@" (clave maximo de 3 caracteres).');
 Delay(3500);
 ClrScr;
 Gotoxy(10,10);
 Write('INTRODUCIR UNA CLAVE : ');
 Gotoxy(10,12);
 Readln(nombre);
 raiz := NIL;
 WHILE nombre<>'@' DO
 BEGIN
 Buscar(nombre,raiz);
 Gotoxy(10,10);
 Write('INTRODUCIR UNA CLAVE : ');
 Gotoxy(10,12);
 Readln(nombre)
 END
 END;

 (* ***** *)

PROCEDURE Bor(VAR d:ref);
 (* Establece las operaciones necesarias para borrar un nodo en el
 caso de que este tenga sucesores en ambas ramas (izquierda y
 derecha *)
BEGIN
 IF d^.derecho<>NIL
 THEN
 Bor(d^.derecho)
 ELSE
 BEGIN
 q^.clave := d^.clave;
 q^.contador := d^.contador;
 q := d;
 d := d^.izquierdo
 END
 END;

 (* ***** *)

PROCEDURE Borrar(x:no;VAR p:ref);
 (* Borra un nodo del árbol de clave "x" al cual apunta "p" *)
BEGIN
 IF p=NIL
 THEN
 BEGIN
 ClrScr;
 Gotoxy(10,10);
 Write('EL ELEMENTO ',x,' NO EXISTE EN EL ARBOL. ');
 Delay(4000)
 END
 ELSE
 IF x<p^.clave
 THEN
 Borrar(x,p^.izquierdo)
 ELSE
 IF x>p^.clave
 THEN
 Borrar(x,p^.derecho)
 ELSE

```

## EJERCICIOS RESUELTOS

```

BEGIN
 q := p;
 IF q^.derecho=NIL
 THEN
 p := q^.izquierdo
 ELSE
 IF q^.izquierdo=NIL
 THEN
 p := q^.derecho
 ELSE
 Bor(q^.izquierdo);
 dispose(q);
 ClrScr;
 Gotoxy(10,10);
 Write('EL ELEMENTO "',nombre);
 Write('" ESTA BORRADO DEL ARBOL');
 Delay(4000)
 END
END;

(* ***** *)

(* Procedimientos de recursividad indirecta para obtener la
representación del árbol de búsqueda con forma arbórea *)
PROCEDURE Printright (p: ref; h: integer); FORWARD;
PROCEDURE Printleft (p: ref; h: integer);
(* Escribe los nodos que son referenciados por el campo
puntero "izquierdo" de otro nodo *)
VAR
 a: ref;
 r: integer; (* Nivel del árbol en que se esta *)
 i: integer;
BEGIN
 a := p^.izquierdo;
 IF a<>NIL
 THEN
 BEGIN
 r := h+1;
 Printleft(a,r);
 line[r*5-2] := ' ';
 FOR i:=1 TO (h-1)*5 DO
 Write(lst,line[i]);
 IF a^.izquierDO<>NIL
 THEN
 Writeln(lst,' ', Chr(218), Chr(196), Chr(193),
 Chr(196), Chr(191));
 ELSE
 Writeln(lst,' ', Chr(218), Chr(196), Chr(196),
 Chr(196), Chr(191));
 FOR i:=1 TO h*5-3 DO
 Write(lst,line[i]);
 Writeln(lst, Chr(218), Chr(196), Chr(196), Chr(180),
 a^.clave:3, Chr(179));
 line[h*5-2] := Chr(179);
 FOR i:=1 TO h*5 DO
 Write(lst,Line[i]);
 IF a^.derecho<>NIL
 THEN
 Writeln(lst, Chr(192), Chr(196), Chr(194),
 Chr(196), Chr(217))
 ELSE
 Writeln(lst, Chr(192), Chr(196), Chr(196),
 Chr(196), Chr(217));
 Printright(a,r)
 END
 END
END;

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

PROCEDURE Printright ;
 (* Escribe los nodos que son referenciados por el campo
 puntero "izquierdo" de otro nodo *)
VAR
 a: ref;
 r: integer; (* Nivel del arbol en que se esta *)
 i: integer;
BEGIN
 a := p^.derecho;
 IF a<>NIL
 THEN
 BEGIN
 r := h+1;
 line[(h-1)*5+3] := Chr(179);
 Printleft(a,r);
 line[h*5+3] := ' ';
 FOR i:=1 TO h*5 DO
 Write(lst, line[i]);
 IF a^.izquierdo<>NIL
 THEN
 Writeln(lst, Chr(218), Chr(196), Chr(193),
 Chr(196), Chr(191))
 ELSE
 Writeln(lst, Chr(218), Chr(196), Chr(196),
 Chr(196), Chr(191));
 FOR i:=1 TO h*5-3 DO
 Write(lst, line[i]);
 Writeln(lst, Chr(192), Chr(196), Chr(196),
 Chr(180), a^.clave:3, Chr(179));
 line[(h-1)*5+3] := ' ';
 FOR i:=1 TO h*5 DO
 Write(lst, line[i]);
 IF a^.derecho<>NIL
 THEN
 Writeln(lst, Chr(192), Chr(196), Chr(194),
 Chr(196), Chr(217))
 ELSE
 Writeln(lst, Chr(192), Chr(196), Chr(196),
 Chr(196), Chr(217));
 Printright(a,r)
 END
 END;
 END;

 (* ***** *)

PROCEDURE Proof (p: ref; altura: integer; VAR hmax: integer);
 (* Halla la altura maxima o profundidad del arbol de busqueda *)
VAR
 s: ref;
BEGIN
 IF p<>NIL
 THEN
 BEGIN
 altura := altura+1;
 s := p^.izquierdo;
 Proof(s,altura,hmax);
 s := p^.derecho;
 Proof(s,altura,hmax)
 END
 ELSE
 IF altura>hmax
 THEN
 hmax := altura
 END;
 END;

 (* ***** *)

```

## EJERCICIOS RESUELTOS

```

PROCEDURE Forma_Arbol;
 (* Representa el arbol con forma arborea *)
VAR
 max:integer; (* Profundidad del arbol *)
 i:integer;
BEGIN
 max := 0;
 Proof(raiz,0,max);
 ClrScr;
 Gotoxy(5,10);
 IF max>16
 THEN
 Write(' ARBOL CON DEMASIADA ALTURA.NO CABE EN LA HOJA')
 ELSE
 IF max=0
 THEN
 Write('ARBOL VACIO ')
 ELSE
 BEGIN
 Write('SE ESTA IMPRIMIENDO EL ARBOL');
 FOR i:=1 TO 80
 DO line[i] := ' ';
 Printleft(raiz,1);
 IF raiz^.izquierdo <> NIL
 THEN
 Writeln(lst, Chr(218), Chr(196), Chr(193),
 Chr(196), Chr(191))
 ELSE
 Writeln(lst, Chr(218), Chr(196), Chr(196),
 Chr(196), Chr(191));
 Writeln(lst, Chr(179), raiz^.clave:3, Chr(179));
 IF raiz^.derecho<> NIL
 THEN
 BEGIN
 Writeln(lst, Chr(192), Chr(196), Chr(194),
 Chr(196), Chr(217));
 line[3] := Chr(179)
 END
 ELSE
 Writeln(lst, Chr(192), Chr(196), Chr(196),
 Chr(196), Chr(217));
 END
 END
 END
 END
 (* ***** PROGRAMA PRINCIPAL ***** *)
BEGIN
 REPEAT
 Inicio1;
 CASE c1 OF
 '1':Insertar;
 '2':BEGIN
 ClrScr;
 Gotoxy(14,10);
 Write('ELEMENTO A BORRAR :');
 Gotoxy(14,12);
 Readln(nombre);
 Borrar(nombre,raiz);
 END;
 '3':BEGIN
 ClrScr;
 Inicio2;
 CASE c2 OF
 '1':BEGIN
 ClrScr;
 Gotoxy(12,1);
 Write(' ARBOL de BUSQUEDA EN PREORDEN ');
 Gotoxy(12,2);
 Write('***** ');
 END;
 END;
 END;
 END;
 UNTIL c1='0';
END;

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

 Writeln(lst);
 Writeln(lst,'ARBOL de BUSQUEDA EN PREORDEN');
 Writeln(lst,'*****');
 Writeln;Writeln;
 Writeln(lst); Writeln(lst);
 Preorden(raiz);
 END;
'2':BEGIN
 ClrScr;
 Gotoxy(12,1);
 Write(' ARBOL de BUSQUEDA EN ORDEN CENTRAL ');
 Gotoxy(12,2);
 Write('*****');
 Writeln(lst,'ARBOL de BUSQUEDA. ORDEN CENTRAL');
 Write(lst,'*****');
 Writeln(lst,'***** ');
 Writeln;Writeln;
 Writeln(lst); Writeln(lst);
 Orden_Central(raiz);
 END;
'3':BEGIN
 ClrScr;
 Gotoxy(12,1);
 Write(' ARBOL de BUSQUEDA EN POSTORDEN ');
 Gotoxy(12,2);
 Write('*****');
 Writeln(lst,'ARBOL de BUSQUEDA. POSTORDEN ');
 Gotoxy(12,2);
 Writeln(lst,'*****');
 Writeln; Writeln;
 Writeln(lst); Writeln(lst);
 Postorden(raiz);
 END;
'4':BEGIN
 ClrScr;
 Gotoxy(10,10);
 Write('LISTANDO EL ARBOL DE BUSQUEDA ');
 Write('EN FORMA ARBOREA');
 Writeln(lst);
 Writeln(lst,' ARBOL DE BUSQUEDA');
 Writeln(lst,' ***** ');
 Writeln(lst); Writeln(lst); Writeln(lst);
 Forma_Arbol
 END
 END; (* Case c2 *)
 Gotoxy(1,24);
 Writeln ;
 Write(' PULSAR UNA TECLA PARA CONTINUAR ');
 c2 := Readkey;
 END (* Opcion 3 *)
 END (* Case c1 *)
UNTIL c1='4'
END.

```

**12.20** Construir mediante una *unit* un *Tipo Abstracto de Datos lista* para manejar una *pila de caracteres*. Ilustrar su utilización por parte de un programa.

### Solución

```
UNIT TADListaCaract;
```



## EJERCICIOS RESUELTOS

```
INTERFACE

TYPE puntero = ^nodo;
 nodo = RECORD
 info:char;
 sig:puntero;
 END;

PROCEDURE Sacar(VAR pila:puntero;VAR componente:char);
PROCEDURE Meter(VAR pila:puntero;VAR componente:char);
PROCEDURE Inicializar(VAR pila:puntero);
FUNCTION Vacia(Pila:puntero):boolean;

IMPLEMENTATION

PROCEDURE Sacar(VAR pila:puntero;VAR componente:char);
{Elimina el primer elemento de la pila, si no está vacía}
{pila = puntero a la cabeza de la pila}
{componente = elemento quitado de la pila}

VAR p:puntero;
BEGIN
p:=pila;
componente:=pila^.info;
pila:=pila^.sig;
dispose(p);
END;

(*****)

PROCEDURE Meter(VAR pila:puntero;VAR componente:char);
{Inserta un componente como primer elemento de la pila, que se supone
inicializada a NIL}

VAR p:puntero;
BEGIN
new(p);
p^.info:=componente;
p^.sig:=pila;
pila:=p;
END;

(*****)

PROCEDURE Inicializar(VAR pila:puntero);
BEGIN
pila := NIL;
END;

(*****)

FUNCTION Vacia(Pila:puntero):boolean;
BEGIN
Vacía := (pila = NIL);
END;

END.
```

A continuación se incluye un pequeño programa como ejemplo de utilización de la *unit* anterior.

## ESTRUCTURAS DINAMICAS DE DATOS

```
PROGRAM UsaTADLista(input,output);
(* Lee una línea de caracteres y la imprime en orden inverso *)

Uses TadlistaCaract;

VAR pila: puntero;
 caracter: char;

BEGIN
 Writeln('Introduzca una secuencia de caracteres...');
 Inicializar(pila);
 WHILE NOT Eoln DO {Lee y guarda caracteres}
 BEGIN
 read(caracter);
 Meter(pila,caracter);
 END;
 Readln;

 Writeln('La secuencia invertida es:');
 WHILE NOT Vacía(pila) DO {Imprime caracteres en orden inverso}
 BEGIN
 Sacar(pila,caracter);
 Write(caracter);
 END;
 Writeln;

 Write('Pulse <Intro> para acabar...');
 Readln;
END.
```

**12.21** La empresa privada ECELAC S.A. ha encargado realizar una encuesta con una muestra de 500 personas por cada comunidad autónoma para conocer su peso y altura. Estos datos serán utilizados posteriormente para realizar estadísticas.

Se pide:

- Construir una *Unit* que permita realizar operaciones sobre un fichero. La definición de estas operaciones se muestra a continuación.

```
PROCEDURE AbrirFichero(VAR f:Archivo);
PROCEDURE ModificarRegistro(VAR f:Archivo; reg:clave);
PROCEDURE BorrarRegistro(VAR f:Archivo; reg:clave);
PROCEDURE InsertarRegistro(VAR f:Archivo);
PROCEDURE VerRegistro(VAR f:Archivo; reg:clave);
PROCEDURE ListarFichero(VAR f:Archivo);
```

- Crear un programa que utilice la *Unit* anterior.

### Análisis

A la hora de implementar estas operaciones es muy cómodo realizar accesos directos sobre el fichero, tanto para escritura como para lectura de registros. Para conseguirlo, se utilizará una *tabla hash* como se muestra en la figura 12.59. Una *tabla hash* es un *ARRAY* cuyo comportamiento difiere un poco a los estudiados en el capítulo 8. La búsqueda, inserción y borrado de elementos en el *ARRAY* se realiza en base a una función (*función hash*), la cual toma el nombre de una variable y calcula

## EJERCICIOS RESUELTOS

la posición de la tabla en que se almacenan los valores relacionados con dicha variable. Si la posición calculada ya está ocupada, se siguen buscando posiciones sucesivas hasta encontrar una libre, y si no se encuentran la tabla está llena.

También es posible utilizar una *tabla hash abierta* (la anterior sería *cerrada*) la cual posee una lista enlazada asociada a cada una de las posiciones de la tabla. De esta forma, los valores se insertan en la lista enlazada con punteros y no habrá limitación del tamaño de la *tabla hash* a la hora de insertar muchos valores.

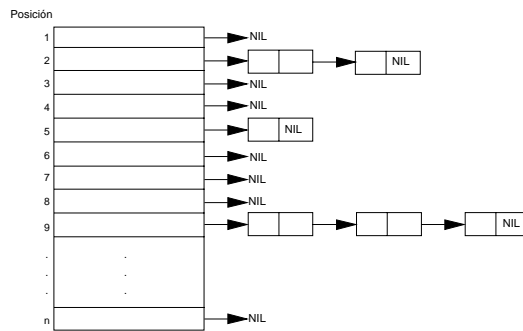


Figura 12.59 Tabla hash abierta

Volviendo al ejercicio, los registros del fichero tienen al menos los siguientes campos:

```
registro = RECORD
 dni: string[10];
 nombre: string[40];
 altura, peso: real;
END;
```

Se puede utilizar el campo `dni` como clave del fichero, por tanto para cada registro del fichero se calculará su posición en la tabla hash transformando el campo `dni` (función hash) y se insertará un nodo en la lista enlazada conteniendo la clave del registro y la posición que ocupa en el fichero.

Las operaciones de inserción, modificación y borrado de registros se realizan por su clave como sigue:

*Modificar:* se busca en la tabla la posición y se accede directamente al fichero.

*Borrar:* se busca en la tabla la posición, se elimina de la lista y se inserta la posición en una lista auxiliar de registros borrados. También se marca el registro en el fichero como borrado.

*Insertar:* se busca en la lista auxiliar una posición. Si existe se inserta en esa posición; si no existe, se inserta al final del fichero.

## ESTRUCTURAS DINAMICAS DE DATOS

### Solución

```
Unit fichero;
INTERFACE
Uses Crt;
CONST m = 101;
TYPE
 nodo = ^claves;
 nodo1 = ^posiciones;
 nodo2 = ^listpersonas;
 posiciones = RECORD
 posicion:integer;
 sucesor:nodo1;
 END;
 claves = RECORD
 valor:string[10];
 posicion:integer;
 sucesor:nodo;
 END;
 persona = RECORD
 dni:string[10];
 nombre:string[40];
 altura,peso:real;
 END;
 listpersonas = RECORD
 valor:persona;
 sucesor:nodo2;
 END;
 Archivo = FILE OF persona;
 tablahash = ARRAY [1 .. m] OF nodo;
 clave = string[10];

VAR
 tab:tablahash;
 cabeza:nodo1;

PROCEDURE Inicializar;
PROCEDURE Liberar;
PROCEDURE AbrirFichero(VAR f:Archivo);
PROCEDURE VolcarFichero(VAR f:Archivo);
PROCEDURE ModificarRegistro(VAR f:Archivo; reg:clave);
PROCEDURE BorrarRegistro(VAR f:Archivo; reg:clave);
PROCEDURE InsertarRegistro(VAR f:Archivo);
PROCEDURE VerRegistro(VAR f:Archivo; reg:clave);
PROCEDURE ListarFichero(VAR f:Archivo);
PROCEDURE SalvarFichero(VAR f:Archivo);
 { Procedimientos y funciones utilizados al implementar
 los procedimientos anteriores}
PROCEDURE LeerRegistro(VAR registro:persona);
PROCEDURE InsertarBorrado(VAR cabeza:nodo1; pos:integer);
PROCEDURE Borrarpos(poshash:integer; reg:clave);
FUNCTION Buscarpos(poshash:integer;reg:clave):integer;
FUNCTION Hash (reg:clave):integer;
PROCEDURE Vercampos(registro:persona);
PROCEDURE Insertarpos(reg:clave; pos:integer);

IMPLEMENTATION
PROCEDURE Inicializar;
 { Inicializa la tabla Hash y la lista de claves borradas}

VAR
 i:integer;
```

## EJERCICIOS RESUELTOS

```
BEGIN
 cabeza:=NIL;
 FOR i:= 1 to m DO
 tab[i]:=NIL;
 END;

 {-----}

 PROCEDURE Liberar;
 { Libera la memoria ocupada durante la ejecución al finalizar el
 trabajo }

 VAR
 i:integer;
 q:nodol;
 p:nodo;
 BEGIN
 WHILE cabeza <> NIL DO
 BEGIN
 q:=cabeza;
 cabeza:=cabeza^.sucesor;
 Dispose(q);
 END;
 FOR i:= 1 TO m DO
 IF tab[i] <> NIL
 THEN
 BEGIN
 p:=tab[i];
 tab[i]:=tab[i]^sucesor;
 Dispose(p);
 END;
 END;
 END;

 {-----}

 PROCEDURE LeerRegistro(VAR registro:persona);
 { Introduce datos en los campos de un registro }

 BEGIN
 WITH registro DO
 BEGIN
 Write (' D.N.I: ');
 Readln(dni);
 Write (' Nombre: ');
 Readln(nombre);
 Write (' Peso: ');
 Readln(peso);
 Write (' Altura: ');
 Readln(altura);
 END;
 END;

 {-----}

 PROCEDURE InsertarBorrado(VAR cabeza:nodol; pos:integer);
 { Inserta en la lista de borrados la posición en el fichero del
 registro que se borró}

 VAR
 q:nodol;
 BEGIN
 New(q);
 q^.posicion:= pos;
 q^.sucesor:= NIL;
 cabeza:=q;
 END;

 {-----}
```

## ESTRUCTURAS DINAMICAS DE DATOS

```

PROCEDURE Borrarpos(poshash:integer; reg:clave);
{ Borra un nodo de la tabla Hash indicado por la posición poshash }
VAR
 encontrado:boolean;
 p,q:nodo;
BEGIN
 p:=tab[poshash];
 encontrado:=false;
 q:=p;
 WHILE (p <> NIL) AND (NOT encontrado) DO
 BEGIN
 IF p^.valor <> reg
 THEN
 BEGIN
 q:=p;
 p:= p^.sucesor;
 END
 ELSE
 encontrado:=true;
 END;
 IF p <> NIL
 THEN
 IF q=p
 THEN
 BEGIN
 tab[poshash]:=NIL;
 Dispose(p)
 END
 ELSE
 BEGIN
 q^.sucesor:=p^.sucesor;
 Dispose(p);
 END;
 END;
 END;
 {-----}

FUNCTION Buscarpos(poshash:integer;reg:clave):integer;
{ Busca un nodo en la tabla Hash indicado por la posición poshash }

VAR
 p:nodo;
 encontrado:boolean;
BEGIN
 p:=tab[poshash];
 encontrado:=false;
 WHILE (p <> NIL) AND (NOT encontrado) DO
 IF p^.valor <> reg
 THEN
 p:= p^.sucesor
 ELSE
 encontrado:=true;
 IF p <> NIL
 THEN
 Buscarpos:= p^.posicion
 ELSE
 Buscarpos:= -1
 END;
 {-----}

FUNCTION Hash (reg:clave):integer;
VAR
 i:integer;
 num:LongInt;

```

## EJERCICIOS RESUELTOS

```

BEGIN
 num := 0;
 FOR i:= 1 to ord(reg[0]) DO
 BEGIN
 num := num + ord(reg[i]) * 100;
 Hash := num MOD m
 END;
 END;

{-----}

PROCEDURE Vercampos(registro:persona);
 { Lista los campos de un registro }

BEGIN
 WITH registro DO
 BEGIN
 Writeln(' D.N.I ',dni);
 Writeln(' NOMBRE ',nombre);
 Writeln(' PESO ',peso:5:1);
 Writeln(' ALTURA ',altura:3:2);
 END;
 Readln;
END;

{-----}

PROCEDURE Insertarpos(reg:clave; pos:integer);
 { Inserta un nuevo nodo en la tabla Hash, cuya posición depende
 de la clave }

VAR
 poshash:integer;
 p:nodo;
BEGIN
 poshash:=Hash(reg);
 New(p);
 p^.valor:=reg;
 p^.posicion:=pos;
 p^.sucesor:=tab[poshash];
 tab[poshash]:=p;
END;

{-----}

PROCEDURE VolcarFichero(VAR f:Archivo);
 { Si el fichero está creado, la clave y la posición en el fichero
 de los registros se insertan en la tabla Hash }

VAR
 contreg:integer; (* Cuenta los registros del fichero*)
 registro:persona;
BEGIN
 contreg:=0;
 Reset(f);
 WHILE NOT Eof(f) DO
 BEGIN
 Read(f,registro);
 Insertarpos(registro.dni,contreg);
 contreg:=contreg+1
 END;
 END;

{-----}

PROCEDURE AbrirFichero(VAR f:Archivo);
 { Prepara el fichero para escritura }

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

BEGIN
 Rewrite(f);
END;

{-----}

PROCEDURE ModificarRegistro(VAR f:Archivo; reg:clave);
 { Permite modificar los campos de un registro excepto la clave }

VAR
 poshash,posfich:integer;
 registro:persona;
BEGIN
 Clrscr;
 Reset(f);
 poshash := Hash(reg);
 posfich := Buscarpos(poshash,reg);
 IF posfich <> -1
 THEN
 BEGIN
 Seek(f,posfich);
 Read(f,registro);
 Vercampos(registro);
 Writeln;
 Seek(f,posfich);
 WITH registro DO
 BEGIN
 Write (' Nombre: ');
 Readln(nombre);
 Write (' Peso: ');
 Readln(peso);
 Write (' Altura: ');
 Readln(altura);
 END;
 Write(f,registro);
 END
 ELSE
 BEGIN
 Writeln (' No existe el registro de clave: ', reg);
 Readln;
 END;
 Close(f);
END;

{-----}

PROCEDURE BorrarRegistro(VAR f:Archivo; reg:clave);
 { Borra un nodo de la tabla Hash, inserta un nodo en la lista de
 borrados y pone una marca en el campo dni dentro del fichero}

VAR
 poshash,posfich:integer;
 registro:persona;
BEGIN
 Reset(f);
 Clrscr;
 poshash := Hash(reg);
 posfich := Buscarpos(poshash,reg);
 Borrarpos(poshash,reg);
 IF posfich <> -1
 THEN
 BEGIN
 InsertarBorrado(cabeza,posfich);
 Seek(f,posfich);
 Read(f,registro);
 registro.dni:= '-1';
 Seek(f,posfich);
 END;
 END;

```



## EJERCICIOS RESUELTOS

```

 Write(f,registro);
 END
ELSE
 BEGIN
 Writeln (' No existe el registro de clave: ', reg);
 Readln;
 END;
Close(f);
END;

{-----}

PROCEDURE InsertarRegistro(VAR f:Archivo);
{ Busca posición en la lista de borrados. Si la encuentra inserta
el registro en el fichero en esa posición sino, lo añade al final
del fichero e inserta un nodo en la tabla Hash }

VAR
 pos:integer;
 registro:persona;
 q:nodol;
BEGIN
 Clrscr;
 Reset(f);
 IF cabeza <> NIL
 THEN
 BEGIN
 q:= cabeza;
 pos:=cabeza^.posicion;
 cabeza:=cabeza^.sucesor;
 Dispose(q)
 END
 ELSE
 pos:=Filesize(f);
 LeerRegistro(registro);
 Seek(f,pos);
 Write(f,registro);
 Insertarpos(registro.dni,pos);
 Close(f);
 END;
END;

{-----}

PROCEDURE VerRegistro(VAR F:Archivo; reg:clave);
{ Permite ver el contenido de un registro arbitrario }

VAR
 poshash,posfich:integer;
 registro:persona;
BEGIN
 Clrscr;
 Reset(f);
 poshash := Hash(reg);
 posfich := Buscarpos(poshash,reg);
 IF posfich <> -1
 THEN
 BEGIN
 Seek(f,posfich);
 Read(f,registro);
 Vercampos(registro);
 END
 ELSE
 BEGIN
 Writeln (' No existe el registro de clave: ', reg);
 Readln;
 END
 END;
END;

```

## ESTRUCTURAS DINAMICAS DE DATOS

```

 END;
 Close(f);
END;

{-----}

PROCEDURE ListarFichero(VAR f:Archivo);
 { Lista el contenido de cada uno de los registros que contiene el
 fichero }

VAR
 registro:persona;
BEGIN
 Clrscr;
 Reset(f);
 IF Eof(f)
 THEN
 BEGIN
 Writeln (' <<<<< FICHERO VACIO >>>>>>> ');
 Delay (2000);
 END
 ELSE
 WHILE NOT Eof(f) DO
 BEGIN
 Read(f,registro);
 IF registro.dni <> '-1' (* Si dni <> '-1' está borrado *)
 THEN
 (* y se marca en el fichero el D.N.I *)
 BEGIN
 Writeln;
 Vercampos(registro);
 END;
 END;
 END;
 Close(f);
 END;

{-----}

PROCEDURE SalvarFichero(VAR f:Archivo);
 { Elimina los registros marcados en el fichero }

VAR
 cabeza,p:nodo2;
BEGIN
 cabeza:=NIL;
 Reset(f);
 WHILE NOT Eof(f) DO
 BEGIN
 New(p);
 Read(f,p^.valor);
 p^.sucesor:=cabeza;
 cabeza:=p;
 END;
 Rewrite(f);
 WHILE cabeza <> NIL DO
 BEGIN
 p:=cabeza;
 IF p^.valor.dni <> '-1'
 THEN
 Write(f,p^.valor);
 cabeza:=p^.sucesor;
 Dispose(p);
 END;
 END;
 END;
END.

```

Veamos un ejemplo de un programa que utiliza la *Unit* anterior.

## EJERCICIOS RESUELTOS

```

PROGRAM Encuesta (input,output,muestra);
Uses crt,fichero;
VAR
 muestra:Archivo;
 clavereg:clave;
 nombrefich:string[20];
 car:char;
 flag:boolean;

PROCEDURE Menu (VAR opcion:char);
BEGIN
 REPEAT
 Clrscr;
 Writeln (' Elija una de las siguientes opciones ');
 Writeln;
 Writeln (' 1. Modificar registro ');
 Writeln (' 2. Borrar registro ');
 Writeln (' 3. Isertar registro ');
 Writeln (' 4. Ver registro ');
 Writeln (' 5. Listar fichero ');
 Writeln (' 6. Fin ');
 Writeln;
 Write (' ¿ Opción ? ');
 Readln(opcion);
 UNTIL opcion IN ['1','2','3','4','5','6','7']
END;

{----- Programa principal -----}

BEGIN
 flag:=true;
 Inicializar;
 Clrscr;
 Write(' Nombre del fichero: ');
 Readln(nombrefich);
 Assign(muestra,nombrefich);
 Write(' ¿ Está ya creado el fichero ? ');
 Readln(car);
 IF NOT (car IN ['s','S'])
 THEN
 AbrirFichero(muestra)
 ELSE
 VolcarFichero(muestra);
 REPEAT
 Clrscr;
 menu(car);
 CASE car OF
 '1':BEGIN
 Clrscr;
 Writeln(' Introduzca clave del registro a modificar ');
 Readln(clavereg);
 ModificarRegistro(muestra,clavereg);
 END;
 '2':BEGIN
 Clrscr;
 Writeln(' Introduzca clave del registro a borrar ');
 Readln(clavereg);
 BorrarRegistro(muestra,clavereg);
 END;
 '3':InsertarRegistro(muestra);
 '4':BEGIN
 Clrscr;
 Writeln(' Introduzca clave del registro ');
 Readln(clavereg);
 VerRegistro(muestra,clavereg);
 END;
 '5':ListarFichero(muestra);

```

```

 '6':flag:=false;
 END;
 UNTIL flag=false;
 Liberar; (* Libera la memoria ocupada por las listas dinámicas *)
 SalvarFichero(muestra);
END.

```

## 12.11 EJERCICIOS PROPUESTOS

**12.22** Modificar el subprograma *Suprimir* del ejercicio resuelto 12.2 para que distinga, en caso de que no se pueda eliminar el nodo, si la causa es que la lista está vacía o que el valor a suprimir no se encuentra en la lista.

**12.23** Modificar el ejercicio resuelto 12.3, añadiendo una opción al menú para ordenar la lista alfabéticamente. Se sugiere inicializar la lista con un nombre ficticio mayor que cualquier otro, y simplificar los algoritmos de inserción y supresión de nodos, como se indica en el capítulo al hablar de listas ordenadas.

**12.24 a)** Suponer la siguiente declaración:

```

TYPE
 vector = ARRAY [1..100] OF integer;
VAR
 a: vector;
 n, x: integer;

```

a: es un vector de enteros

n: es el número de enteros que hay en a en un momento dado. Se supone que los enteros están situados en un orden creciente, y que no hay dos iguales en el vector.

Se pide:

- Escribir un procedimiento *Insertar*, que llamado por la sentencia *Insertar(a,n,x)*, inserte el entero x en el vector a, de forma que el vector siga estando ordenado. Cuando finalice su ejecución, debe devolver el nuevo valor de a y n.
- Escribir un procedimiento *Borrar*, llamado por la sentencia *Borrar(a,n,x)*, que elimine el entero x del vector a, devolviendo el nuevo valor de a y n. Si no se encuentra x en a, deberá escribir 'clave no existente'.

**b)** Sea la declaración:

```

TYPE
 elemento = ^nodo;
 nodo = RECORD
 clave: integer;
 sucesor: elemento;
 END;

```

## EJERCICIOS PROPUESTOS

```
VAR
 p: elemento;
 x, n: integer;
```

Suponer que se tenga creada una lista encadenada de  $n$  nodos ordenados según claves crecientes tal como la de la figura 12.60.

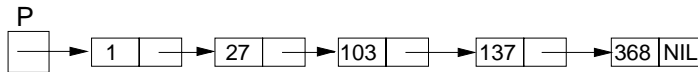


Figura 12.60 Ejemplo de lista ordenada

Se pide:

- Escribir un procedimiento *Insertar*, llamado por la sentencia `Insertar(p,n,x)`, que inserte el nodo de clave  $x$ , de modo que la lista siga estando ordenada.  $p$  es el puntero externo a la lista. Cuando finalice, se devolverá el nuevo valor de  $n$ . Si la clave  $x$  ya existía, no se realizará la inserción.
- Escribir un procedimiento *Borrar*, llamado por la sentencia `Borrar(p,n,x)`, que borre el nodo de clave  $x$  de la lista apuntada por  $p$ . Cuando finalice, devolverá el nuevo valor de  $n$ . Si no existe la clave, se escribirá 'Clave no existente'.
- c) Razonar las ventajas e inconvenientes (ocupación de memoria, tiempo de ejecución de *Insertar* y *Borrar*) de la utilización, en el mantenimiento de una lista ordenada de números enteros, de los tipos de estructuras de datos de los apartados a) y b).

**12.25** Escribir un procedimiento que escriba en orden inverso los valores clave de una lista circular formada por nodos del tipo :

```
TYPE
 nodo = ^elemento ;
 elemento = RECORD
 clave: integer ;
 sucesor: nodo
 END;
```

Por ejemplo, con la lista representada en la figura 12.61, el procedimiento debe escribir:

368 137 103 27 1

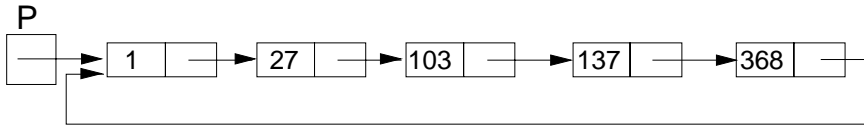
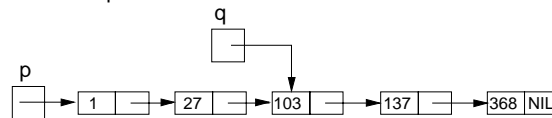


Figura 12.61 Ejemplo de lista circular

**12.26** Construir una función booleana que reciba como parámetros dos *punteros*  $p$  y  $q$  que apunten a la cabecera (primer elemento) de sendas listas. La función debe comprobar que  $q$  es una sublista de  $p$ .

Decimos que  $q$  es una *sublista* de  $p$  si  $q$  apunta a uno de los elementos de la lista de  $p$ , según se observa en la figura 12.62. Si  $p$  o  $q$  o ambos son *NIL* la función devolverá el valor *false*.

$q$  es sublista de  $p$ :



$q$  no es sublista de  $p$ :

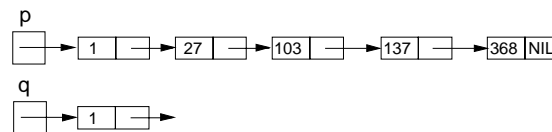


Figura 12.62 Ejemplo de sublista enlazada

**12.27** Existe un fichero cuyos registros tienen la siguiente estructura :

```
TYPE
 registro = RECORD
 pais, capital: string[20];
 END;
```

Hacer un programa que tenga las siguientes operaciones:

- *crear lista*: un procedimiento que crea una lista encadenada a partir del fichero.

Los nodos de la lista tienen la estructura:

## EJERCICIOS PROPUESTOS

```
TYPE
 puntero = ^nodo;
 nodo = RECORD
 pais, cap: string[20];
 suc : puntero;
 END;
```

- *grabar fichero*: un procedimiento que a partir de la lista crea un fichero con la información de la lista.

- *insertar nodo*: un procedimiento que inserta un nodo en la lista; se trata de una lista no ordenada, por lo que se puede insertar en cualquier parte. Se deben leer desde *input* los campos.

- *meter país*: un procedimiento que lea un país y recorra la lista, nos deberá decir la capital del país al que pertenece; si no está el país en la lista nos lo debe indicar.

- *meter capital*: un procedimiento que lea una capital y recorra la lista, nos debe decir el país al que pertenece; si no existe esa capital en la lista se debe indicar.

El programa debe presentar por pantalla al comienzo un menú con seis opciones:

- 1 - grabar fichero
- 2 - crear lista
- 3 - insertar nodo
- 4 - meter país
- 5 - meter capital
- 6 - fin

Tras introducir una clave de 1 a 5 se hace la operación correspondiente y se presenta otra vez el menú de opciones. Este proceso se repite hasta que se introduce la clave 6, en cuyo caso finaliza el programa.

**12.28** Escribir un procedimiento al que se le de un puntero  $p$  apuntando al primer elemento de una lista de claves enteras y cree otra lista encabezada por  $q$  que sólo tenga los elementos que aparecen menos de  $n$  veces en la lista inicial. No se preocupe del orden de los elementos en las listas. En la lista de salida cada clave sólo puede aparecer una vez.

La llamada será `nombre1(p, q, n)`, donde  $p$  y  $q$  serán los *punteros* cabecera a las listas de entrada y salida respectivamente y  $n$  el entero antes indicado. El procedimiento `nombre1` creará una lista auxiliar de nodos con clave y contador de veces que aparece esa clave, una vez creada esa lista a partir de la lista de entrada crearemos la lista de salida de claves que aparecen menos de  $n$  veces.

La memoria ocupada por la lista auxiliar se liberará antes de finalizar el procedimiento.

**12.29** Sea la declaración:

```

TYPE
 campo1 = RECORD
 clave1: integer;
 clave2: char;
 END;
 puntero = ^nodo;
 nodo = RECORD
 info: campo1;
 siguiente: puntero;
 END;
VAR
 cab: puntero;

```

Supongamos que existe la lista circular de la figura 12.63, con al menos un elemento:

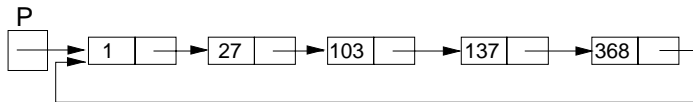


Figura 12.63 Ejemplo de lista circular

Escribir una función que, tras la llamada `suma(cab)`, nos devuelva el valor de la suma de las claves enteras que ocupan posición par y que además cumplen que `clave2`, de tipo *character*, sea una letra mayúscula.

**12.30** Se adjunta el listado del bloque principal de un programa que realiza las siguientes tareas:

- El procedimiento *Lee\_Datos* lee de un dispositivo externo una pareja de números reales (x,y), correspondientes a las características técnicas de cierta máquina en funcionamiento, y asocia a la variable booleana `enMarcha` el valor *true* si la máquina está funcionando, y *false* en caso contrario.
- Con dichas parejas de datos se crea una lista enlazada según valores crecientes de x, apuntada por la variable `inicio`.
- El subprograma *CreaFichero* crea un fichero de texto con la información de dicha lista. Cada línea del fichero contiene el carácter x seguido del campo x de un nodo de la lista, y a continuación el carácter y seguido del campo y del mismo nodo.

Se pide:

- a) Escribir la cabecera del programa y las declaraciones globales necesarias.
- b) Escribir el procedimiento *Insertar*, que añade un elemento a la lista manteniéndola ordenada. Su llamada será la que aparece en el bloque principal que se adjunta.



## AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

c) Programar el procedimiento *CreaFichero*. Su llamada será la que aparece en el bloque principal que se adjunta.

```
(* Bloque principal del programa *)
...
BEGIN
 inicio := NIL;
 Lee_Datos(x,y, enMarcha);
 WHILE enMarcha DO
 BEGIN
 Insertar(inicio, x, y);
 Lee_Datos(x,y, enMarcha);
 END;
 Writeln('LA MAQUINA SE HA PARADO');
 IF inicio <> NIL
 THEN CreaFichero(texto, inicio)
 ELSE Writeln('NO SE HA LEIDO NINGUN DATO');
END.
```

- 12.31 Construir una *unit* que implemente el tipo abstracto de datos lista simplemente encadenada con ficheros de acceso directo. Escribir un programa que use dicha *unit*.
- 12.32 Construir una *unit* que implemente el tipo abstracto de datos pila con ficheros de acceso directo. Escribir un programa que use dicha *unit*.
- 12.33 Construir una *unit* que implemente el tipo abstracto de datos cola con ficheros de acceso directo. Escribir un programa que use dicha *unit*.
- 12.34 Construir una *unit* que implemente el tipo abstracto de datos lista circular con ficheros de acceso directo. Escribir un programa que use dicha *unit*.
- 12.35 Construir una *unit* que implemente el tipo abstracto de datos árbol binario con ficheros de acceso directo. Escribir un programa que use dicha *unit*.
- 12.36 Construir una *unit* que implemente el tipo abstracto de datos lista doblemente encadenada con ficheros de acceso directo. Escribir un programa que use dicha *unit*.

## 12.12 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

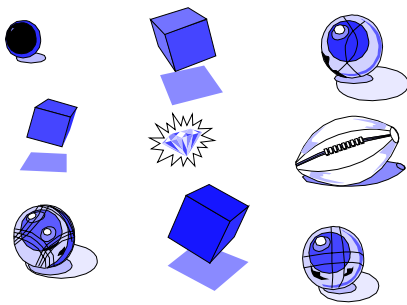
Para profundizar en el conocimiento de las estructuras dinámicas de datos, especialmente en el caso de árboles y direccionamiento *hash*, se recomienda la lectura del capítulo *Estructuras dinámicas de información* de la obra de N. Wirth titulada *Algoritmos + estructuras de datos = programas* (Ed. del Castillo, 1980).

## ESTRUCTURAS DINAMICAS DE DATOS

Para un mayor conocimiento en la construcción de tipos abstractos de datos puede consultarse la obra de *M. Collado Machuca, R. Morales Fernández, y J.J. Moreno Navarro* titulada *Estructuras de datos. Realización en Pascal* (Ed. Díaz de Santos, 1987).

Si se desea conocer mejor las estructuras dinámicas de datos no lineales, y en especial los grafos, se recomienda consultar la obra de *A. V. Aho, J.E. Hopcroft, y J. D. Ullman* titulada *Estructuras de datos y algoritmos* (Ed. Addison-Wesley Iberoamericana, 1988). También se debe consultar esta obra si se desean conocer algoritmos de gestión de memoria. En esta obra también se puede encontrar la implementación de estructuras dinámicas mediante *arrays* y ficheros.

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS



## CAPITULO 13

# PROGRAMACION ORIENTADA A OBJETOS

### CONTENIDOS

- 13.1 Introducción
- 13.2 Lenguajes orientados a objetos
- 13.3 Conceptos básicos de POO
- 13.4 Encapsulación
- 13.5 Ocultación de información
- 13.6 Herencia
- 13.7 Polimorfismo
- 13.8 Objetos dinámicos
- 13.9 Abstracción
- 13.10 Genericidad
- 13.11 Representación interna de los tipos objeto
- 13.12 Ejercicios resueltos
- 13.13 Ejercicios propuestos
- 13.14 Ampliaciones y notas bibliográficas

### 13.1. INTRODUCCION

La programación orientada a objetos (POO) tiene su origen en los modelos de simulación del mundo real. El objetivo de la programación orientada a objetos es acercar los procesos y situaciones que se producen en el mundo real al software que trata de simularlos en los ordenadores.

## LENGUAJES ORIENTADOS A OBJETOS

Por otra parte la programación orientada a objetos permite construir un software más robusto y con componentes más fácilmente reutilizables que la programación tradicional, que implica directamente un acortamiento de los tiempos y de los ciclos de producción de software.

La programación orientada a objetos es una aproximación a las metodologías de programación utilizadas en la ingeniería del software, que han demostrado importantes mejoras en la productividad de los proyectos de construcción de software. En algunos aspectos la programación orientada a objetos es un refinamiento de las técnicas de la programación estructurada, remarcando los aspectos de modularidad y de ocultación de información. La programación orientada a objetos no es una panacea, pero hace más manejable el desarrollo y mantenimiento de los programas.

Las técnicas de programación orientada a objetos tienen su principal campo de aplicación en la reutilización de software. Este software puede ser de desarrollo propio, o la utilización de bibliotecas desarrolladas por terceros. Así las bibliotecas *Turbo Vision* y *ObjectWindows*, incorporadas por Turbo Pascal y Borland Pascal, permiten utilizando técnicas de POO el desarrollo sencillo de programas en entornos DOS (*Turbo Vision*) o en entorno gráfico Windows (*ObjectWindows*).

### 13.2. LENGUAJES ORIENTADOS A OBJETOS

El primer lenguaje que incorporó explícitamente las técnicas de programación orientadas a objetos fue el *Simula 67*, desarrollado por dos científicos noruegos *Kristen Nygaard* y *Ole-Johan Dahl* en 1967. El lenguaje *Simula 67* es un superconjunto del lenguaje *ALGOL 60*, al que se le añadieron nuevas características que constituyen el núcleo principal de lo que se denomina actualmente programación orientada a objetos. Se puede decir que todos los lenguajes de programación orientados a objetos descienden de alguna forma del *Simula 67* (véase fig. 13.1).

Las extensiones del lenguaje Pascal que incorpora Turbo Pascal para soportar la programación orientada a objetos tienen sus antecesores en los lenguajes *Clascal*, *Object Pascal* y *C++*.

*Clascal* es una versión de Pascal orientado a objetos desarrollada por Apple Computer para su ordenador Lisa a principios de los años 80. Posteriormente Apple Computer lo adaptó para el entorno Macintosh en 1986, con la colaboración de *Niklaus Wirth* (autor del Pascal) dando lugar al lenguaje *Object Pascal*. El lenguaje *Object Pascal* fue el primer lenguaje soportado por el *Macintosh Programmer's Workshop (MPW)*, para el desarrollo de aplicaciones en el entorno Macintosh.

El lenguaje *C++* fue diseñado por *Bjarne Stroustrup* de AT&T en 1985. Su antecesor es el lenguaje *C con clases*, también desarrollado por *Bjarne Stroustrup* en 1980. El lenguaje *C++* es una mezcla de *C* y *Simula 67*, donde prima la eficiencia, siguiendo la pauta de su antecesor el lenguaje *C*.

El lenguaje Pascal con las extensiones orientadas a objetos de Turbo Pascal se define como un lenguaje orientado a objetos *híbrido*. Se definen los lenguajes orientados a objetos híbridos como los lenguajes que tienen tipos orientados a objetos y tipos no orientados a objetos. Habitualmente los lenguajes híbridos son lenguajes tradicionales, como el Pascal, a los que se les ha añadido los tipos y las características de los lenguajes orientados a objetos. El lenguaje orientado a objetos híbrido más extendido es el lenguaje C++.

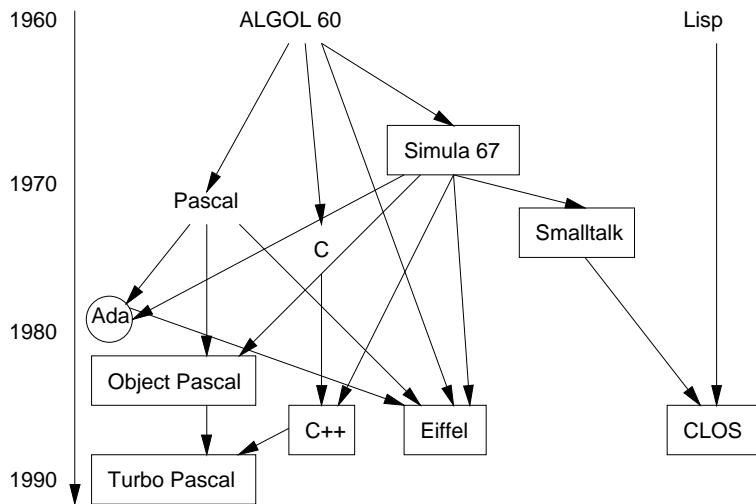


Figura 13.1 Principales relaciones en la evolución de algunos lenguajes orientados a objetos

Se denominan lenguajes orientado a objetos *puros* a los lenguajes en los cuales todos sus tipos de datos son orientados a objetos. Ejemplos de lenguajes orientados objetos puros son *Smalltalk* y *Eiffel*.

Las extensiones orientadas a objetos de Turbo Pascal tratan de mantener la compatibilidad total con el lenguaje Pascal, de ahí su definición como lenguaje híbrido. La primera versión que incluyó las extensiones (Turbo Pascal 5.5) tan sólo añadió cuatro palabras reservadas nuevas: *object*, *constructor*, *destructor* y *virtual*. Turbo Pascal 6 introdujo *private*. Turbo Pascal 7 ha incluido *public* e *inherited*.

### 13.3. CONCEPTOS BASICOS DE POO

Los conceptos que constituyen la base del paradigma de la orientación a objetos son la *encapsulación*, la *herencia*, y el *polimorfismo*.

## ENCAPSULACION

La *encapsulación* es el término formal que describe el conjunto de métodos y datos dentro de un tipo objeto, de forma que el acceso a los datos se realiza a través de los propios métodos del objeto. Los métodos son subprogramas creados para los datos del objeto. Una parte de los datos y de los métodos del objeto tan sólo son accesibles a través de sus métodos, quedando ocultos para su manejo desde el exterior (parte *privada*). Los datos y métodos que son accesibles desde el exterior son la parte *pública* del objeto. Se denomina *ocultación de información* a la restricción de acceso a la parte privada de los objetos.

La *herencia* es el mecanismo por el cual los tipos objeto pueden compartir los métodos y datos de otros objetos. Es un mecanismo potente que no se encuentra en los lenguajes de programación tradicionales. Este mecanismo permite al programador crear nuevos tipos de objetos, programando solamente las diferencias con el tipo objeto padre. La aplicación de la herencia repetidamente permite construir *jerarquías de tipos de objetos* con distintas especializaciones. Habitualmente los compiladores incluyen bibliotecas de tipos de objetos que se suelen denominar *framework* o marco de trabajo, que permiten al programador adaptarlas a sus necesidades.

El *polimorfismo* es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos, ligados entre sí por el mecanismo de herencia. Dada una jerarquía de tipos de objetos definidos con unos métodos especiales denominados *constructores*, *virtuales*, y *destructores*; el polimorfismo permite procesar objetos cuyo tipo no se conoce en tiempo de compilación.

| <b>Término en POO</b> | <b>Equivalencia en Turbo Pascal</b> |
|-----------------------|-------------------------------------|
| Clase                 | Tipo OBJECT                         |
| Objeto                | Variable de tipo OBJECT             |
| Método                | Subprograma                         |

Tabla 13.1 Cambio de notación entre POO y Turbo Pascal

Los términos básicos utilizados en POO tienen una notación en Turbo Pascal que difiere de las notaciones empleadas en otros lenguajes, tal y como se muestra en la tabla 13.1.

### 13.4. ENCAPSULACION

La encapsulación (en inglés *encapsulation*) consiste en la combinación de datos y subprogramas en un nuevo tipo de datos denominado en Turbo Pascal **tipo objeto** (*object*), que en otras notaciones se denomina *tipo abstracto de datos*, y en otros lenguajes se denomina *clase*. En un tipo objeto se define la información que contiene (datos) y los procesos que se realizan sobre los datos (métodos). Los métodos son subprogramas (procedimientos o funciones) que realizan operaciones con los datos.

En la programación convencional las estructuras de datos y los subprogramas se definían por separado. Cuando el código y los datos son entidades separadas, siempre existe el peligro de llamar al procedimiento correcto con los datos equivocados, o de llamar a un procedimiento equivocado con los datos correctos. Sin embargo en POO se define ambos dentro de los tipos *object*, que se puede representar gráficamente con el diagrama de la figura 13.2.

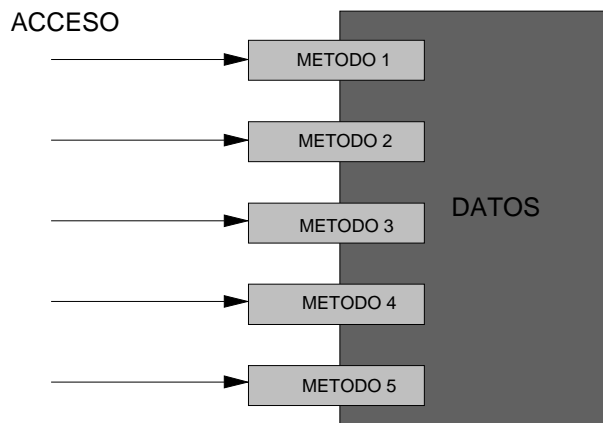


Figura 13.2 Acceso a los datos a través de los métodos

La encapsulación no consiste únicamente en la mera unión de los datos y los subprogramas, además el acceso a los datos se realiza a través de los métodos o subprogramas. Es decir no se manipulan directamente los datos, sino que se construyen métodos tanto para inicializar datos, calcular sobre los datos, modificar los datos, mostrarlos, o destruirlos. Agrupando el código y la declaración de datos juntos, se pueden evitar errores de sincronización entre ambos.

El paradigma de la programación orientada a objetos intenta representar los componentes de un problema como tipos *object*, en los cuales sus características son los datos, y su comportamiento son los métodos. Así la mayor parte de las cosas del mundo real, que nos rodea se pueden describir como objetos. Por ejemplo una botella, tiene unas características: líquido que contiene, volumen, color, textura, forma,... y unos comportamientos: introducir líquidos, agitarla, moverla, servir líquidos, vaciarla, romperla,...

## DECLARACION DE TIPOS OBJETO

La sintaxis de la declaración de tipos objeto (*object*) es muy parecida a la de *record* con la diferencia de que además de campos existen métodos (subprogramas). OBJECT es una nueva palabra reservada del lenguaje. La sintaxis simplificada de los tipos objeto en notación EBNF es la siguiente:



## ENCAPSULACION

```

<Tipo object> ::= OBJECT {<lista de campos>} {<lista de métodos>} END
<lista de campos> ::= <lista de identificadores> : <tipo> ;
<lista de métodos> ::= <cabecera procedimiento> | <cabecera función>

```

El diagrama sintáctico de la declaración más simple de un tipo OBJECT se presenta en la figura 13.3.

Los variables de tipo objeto (*object*) se pueden declarar tanto en una *Unit* como en un programa principal. Sin embargo no se permiten tipos objeto locales a un procedimiento o función.

Los tipos objeto al igual que los *record* se pueden pasar como parámetros de procedimientos y funciones, y también se pueden declarar punteros a tipos objeto.

Dentro de la declaración de un tipo objeto, una cabecera de método puede especificar parámetros del tipo objeto que está siendo declarado, aún cuando la declaración no está todavía completa.

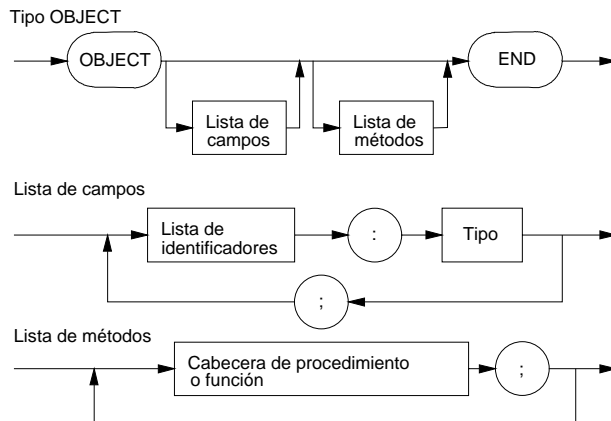


Figura 13.3 Diagrama sintáctico simplificado de la declaración de tipo objeto.

### Ejemplo 13.1: Declaración del tipo objeto *Tplaneta*

Se supone que se desea construir una estructura de datos con información sobre planetas. Se crea un tipo de datos denominado *Tplaneta* de tipo *object*. Se acostumbra a elegir identificadores que comienzan por *T* para indicar que son tipos de datos.

```

TYPE
 Tplaneta=OBJECT
 nombre:string;
 radio:real;
 masa:real;
 PROCEDURE crear(unNombre:string;unRadio:real;unaMasa:real);
 PROCEDURE modificarNombre(nuevoNombre:string);
 PROCEDURE modificarRadio(nuevoRadio:real);
 PROCEDURE modificarMasa(nuevaMasa:real);
 PROCEDURE muestraTodo;

```

## PROGRAMACION ORIENTADA A OBJETOS

```
FUNCTION LeerNombre:string;
FUNCTION LeerRadio:real;
FUNCTION LeerMasa:real;
PROCEDURE eliminar;
END;
```

### Ejemplo 13.2: Declaración del tipo objeto *TanguloSexagesimal*

Las funciones trigonométricas que incorpora el lenguaje Pascal tan sólo manejan ángulos en radianes. Se construye un nuevo tipo de datos para manejar ángulos en grados sexagesimales, por lo tanto se crea un tipo objeto con el dato y las funciones trigonométricas para ángulos sexagesimales. Las funciones trigonométricas para ángulos sexagesimales tienen identificadores en español.

```
TYPE
TanguloSexagesimal = OBJECT
 angulo:real;
 PROCEDURE crear (a:real);
 PROCEDURE crearGMS (g,m,s:integer);
 FUNCTION seno:real;
 FUNCTION coseno:real;
 FUNCTION tangente:real;
 PROCEDURE mostrar;
END;
```

### Ejemplo 13.3: Declaración del tipo objeto *Tlista*

Se desea construir un tipo abstracto de datos para manejar listas simplemente enlazadas. Se construye el tipo objeto *Tlista*, cuyo único dato es el campo *cabeza* de tipo *Pnodo*, donde *Pnodo* es un puntero a los elementos de la lista, que son de tipo *Tnodo*. Puede observarse que en este caso se maneja una lista de enteros.

Se sigue la constumbre de nombrar a los identificadores de tipos puntero anteponiendo la letra *P*, y a otros tipos de datos con la letra *T*.

```
TYPE
Pnodo=^Tnodo;
Tnodo=RECORD
 info:integer;
 sig:Pnodo;
END;

Tlista=OBJECT
 nombre:string;
 cabeza:Pnodo;
 PROCEDURE inicializar(unNombre:string);
 PROCEDURE destruir;
 PROCEDURE insetarEnCabeza(dato:integer);
 PROCEDURE eliminar(dato:integer);
 PROCEDURE mostrar;
END;
```

## ENCAPSULACION

### IMPLEMENTACION DE LOS METODOS

La implementación de los métodos de los tipos *object* se realiza de igual forma que los procedimientos y funciones de Pascal estándar, con la diferencia de que es necesario colocar el nombre del tipo *object* seguido de un punto y el nombre del método, como si se tratase del acceso a un campo de un *record*. Los parámetros del procedimiento o función no es necesario colocarlos en la implementación dado que es obligatorio su definición dentro del tipo *object*, sin embargo se permite volverlos a escribir de forma redundante en la implementación del método. Así por ejemplo en el tipo *TanguloSexagesimal* se construyen los métodos:

```
CONST
 gradosAradianes= Pi/180; (* Pi es una función que devuelve el valor de π *)

PROCEDURE TanguloSexagesimal.crear;
BEGIN
 angulo:=a*gradosAradianes;
END;

PROCEDURE TanguloSexagesimal.crearGMS;
BEGIN
 angulo:=(g+m/60+s/3600)*gradosAradianes;
END;

FUNCTION TanguloSexagesimal.seno;
BEGIN
 seno:=Sin(angulo);
END;

PROCEDURE TanguloSexagesimal.mostrar;
BEGIN
 Writeln(angulo*180/Pi:5:3);
END;
```

Puede observarse que los métodos se programan para que los usuarios de dichos métodos accedan a los datos a través de los métodos, y no directamente a los campos del tipo *object* (véase figura 13.7).

### INSTANCIACION DE OBJETOS

Se dice que se realiza una *instanciación* de un objeto cuando se declara una variable de tipo objeto. El objeto resultante se denomina *instancia* del tipo objeto. Habitualmente a las instancias se las denomina *objetos*. Por ejemplo dadas las siguientes declaraciones de variables:

```
VAR
 a:TanguloSexagesimal;
 unaLista:Tlista;
```

*a* es una instancia del tipo objeto *TanguloSexagesimal* y *unaLista* es una instancia del tipo objeto *Tlista*.

La instanciación de un objeto no inicializa al objeto, al igual que la declaración de una variable no inicializa la variable. Habitualmente se construye un método que se suele denominar *crear* o *inicializa* para inicializar el objeto (en inglés a estos métodos se les suele denominar *init*). En el apartado de este capítulo titulado *polimorfismo* se explicará que los tipos objeto que tienen unos métodos especiales, denominados *virtuales*, deben inicializarse obligatoriamente con los métodos denominados *constructores*.

## ACTIVACION DE METODOS

Un método se activa a través de una sentencia de llamada a una función o procedimiento, precedida de la variable de tipo objeto correspondiente y un punto, tal y como se muestra en el diagrama sintáctico de la figura 13.4.

Activación de métodos

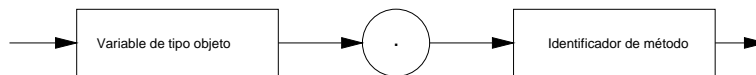


Figura 13.4 Diagrama sintáctico de activación de métodos.

Los métodos se pueden activar llamándolos como a cualquier subprograma, con sólo anteponer el identificador de la variable de tipo objeto. Así continuando con el ejemplo de *TanguloSexagesimal* se muestra como se utilizan.

```

VAR
 a:TanguloSexagesimal;

BEGIN
 a.crear(45);
 Writeln(a.seno:5:3); (* 0.707 *)
 Writeln(a.coseno:5:3); (* 0.707 *)
 Writeln(a.tangente:5:3); (* 1.000 *)
 a.mostrar; (* 45.000 *)

```

## USO DE WITH CON OBJETOS

Al igual que con el tipo registro también se permite el uso de la instrucción *WITH* con el tipo objeto. Así continuando con el ejemplo de *TanguloSexagesimal* se muestra como se utiliza *WITH*.

```

WITH a DO
 BEGIN
 crear(30);
 Writeln(seno:5:3); (* 0.500 *)
 Writeln(coseno:5:3); (* 0.866 *)
 Writeln(tangente:5:3); (* 0.577 *)
 mostrar; (* 30.000 *)
 END;

```

## ENCAPSULACION

### AMBITO DE LOS METODOS Y EL PARAMETRO *SELF*

Aunque dentro del código fuente el cuerpo del tipo objeto y los cuerpos de los métodos están separados, sin embargo comparten el mismo ámbito. Es decir los campos de datos y los métodos pueden utilizarse libremente dentro de la definición de los métodos del objeto. Existe una instrucción *WITH* implícita que enlaza el tipo objeto y sus métodos dentro del mismo ámbito. Además de esta instrucción *with* implícita se pasa un parámetro invisible al método cada vez que se le llama. Este parámetro se denomina *Self*, y es un puntero de 32 bits a la instancia del objeto al cual el método efectuó la llamada. Así por ejemplo el método *mostrar*, también se podría escribir de la siguiente forma:

```
PROCEDURE TanguSexagesimal.mostrar(VAR Self:TanguSexagesimal);
BEGIN
 Writeln(Self.angulo*180/Pi:5:3);
END;
```

Habitualmente el conocimiento del parámetro *Self* no es necesario, dado que Turbo Pascal genera el código de forma automática. Solamente es necesario su conocimiento si se construyen métodos externos en lenguaje ensamblador, o en el manejo de las bibliotecas *Turbo Vision* y *ObjectWindows*.

### CAMPOS DE DATOS Y PARAMETROS FORMALES DE LOS METODOS

Una consecuencia del hecho de que los métodos y los tipos objeto compartan el mismo ámbito, es que un parámetro formal de un método no puede ser idéntico a ninguno de los campos de datos del tipo objeto. Así por ejemplo **no está permitido** el fragmento de código siguiente. Si se intentase compilar daría un error de compilación, indicando la duplicidad de un identificador.

```
TYPE
TanguSexagesimal = OBJECT
 angulo:real;
 PROCEDURE crear (angulo:real); (* Error 4: identificador duplicado *)
 PROCEDURE crearGMS (g,m,s:integer);
 FUNCTION seno:real;
 FUNCTION coseno:real;
 FUNCTION tangente:real;
 PROCEDURE mostrar;
END;
```

Esta no es una nueva restricción impuesta por las extensiones de Pascal orientado a objetos, sino que es debido a las reglas del ámbito del Pascal estándar. Es por lo mismo que no está permitido llamar a un parámetro formal de un subprograma igual que a una variable local. Las variables locales de un procedimiento y función comparten el mismo ámbito que sus parámetros formales, y por eso no pueden denominarse igual. Es decir no está permitido el siguiente fragmento de programa:

## PROGRAMACION ORIENTADA A OBJETOS

```
PROCEDURE eliminarLista(valor:integer);
VAR
 valor:integer; (* Error 4: identificador duplicado *)
BEGIN
 ...
```

Se puede concluir que las extensiones orientadas a objetos incorporadas no modifican las reglas del ámbito del Pascal estándar.

### Ejemplo 13.4: Manejo del tipo *TanguloSexagesimal*

A continuación se muestra un programa completo, como ejemplo de encapsulación con tipos objeto. Puede observarse que antes de usar cada *instancia* del tipo *TanguloSexagesimal*, se utiliza el método *crear* para inicializarla, de no utilizarse el campo *angulo*, tendría un valor no determinado. Para acceder al campo *angulo*, siempre se hace a través de un método y nunca directamente (mal hábito). También se ilustra el uso de la instrucción *with* con métodos. En los comentarios se muestran los valores que resultan de la ejecución del programa.

```
PROGRAM Sexagesimal (Output);

CONST
 gradosAradianes=Pi/180;

TYPE
 TanguloSexagesimal = OBJECT
 angulo:real;
 PROCEDURE crear (a:real);
 PROCEDURE crearGMS (g,m,s:integer);
 FUNCTION seno:real;
 FUNCTION coseno:real;
 FUNCTION tangente:real;
 PROCEDURE mostrar;
 END;

PROCEDURE TanguloSexagesimal.crear;
BEGIN
 angulo:=a*gradosAradianes;
END;

PROCEDURE TanguloSexagesimal.crearGMS;
BEGIN
 angulo:=(g+m/60+s/3600)*gradosAradianes;
END;

FUNCTION TanguloSexagesimal.seno;
BEGIN
 seno:=Sin(angulo);
END;

FUNCTION TanguloSexagesimal.coseno;
BEGIN
 coseno:=Cos(angulo);
END;

FUNCTION TanguloSexagesimal.tangente;
BEGIN
 tangente:=Sin(angulo)/Cos(angulo)
END;
```

## ENCAPSULACION

```
PROCEDURE TanguloSexagesimal.mostrar;
BEGIN
 Writeln(angulo*180/Pi:5:3);
END;

VAR
 a,b,c:TanguloSexagesimal;

BEGIN
 a.crear(45);
 Writeln(a.seno:5:3); (* 0.707 *)
 Writeln(a.coseno:5:3); (* 0.707 *)
 Writeln(a.tangente:5:3); (* 1.000 *)
 a.mostrar; (* 45.000 *)

 WITH b DO
 BEGIN
 crear(30);
 Writeln(seno:5:3); (* 0.500 *)
 Writeln(coseno:5:3); (* 0.866 *)
 Writeln(tangente:5:3); (* 0.577 *)
 mostrar; (* 30.000 *)
 END;

 c.crearGMS(40,20,10);
 WITH c DO
 BEGIN
 Writeln(seno:5:3); (* 0.647 *)
 Writeln(coseno:5:3); (* 0.762 *)
 Writeln(tangente:5:3); (* 0.849 *)
 mostrar; (* 40.336 *)
 END;

END.
```

### Ejemplo 13.5: Manejo del tipo *Tlista*

Se presenta un programa completo para el manejo del tipo *Tlista*. Se construye un programa que maneja listas de enteros. Uno de los campos del tipo *Tlista* es el puntero *cabeza* (cabeza de lista), que apunta a *Tnodo*, que es el *record* que contiene los elementos de la lista. Puede observarse que se utilizan declaraciones redundantes en los métodos, es decir se definen doblemente los parámetros de los métodos en el tipo objeto y en su implementación. Los métodos que se implementan son las operaciones más habituales con una lista: *inicializar* (crea la cabeza de lista y le da un nombre), *destruir* (elimina todos los elementos de la lista), *insertarEnCabeza* (inserta un elemento en cabeza de lista), *eliminar* (busca un elemento en la lista, y si lo encuentra lo elimina), y *mostrar* (recorre toda la lista escribiendo sus elementos). En el programa principal se hacen dos instancias, *unaLista* y *otraLista*, al tipo *Tlista*. Sin embargo la instancia *unaLista* se emplea en dos listas, es decir se reinicializa.

```
PROGRAM ManejaListas (Output);

TYPE
 Pnodo=^Tnodo;
 Tnodo=RECORD
 info:integer;
 sig:Pnodo;
 END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```
Tlista=OBJECT
 nombre:string;
 cabeza:Pnodo;
 PROCEDURE inicializar(unNombre:string);
 PROCEDURE destruir;
 PROCEDURE insertarEnCabeza(dato:integer);
 PROCEDURE eliminar(dato:integer);
 PROCEDURE mostrar;
END;

PROCEDURE Tlista.inicializar(unNombre:string);
BEGIN
 cabeza:=NIL;
 nombre:=unNombre;
 Writeln('Creada la lista: ',nombre);
END;

PROCEDURE Tlista.destruir;
VAR
 p:Pnodo;
BEGIN
 WHILE cabeza<>NIL DO
 BEGIN
 p:=cabeza;
 cabeza:=p^.sig;
 Dispose(p);
 END;
 Writeln('Destruída la lista: ', nombre);
END;

PROCEDURE Tlista.insertarEnCabeza(dato:integer);
VAR
 p:Pnodo;
BEGIN
 New(p);
 p^.info:=dato;
 p^.sig:=cabeza;
 cabeza:=p;
 Writeln('Insertado en cabeza de ',nombre, ' el valor ',dato);
END;

PROCEDURE Tlista.eliminar(dato:integer);
VAR
 p,q:Pnodo;
BEGIN
 p:=cabeza;
 WHILE p<>NIL DO
 IF p^.sig^.info=dato
 THEN
 BEGIN
 q:=p^.sig;
 p^.sig:=q^.sig;
 Dispose(q);
 Writeln('Eliminado de la lista ',nombre,' el valor ',dato);
 END
 ELSE
 p:=p^.sig;
 END;
END;
```



## ENCAPSULACION

```
PROCEDURE Tlista.mostrar;
VAR
 p:Pnodo;
BEGIN
 p:=cabeza;
 Writeln ('Mostrando la lista ', nombre, ' desde la cabeza');
 IF p=NIL THEN Writeln('Lista vacia');
 WHILE p<>NIL DO
 BEGIN
 Writeln(p^.info);
 p:=p^.sig;
 END;
 END;

VAR
 unaLista, otraLista:Tlista;

BEGIN (* Programa principal *)

 unaLista.inicializar('A');

 unaLista.insertarEnCabeza(10);
 unaLista.insertarEnCabeza(20);
 otraLista.inicializar('B');
 unaLista.insertarEnCabeza(30);
 otraLista.insertarEnCabeza(1);
 otraLista.insertarEnCabeza(2);

 unaLista.mostrar;
 unaLista.destruir;
 unaLista.mostrar;
 otraLista.mostrar;

 unaLista.inicializar('C');
 unaLista.insertarEnCabeza(11);
 unaLista.insertarEnCabeza(22);
 unaLista.insertarEnCabeza(33);
 unaLista.mostrar;

 unaLista.eliminar(22);
 unaLista.mostrar;
 otraLista.mostrar;

END.
```

En este ejemplo es una primera aproximación al manejo del tipo abstracto lista, que se refinará en ejercicios sucesivos. El resultado de la ejecución del programa anterior puede observarse a continuación:

```
Creada la lista: A
Insertado en cabeza de A el valor 10
Insertado en cabeza de A el valor 20
Creada la lista: B
Insertado en cabeza de A el valor 30
Insertado en cabeza de B el valor 1
Insertado en cabeza de B el valor 2
Mostrando la lista A desde la cabeza
30
20
10
Destruída la lista: A
Mostrando la lista A desde la cabeza
Lista vacia
```

```

Mostrando la lista B desde la cabeza
2
1
Creada la lista: C
Insertado en cabeza de C el valor 11
Insertado en cabeza de C el valor 22
Insertado en cabeza de C el valor 33
Mostrando la lista C desde la cabeza
33
22
11
Eliminado de la lista C el valor 22
Mostrando la lista C desde la cabeza
33
11
Mostrando la lista B desde la cabeza
2
1

```

### 13.5. OCULTACION DE INFORMACION

La ocultación de información, también denominada protección de la información (en inglés *data hiding*), restringe el acceso a una parte de los datos y métodos que se denominarán privados. La ocultación de información deja invisibles desde afuera los detalles de la implementación interna y la disposición de los datos. La manipulación de los campos de datos del tipo objeto sólo podrá realizarse a través de los métodos públicos.

La ocultación de la información es una parte de la propiedad de la POO denominada *encapsulación*.

En el desarrollo de aplicaciones es habitual separar el programa en distintos subprogramas o en módulos (*Unit* en Turbo Pascal), reduciéndose el número de variables globales, y utilizando variables locales para ocultar información entre módulos. Estas técnicas utilizadas en la programación estructurada se refuerzan en la POO, dado que en el mecanismo de encapsulación se pueden definir partes públicas y privadas. Una ilustración habitual de este concepto es un *iceberg*, en el cual sólo sale a la superficie la parte pública o visible, y permanece sumergida la parte privada (fig. 13.5).

La combinación de la encapsulación (con ocultación de la información) y la modularidad en las *units* de Turbo Pascal, hacen de las *units* una herramienta imprescindible para la construcción de aplicaciones informáticas con Turbo Pascal, y son de gran ayuda para vencer la complejidad del software.

Los componentes de un tipo objeto son públicas por defecto, es decir puede accederse a ellas sin ningún tipo de restricción. Se puede indicar explícitamente las partes públicas de un tipo objeto empleando la palabra reservada *PUBLIC*. Para indicar las partes privadas se emplea la palabra reservada *PRIVATE*.

La definición de tipo objeto con *PUBLIC* y *PRIVATE* en notación EBNF es la siguiente:

## OCULTACION DE INFORMACION

```
<Tipo OBJECT> ::= OBJECT
 <lista de componentes>
 {<sección de componentes>}
 END

<lista de componentes> ::= {<lista de campos>} {<lista de métodos>}
<sección de componentes> ::= (PUBLIC | PRIVATE) <lista de componentes>
<lista de campos> ::= <lista de identificadores> : <tipo> ;
<lista de métodos> ::= <cabecera procedimiento> | <cabecera función>
```

### PUBLICO

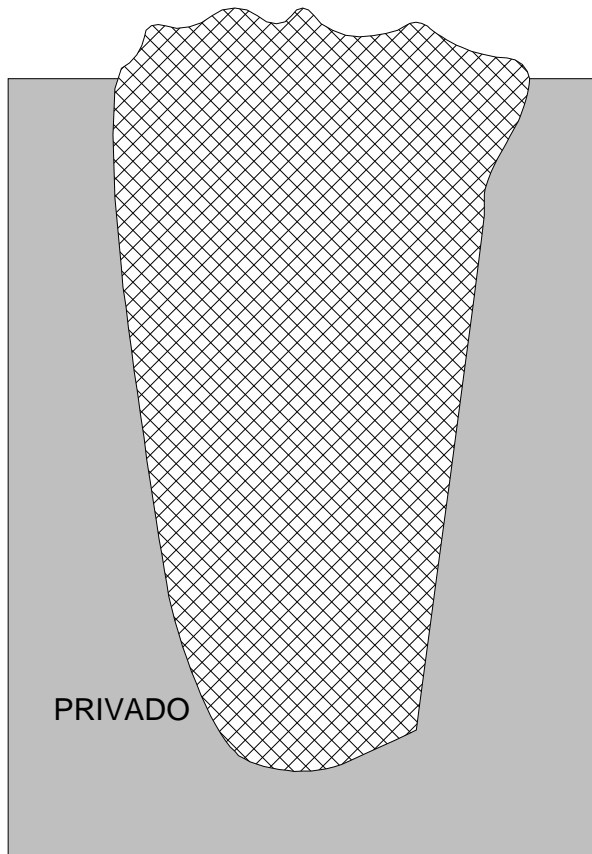


Figura 13.5 Iceberg mostrando la parte visible (pública) y ocultando la sumergida (privada).

El diagrama sintáctico de la declaración de tipos objeto con las secciones públicas y privadas se muestran en la figura 13.6.

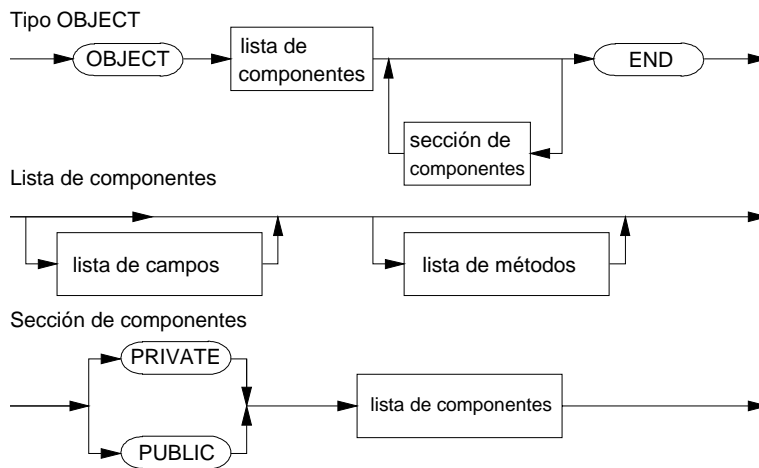


Figura 13.6 Diagrama sintáctico de tipo objeto, incorporando la secciones PUBLIC y PRIVATE

Así en el ejemplo de la declaración del tipo *TanguloSexagesimal*, del apartado anterior, todo es público por defecto, y se podría acceder al dato *angulo* directamente, sin emplear el método *mostrar*:

```
VAR a:TanguloSexagesimal;
...
BEGIN
...
Writeln(a.angulo); (* en radianes *)
...
```

Pero el resultado será en radianes, en contra de lo que podía parecer si no se examina detenidamente la implementación. Para evitar este tipo de accesos que pueden ser difíciles de controlar en grandes proyectos de programación en los que intervienen distintos equipos de trabajo, es necesario definir explícitamente la parte que es privada y la parte pública en la declaración del tipo objeto. De esta forma se consigue controlar las partes que el usuario de un tipo objeto utilizará. Este concepto se ha representado gráficamente tal y como se muestra en la figura 13.7.

```
TYPE
TanguloSexagesimal = OBJECT
PRIVATE
 angulo:real;
PUBLIC
 PROCEDURE crear (a:real);
 PROCEDURE crearGMS (g,m,s:integer);
 FUNCTION seno:real;
 FUNCTION coseno:real;
```

## OCULTACION DE INFORMACION

```
FUNCTION tangente:real;
PROCEDURE mostrar;
END;
```

En este caso si se deseara acceder directamente se obtendría un error de compilación en el caso de que estuviera en otro módulo (*Unit* en Turbo Pascal). En la figura 13.7 se pretende ilustrar como se debe pasar a través de los métodos para llegar a los datos privados.

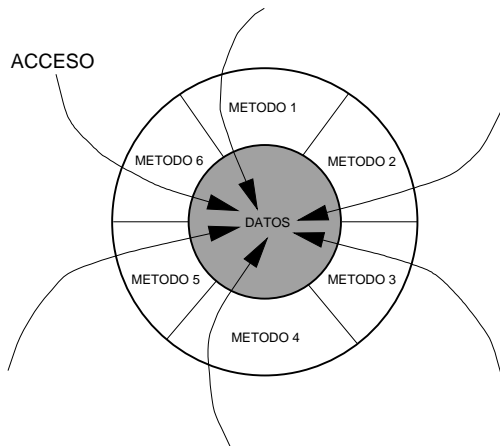


Figura 13.7 Acceso a los datos a través de los métodos.

En el desarrollo de proyectos software los tipos objeto se construyen dentro de distintos módulos, que son utilizados por diferentes equipos de programadores. Cada equipo maneja solamente los datos y métodos dejados como públicos por otros equipos, y mantiene los suyos propios (privados). La ocultación de la representación interna de los datos de un tipo objeto tiene la ventaja de que se pueden cambiar, sin que los módulos externos que usan ese tipo objeto tengan que ser modificados. En el ciclo de vida de un programa es posible que los datos estén almacenados en un principio como un array, pero posteriormente (quizá al crecer el ámbito de la aplicación crece el volumen de datos) y deba de representarse la información en una lista, un árbol binario o en un fichero. Si el tipo objeto está perfectamente encapsulado y con su estructura interna oculta, un cambio de su estructura interna no afecta al uso del tipo objeto. El interface del objeto permanece completamente igual, permitiendo al programador afinar el comportamiento de un objeto sin estropear cualquier código que utilice el tipo objeto. Esta característica de la POO permite facilitar el mantenimiento y la extensibilidad del software.

## TIPOS OBJETO Y UNITS

Se pueden definir tipos objeto dentro de *units*, con la declaración del tipo objeto en la sección de *interface* de la *unit*, y los cuerpos de los métodos en la sección de *implementation*. Es decir las *unit* pueden exportar tipos objeto.

Las *units* pueden tener sus propias definiciones de tipos objeto privados en la sección de *implementation*, y estos tipos están sujetos a las mismas restricciones que el resto de los tipos definidos en la sección de *implementation* de una *unit*.

Un tipo objeto definido en la sección de *interface* de una *unit* puede tener tipos objeto descendientes definidos en la sección *implementation*.

### Ejemplo 13.6: Unit grados

Las palabras reservadas *PUBLIC* y *PRIVATE* sólo actúan si los accesos se realizan desde distintos módulos. Para comprobar su funcionamiento se vuelve al ejemplo del tipo *TanguloSexagesimal*, se declara privado *angulo* y todos los métodos se declaran públicos. Todo esto se construye en una *unit*. Todo módulo que use esta *unit* no podrá acceder al campo *angulo* directamente, sino que tendrá que hacerlo a través de los métodos.

```
UNIT grados;

(* grados.pas *)

INTERFACE

CONST
 gradosAradianes=Pi/180;

TYPE
 TanguloSexagesimal = OBJECT
 PRIVATE
 angulo:real;
 PUBLIC
 PROCEDURE crear (a:real);
 PROCEDURE crearGMS (g,m,s:integer);
 FUNCTION seno:real;
 FUNCTION coseno:real;
 FUNCTION tangente:real;
 PROCEDURE mostrar;
 END;

IMPLEMENTATION

PROCEDURE TanguloSexagesimal.crear;
BEGIN
 angulo:=a*gradosAradianes;
END;

PROCEDURE TanguloSexagesimal.crearGMS;
BEGIN
 angulo:=(g+m/60+s/3600)*gradosAradianes;
END;

FUNCTION TanguloSexagesimal.seno;
BEGIN
 seno:=Sin(angulo);
END;

FUNCTION TanguloSexagesimal.coseno;
BEGIN
 coseno:=Cos(angulo);
END;
```

## HERENCIA

```
FUNCTION TanguloSexagesimal.tangente;
BEGIN
 tangente:=Sin(angulo)/Cos(angulo)
END;

PROCEDURE TanguloSexagesimal.mostrar;
BEGIN
 Writeln(angulo*180/Pi:5:3);
END;

END.
```

### Ejemplo 13.7: Uso de la unit grados

A continuación se vuelve a escribir el programa *Sexagesimal*, pero con la *Unit grados*. Cualquier acceso al campo privado *angulo*, dará un error en tiempo de compilación, señalando que no conoce dicho campo del tipo *TanguloSexagesimal*.

```
PROGRAM Sexagesimal2 (Output);

USES grados;

VAR
 a,b,c:TanguloSexagesimal;

BEGIN
 a.crear(45);
 Writeln(a.seno:5:3);
 Writeln(a.coseno:5:3);
 Writeln(a.tangente:5:3);
 (* Writeln(a.angulo); *) (* Indicará un error en tiempo de compilación *)
 a.mostrar;

 WITH b DO
 BEGIN
 crear(30);
 Writeln(seno:5:3);
 Writeln(coseno:5:3);
 Writeln(tangente:5:3);
 mostrar;
 END;

 c.crearGMS(40,20,10);
 WITH c DO
 BEGIN
 Writeln(seno:5:3);
 Writeln(coseno:5:3);
 Writeln(tangente:5:3);
 mostrar;
 END;

END.
```

## 13.6. HERENCIA

La *herencia* es un mecanismo que permite a un tipo objeto heredar propiedades (datos y métodos) de otros tipos objeto. Así la herencia permite que los programas desarrollados puedan tener otra de las propiedades de la POO: *la extensibilidad*. El mecanismo más parecido a la herencia

## PROGRAMACION ORIENTADA A OBJETOS

en programación estructurada son los *registros anidados*. El problema de los registros anidados es que tan sólo permiten extender los datos, y además exigen el uso de instrucciones *with* anidadas o gran cantidad de puntos y nombres de campos para el acceso a los campos más interiores del anidamiento. Así por ejemplo se definen los siguientes registros anidados:

```
TYPE
 Ttiempo = RECORD
 hora:0..24;
 minuto:0..59;
 segundos:0..59;
 END;
 TtiempoD = RECORD
 tiempo:Ttiempo;
 decimas:0..9;
 END;
 TfechaHms= RECORD
 dia:1..31;
 mes:1..12;
 any:0..5000;
 tiempo:Ttiempo;
 END;
 TtiempoC = RECORD
 tiempo:TtiempoD;
 centesimas:0..99;
 END;
 ...
```

[PICTURE]

Figura 13.8 Registros extensibles

El esquema de anidamiento de los registros anteriores se muestra en la figura 13.8. Utilizando las declaraciones anteriores, si se declaran las variables denominadas *tExacto* y *fExacta*, para inicializarlas serían necesarias las siguientes instrucciones:

```
VAR
 tExacto: TtiempoC;
 fExacta: TfechaHms;
 ...
WITH tExacto DO
 BEGIN
 hora:=10;
 minuto:=0;
 segundo:=0;
 WITH TtiempoD DO
 BEGIN
 decimas:=0;
 WITH TtiempoC DO centesimas:=0;
 END;
 END;
 ...
WITH fExacta DO
 BEGIN
 dia:=29;
 mes:=3;
 any:=1958;
 WITH Ttiempo DO
 BEGIN
 hora:= 18;
 minuto := 27;
```



## HERENCIA

```
 segundo := 27;
 END;
END;
...
```

Si se utiliza el mecanismo de herencia que incorporan los tipos objeto el programa anterior sería el siguiente:

```
TYPE
 Ttiempo = OBJECT
 hora:0..24;
 minuto:0..59;
 segundos:0..59;
 END;
 TtiempoD = OBJECT(Ttiempo)
 decimas:0..9;
 END;
 TfechaHms= OBJECT(Ttiempo)
 dia:1..31;
 mes:1..12;
 any:0..5000;
 END;
 TtiempoC = OBJECT(TtiempoD)
 centesimas:0..99;
 END;
...
```

Con el esquema de herencia de los tipos objeto anteriores se pueden inicializar las instancias denominadas `tExacto` y `fExacta` con las siguientes instrucciones:

```
VAR
 tExacto: TtiempoC;
 fExacta: TfechaHms;
...
WITH tExacto DO
 BEGIN
 hora:=10;
 minuto:=0;
 segundo:=0;
 decimas:=0;
 centesimas:=0;
 END;
...
WITH fExacta DO
 BEGIN
 dia:=29;
 mes:=3;
 any:=1958;
 hora:= 18;
 minuto := 27;
 segundo := 27;
 END;
...
```

Como conclusión se puede indicar que los registros son un caso particular de los tipos objeto, con solamente datos, y además todos los datos son públicos. La extensibilidad con registros anidados se mejora con la herencia de los tipos objeto, dado que se puede aplicar tanto a datos como a métodos tal y como se estudia a continuación.

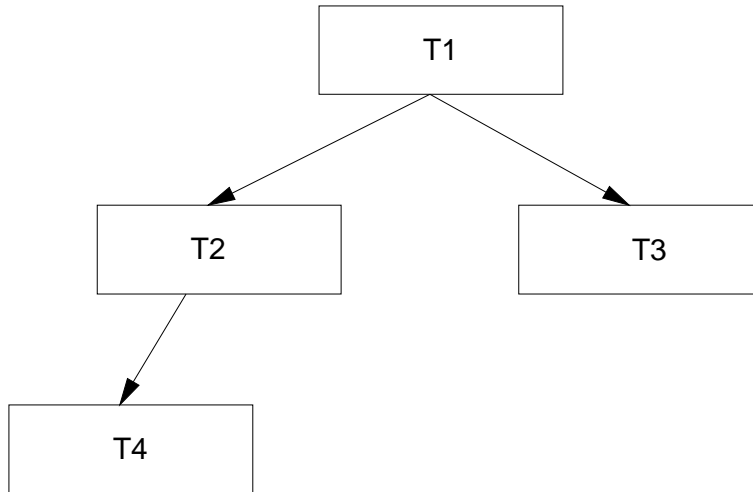
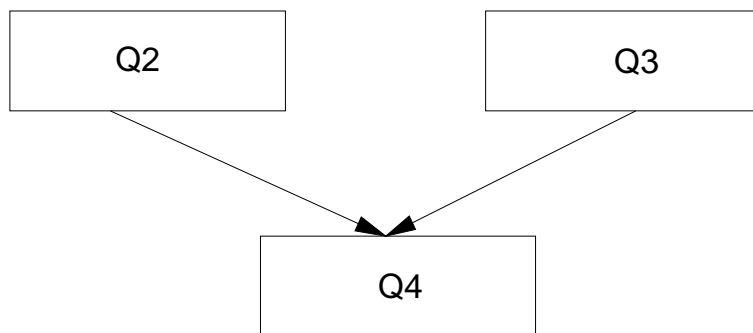


Figura 13.9 Esquema de herencia y jerarquía de tipos *object*

Se van a definir varios términos ligados al concepto de herencia, manejando el esquema de la figura 13.9. Un tipo objeto T2 puede *heredar* los componentes de otro tipo objeto T1. Si T2 hereda de T1, entonces T2 es un *descendiente* de T1, y T1 es un *ascendiente* o *antepasado* de T2.

La herencia tiene la *propiedad transitiva*; es decir si T4 hereda de T2, y T2 hereda de T1, entonces T4 también hereda de T1. El *dominio* de trabajo de un tipo objeto es él mismo y todos sus descendientes.

Sea una *unit* U2 que use otra *unit* U1, la *unit* U2 puede también definir tipos descendientes de cualquier tipo *object* definido en la sección *interface* de la *unit* U1.



## HERENCIA

Figura 13.10 Esquema de herencia múltiple (no permitida en Turbo Pascal 7)<sup>23</sup>

Turbo Pascal 7 tan sólo implementa la *herencia simple*, es decir un tipo objeto puede tener cualquier número de descendientes inmediatos, pero sólo un antepasado inmediato. Es decir un tipo objeto tan sólo puede heredar directamente de un sólo tipo objeto. La *herencia múltiple* (fig. 13.10) permitiría heredar de más de un tipo objeto simultáneamente, pero no esta implementada en Turbo Pascal 7<sup>24</sup>.

La sintaxis del tipo *object* con herencia se puede mostrar por medio del diagrama sintáctico de la figura 13.11, o también mediante la notación EBNF siguiente:

```
<Tipo OBJECT> ::= OBJECT (<herencia>|<vacío>)
 <lista de componentes>
 {<sección de componentes>}
 END

<herencia> ::= "(" <identificador de tipo OBJECT> ")"

<lista de componentes> ::= {<lista de campos>} {<lista de métodos>}

<sección de componentes> ::= (PUBLIC | PRIVATE) <lista de componentes>

<lista de campos> ::= {<lista de identificadores> : <tipo> ;}

<lista de métodos> ::= {<cabecera procedimiento> | <cabecera función>}
```

Una parte importante de la POO es el diseño de la jerarquía de tipos objeto. Es necesario observar la aplicación a realizar y elegir los tipos objeto más generales, e ir construyendo una especie de árbol genealógico, para ir logrando los tipos objeto más especializados.

---

<sup>23</sup> Borland ya ha anunciado que incorporará la herencia múltiple en las siguientes versiones de sus compiladores de Pascal

<sup>24</sup> El lenguaje Object Pascal diseñado por N. Wirth no incorpora herencia múltiple, pero ya existen compiladores en el mercado como el TopSpeed Pascal que la soportan. Borland parece que seguirá el camino del lenguaje C++ para incorporar la herencia múltiple.

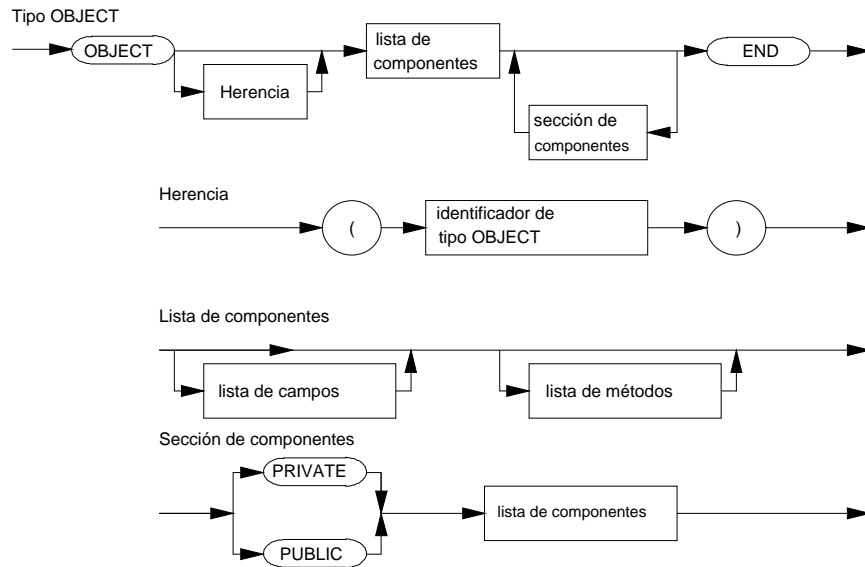


Figura 13.11 Diagrama sintáctico del tipo *object*, incluyendo la herencia

### Ejemplo 13.8

Se define un tipo *Tvehículo* que es muy general, y que pretende describir algunas de las características comunes de todos los vehículos de carretera que se van a usar en una aplicación. El resto de los tipos que heredan de *Tvehículo* van a ser especializaciones de este tipo, así el tipo *Tcoche* va a ser una especialización del tipo *Tvehículo* al que se le van a añadir alguna característica particular de los coches. Por otra parte también se puede definir un tipo *Tcamion* que se especializa con algunas características particulares de los camiones (véase fig. 13.12).

```

TYPE
Tvehiculo= OBJECT
PRIVATE
matricula:string[10];
marcaModelo:string[25];
cv:integer;
PROCEDURE crear(mt,mr:string;c:integer);
PROCEDURE mostrar;
END;

Tcoche= OBJECT (Tvehiculo)
PRIVATE
pasajeros:integer;
PUBLIC
PROCEDURE crear(mt,mr:string;c,pasa:integer);
PROCEDURE mostrar;
END;

```

## HERENCIA

```
Tcamion= OBJECT (Tvehiculo)
PRIVATE
 tara:real;
 carga:real;
 ejes:integer;
PUBLIC
 PROCEDURE crear(mt,mr:string;c:integer;t,cr:real;e:integer);
 PROCEDURE mostrar;
END;
```

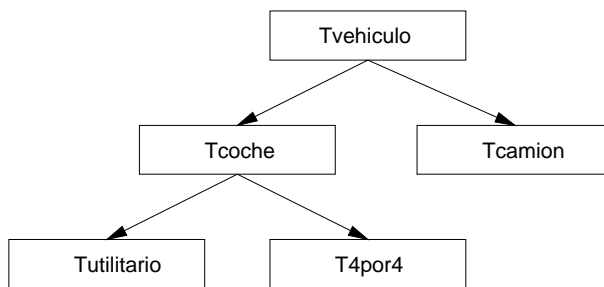


Figura 13.12 Jerarquía de los tipos derivados de *Tvehiculo*

La herencia se puede aplicar otra vez al tipo *Tcoche* para lograr tipos más especializados como son *Tutilitario* y *T4por4*. Se obtiene una *jerarquía de tipos objeto*, representada en la figura 13.12, que se puede obtener directamente en el entorno integrado de desarrollo de Turbo Pascal (IDE), usando el inspector de tipos objeto (*browser*).

```
Tutilitario= OBJECT (Tcoche)
PRIVATE
 consumoUrbano:real;
PUBLIC
 PROCEDURE crear(mt,mr:string;c,pasa:integer;consu:real);
 PROCEDURE mostrar;
END;

T4por4 =OBJECT (Tcoche)
PRIVATE
 alturaVadeo:real;
PUBLIC
 PROCEDURE crear(mt,mr:string;c,pasa:integer;av:real);
 PROCEDURE mostrar;
END;
```

## REDEFINICION DE METODOS

El mecanismo de herencia trata de especializar los tipos objeto, con nuevos tipos objeto que añaden nuevas características, es decir más datos y más métodos. En este ejemplo se puede observar que cada tipo más especializado incorpora nuevos datos, así por ejemplo *Tcoche* incorpora el campo *pasajeros* a los que ya tiene *Tvehiculo*, sin embargo los métodos son los mismos en todos los casos *crear* y *mostrar*. Es decir los métodos se redefinen.

Para *redefinir* (en inglés *override*) un método heredado, simplemente se define un nuevo método con el mismo nombre que el método heredado, pero con cuerpo diferente y (si es necesario) con un conjunto de parámetros diferente. A continuación se marcan en negrita los métodos redefinidos:

```
Tcoche= OBJECT (Tvehiculo)
PRIVATE
 pasajeros:integer;
PUBLIC
 PROCEDURE crear(mt,mr:string;c,pasa:integer);
 PROCEDURE mostrar;
END;

Tcamion= OBJECT (Tvehiculo)
PRIVATE
 tara:real;
 carga:real;
 ejes:integer;
PUBLIC
 PROCEDURE crear(mt,mr:string;c:integer;t,cr:real;e:integer);
 PROCEDURE mostrar;
END;
```

Mientras que los métodos pueden ser redefinidos, los campos de datos no. Una vez definido un campo de datos en la jerarquía de objetos, ningún tipo descendiente puede definir un campo de datos con el mismo identificador.

### USO DE *INHERITED*

En la implementación de los métodos del ejemplo de herencia puede observarse que cuando se redefine el método de su ascendiente (padre), se usa la palabra reservada *INHERITED* y el nombre del método de su antecesor (véase figura 13.13).

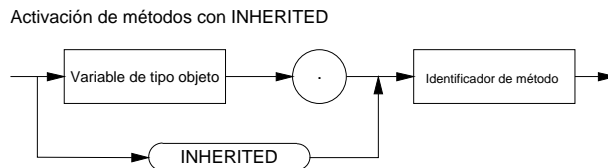


Figura 13.13 Diagrama sintáctico de uso de *inherited*

Por ejemplo para el método `mostrar` de `Tcoche` se ejecuta en primer lugar `INHERITED mostrar`, es decir el método `mostrar` de `Tvehiculo`, y posteriormente se añaden las nuevas características de `Tcoche`.

```
PROCEDURE Tcoche.mostrar;
BEGIN
 INHERITED mostrar;
 Writeln('N° de pasajeros: ',pasajeros);
END;
```

## HERENCIA

Llamando a un método redefinido, se asegura que el tipo de objeto descendiente tiene la funcionalidad de su ascendiente. Además, cualquier cambio realizado en el método ascendiente, automáticamente afecta a sus descendientes.

## METODOS ABSTRACTOS

Un método se denomina abstracto si solamente va a ser utilizado por métodos de los descendientes del tipo objeto al que pertenece, y nunca directamente desde una instrucción. Por ejemplo en el programa siguiente los métodos *crear* y *mostrar* de *Tvehiculo* no se utilizan directamente, siempre son manejados a través de los métodos redefinidos en los descendientes de *Tvehiculo*.

## TIPOS OBJETO ABSTRACTOS

Un tipo objeto se denomina abstracto si nunca va a ser utilizado directamente, y sólo se usa a través de sus descendientes. Por ejemplo el tipo objeto *Tvehiculo*. Para prevenir accesos no previstos se aconseja declarar privados todos sus campos de datos y métodos. Se volverán a tratar los tipos objetos abstractos en el apartado 13.9 de este capítulo titulado *Abstracción*.

### Ejemplo 13.9

Se desarrolla el programa completo usado en los ejemplos anteriores de herencia. Se implementan los métodos definidos en la jerarquía de los tipos objeto del ejemplo 13.8. También se ilustra el manejo de *INHERITED*.

```
PROGRAM Vehiculos (Output);

TYPE
 Tvehiculo= OBJECT
 PRIVATE
 matricula:string[10];
 marcaModelo:string[25];
 cv:integer;
 PROCEDURE crear(mt,mr:string;c:integer);
 PROCEDURE mostrar;
 END;

 Tcoche= OBJECT (Tvehiculo)
 PRIVATE
 pasajeros:integer;
 PUBLIC
 PROCEDURE crear(mt,mr:string;c,pasa:integer);
 PROCEDURE mostrar;
 END;

 Tcamion= OBJECT (Tvehiculo)
 PRIVATE
 tara:real;
 carga:real;
 ejes:integer;
 PUBLIC
```

## PROGRAMACION ORIENTADA A OBJETOS

```
PROCEDURE crear(mt,mr:string;c:integer;t,cr:real;e:integer);
PROCEDURE mostrar;
END;

Tutilitario= OBJECT (Tcoche)
PRIVATE
consumoUrbano:real;
PUBLIC
PROCEDURE crear(mt,mr:string;c,pasa:integer;consu:real);
PROCEDURE mostrar;
END;

T4por4 =OBJECT (Tcoche)
PRIVATE
alturaVadeo:real;
PUBLIC
PROCEDURE crear(mt,mr:string;c,pasa:integer;av:real);
PROCEDURE mostrar;
END;

PROCEDURE Tvehiculo.crear;
BEGIN
matricula:=mt;
marcaModelo:=mr;
cv:=c;
END;

PROCEDURE Tvehiculo.mostrar;
BEGIN
Writeln('Matrícula: ',matricula,' Marca: ',marcaModelo);
Writeln('CV: ',cv);
END;

PROCEDURE Tcoche.crear;
BEGIN
INHERITED crear(mt,mr,c);
pasajeros:=pasa;
END;

PROCEDURE Tcoche.mostrar;
BEGIN
INHERITED mostrar;
Writeln('N° de pasajeros: ',pasajeros);
END;

PROCEDURE Tcamion.crear;
BEGIN
INHERITED crear(mt,mr,c);
tara:=t;
carga:=cr;
ejes:=e;
END;

PROCEDURE Tcamion.mostrar;
BEGIN
INHERITED mostrar;
Writeln('Tara: ',tara:7:2,' Carga: ', carga:7:2, ' N° de ejes: ', ejes);
END;

PROCEDURE Tutilitario.crear;
BEGIN
INHERITED crear(mt,mr,c,pasa);
consumoUrbano:=consu;
END;
```



## HERENCIA

```
PROCEDURE Tutilitario.mostrar;
BEGIN
 INHERITED mostrar;
 Writeln('Consumo urbano: ',consumoUrbano:5:2);
END;

PROCEDURE T4por4.crear;
BEGIN
 INHERITED crear(mt,mr,c,pasa);
 alturaVadeo:=av;
END;

PROCEDURE T4por4.mostrar;
BEGIN
 INHERITED mostrar;
 Writeln('Altura de vadeo: ',alturaVadeo:4:2);
END;

VAR
 b:Tcoche;
 c:Tcamion;
 d:Tutilitario;
 e:T4por4;

BEGIN
 b.crear('O-6297-AG','Renault 11 TSE',110,5);
 b.mostrar;
 c.crear('O-6333-AZ','Pegaso Tronner',500,20000,3000,3);
 c.mostrar;
 d.crear('O-8929-BJ','Seat IBIZA', 85, 4, 5.2);
 d.mostrar;
 e.crear('O-3333-BC','Lada Niva',140,4,0.6);
 e.mostrar;
END.
```

La ejecución del programa anterior es la siguiente:

```
Matrícula: O-6297-AG Marca: Renault 11 TSE
CV: 110
Nº de pasajeros: 5
Matrícula: O-6333-AZ Marca: Pegaso Tronner
CV: 500
Tara: 20000.00 Carga: 3000.00 Nº de ejes: 3
Matrícula: O-8929-BJ Marca: Seat IBIZA
CV: 85
Nº de pasajeros: 4
Consumo urbano: 5.20
Matrícula: O-3333-BC Marca: Lada Niva
CV: 140
Nº de pasajeros: 4
Altura de vadeo: 0.60
```

## COMPATIBILIDAD DE TIPOS OBJETO

Las variables de tipo objeto siguen unas reglas de compatibilidad de tipos ligeramente diferentes a las variables normales de Turbo Pascal. La diferencia básica está en que *el tipo de un ascendiente es compatible con el tipo de un descendiente, pero no a la inversa*. Esta extensión de la compatibilidad de tipos toma tres formas:

- Entre instancias de objetos
- Entre punteros a instancias de objetos
- Entre parámetros actuales y parámetros formales

En las tres formas anteriores es muy importante recordar que la compatibilidad se extiende *sólo* desde el descendiente al ascendiente. Los tipos descendientes contienen al menos lo mismo que sus ascendientes (en virtud de la herencia). Asignar un objeto ascendiente a un objeto descendiente podría dejar algunos campos del descendiente indefinido después de la asignación, lo que sería peligroso, y por tanto el compilador Turbo Pascal lo considera incorrecto.

La compatibilidad de tipos también opera entre punteros a tipos objeto, bajo las mismas reglas que las instancias de tipo objeto. Los punteros a tipos oobjeto descendientes pueden ser asignados a punteros a tipos objeto ascendientes.

La compatibilidad de tipos entre parámetros actuales y parámetros formales implica que puede tomarse como parámetro actual (ya sea por valor o por dirección **var**) un objeto del tipo definido como parámetro formal, o un objeto de un tipo descendiente del tipo anterior.

La flexibilidad de la compatibilidad de tipos de objetos es fundamental para realizar el *polimorfismo*, que se estudiará en el siguiente apartado.

### 13.7. POLIMORFISMO

Polimorfismo es una palabra que proviene del griego que quiere decir *muchas formas*, y polimorfismo es precisamente ésto: una forma de darle un nombre a una acción que es compartida por los distintos niveles de la jerarquía de tipos objeto, y que con cada objeto de la jerarquía implementa la acción de forma apropiada a sí mismo.

El *polimorfismo* es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos, ligados entre sí por el mecanismo de herencia. Dada una jerarquía de tipos de objetos definidos con unos métodos especiales denominados *constructores*, *virtuales*, y *destructores*; el polimorfismo permite procesar objetos cuyo tipo no se conoce en tiempo de compilación.

## POLIMORFISMO

Supongamos que se desea escribir un programa que dibuja en la pantalla diversas formas geométricas: círculos, cuadrados, rectángulos, arcos, líneas, flechas, etc... Se quiere escribir un módulo para arrastrar las distintas formas geométricas por la pantalla. La forma clásica es escribir un subprograma de arrastre para cada forma geométrica, así tendríamos: *arrastraCirculo*, *arrastraCuadrado*, *arrastraRectangulo*, *arrastraArco*, etc...

Si se definen las formas geométricas como tipos objeto dentro de una jerarquía, con el polimorfismo se puede llamar a un método *arrastra* que opera sobre cualquiera de las formas geométricas, tomando en tiempo de ejecución la decisión de cual es la forma geométrica que debe arrastrar<sup>25</sup>.

Por medio del polimorfismo se puede aplicar un método a un objeto de una jerarquía de tipos objeto, y en tiempo de ejecución se decide el objeto al que se le debe aplicar dentro de la jerarquía.

### OBJETOS POLIMORFICOS

Un objeto polimórfico es un objeto que ha sido asignado a un tipo ascendiente en virtud de las reglas de compatibilidad de tipos extendidas del compilador Turbo Pascal (véase último epígrafe del apartado anterior).

Los objetos polimórficos permiten el procesamiento de objetos cuyo tipo se desconoce en tiempo de compilación.

### METODOS VIRTUALES

Los métodos pueden ser de tres tipos: *estáticos*, *virtuales* y *dinámicos*.

Los métodos estudiados hasta aquí en este capítulo son *métodos estáticos*, es decir el compilador los asigna y resuelve todas las referencias a ellos en *tiempo de compilación*, produciéndose lo que se denomina "ligadura temprana" (en inglés *early binding*). Los métodos estáticos no permiten hacer uso de una de las principales características de la POO el *polimorfismo*, por lo que es necesario introducir unos nuevos métodos denominados *métodos virtuales*.

Con los *métodos virtuales* el compilador permite que se resuelvan sus referencias en *tiempo de ejecución*, produciéndose lo que se denomina "ligadura tardía" (en inglés *late binding*).

Los *métodos dinámicos* son un caso particular de los métodos virtuales, solamente difieren de los virtuales en la forma en que se realizan las llamadas a métodos dinámicos en tiempo de ejecución. Para el resto de los propósitos, el método dinámico puede considerarse equivalente al virtual.

---

<sup>25</sup> Ver ejercicios resueltos 13.3, 13.4 y 13.5.

Para crear un método virtual se debe poner la palabra reservada *VIRTUAL*, después de la cabecera de declaración del método dentro de la definición del tipo objeto. Véase el diagrama sintáctico de la figura 13.14, o la notación EBNF siguiente:

```

<Tipo OBJECT> ::= OBJECT (<herencia>|<vacío>)
 <lista de componentes>
 {<sección de componentes>}
 END

<herencia> ::= "(" <identificador de tipo OBJECT> ")"

<lista de componentes> ::= {<lista de campos>} {<lista de métodos>}

<sección de componentes> ::= (PUBLIC | PRIVATE) <lista de componentes>

<lista de campos> ::= {<lista de identificadores> : <tipo> ;}

<lista de métodos> ::= { <cabecera de métodos>
 (; VIRTUAL
 (<cte. entera> | <vacío>)
 | <vacío>) ; }

<cabecera de métodos> ::= <cabecera de procedimiento> |
 <cabecera de función> |
 <cabecera de constructor> |
 <cabecera de destructor>

```

Si un tipo objeto declara o hereda un método virtual, entonces las variables de ese tipo deben ser *inicializadas* a través de una llamada a unos métodos especiales denominados *constructores*, antes de cualquier llamada a un método virtual. Así pues, cualquier tipo objeto que declare o herede algún método virtual debe también declarar o heredar al menos un método constructor<sup>26</sup>.

Un tipo objeto puede redefinir (*override*) cualquiera de los métodos que hereda de sus ascendientes. La redefinición de un método estático puede cambiar la cabecera del método de la forma que desee. En contraposición, una redefinición (*override*) de un método virtual debe coincidir exactamente con el orden, tipo y nombres de los parámetros del método antepasado, y el caso de que el método sea una función también debe coincidir el tipo de resultado devuelto. La redefinición debe incluir de nuevo una directiva *virtual*.

Los *destructores* son métodos que se utilizan para la eliminación de los objetos después de su uso. Se estudiarán dentro del apartado *Objetos dinámicos* de este capítulo.

---

<sup>26</sup> Véase más adelante el epígrafe *Constructores* dentro de este mismo apartado.

POLIMORFISMO

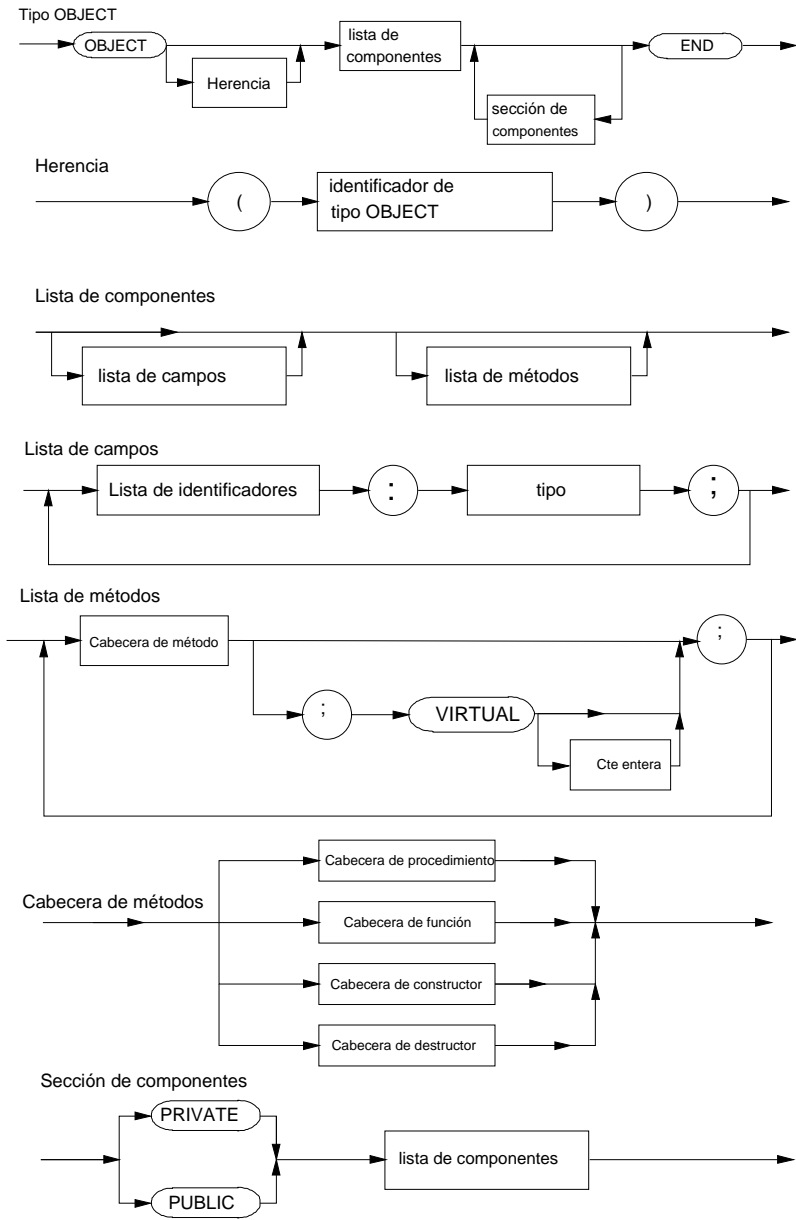


Figura 13.14 Diagrama sintáctico completo del tipo *object*

**Ejemplo 13.10**

Se declara una jerarquía de tipos objeto con métodos virtuales y constructores, cuyo esquema gráfico se representa en la figura 13.15. Es la solución adoptada para desarrollar un programa que determine la situación en un almacén y el costo de tres tipos de quesos (de bola, de barra y de rosca). Este ejemplo ya se planteó en el capítulo 2, en el apartado *Diseño orientado a objetos*, y fue desarrollado en los ejemplos 2.2, 2.3, y 2.4. A continuación se presenta el programa con la declaración de la jerarquía de tipos objeto correspondiente a la figura 13.15. Puede observarse que la función *Volumen* se define vacía en *Tsolido*, pero que se utiliza en el método *Costo* de *Tsolido*. Sin embargo cuando se utiliza el método *Costo*, no se conoce en tiempo de compilación que función *Volumen* va a utilizar, dado que hay una función *volumen* para cada tipo de sólido (esfera, cilindro y toroide).

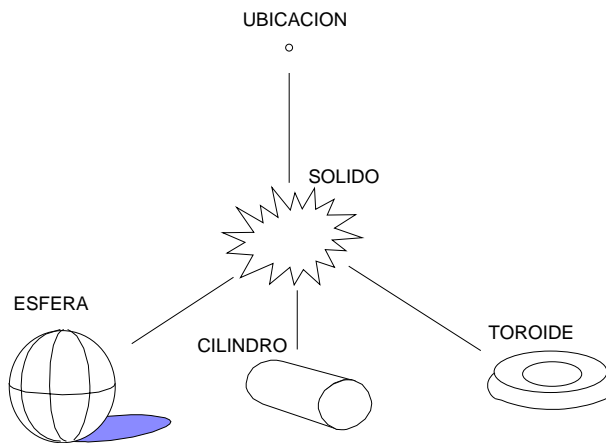


Figura 13.15 Esquema de herencia con polimorfismo

```

PROGRAM Costo_de_cuerpos (Output);
TYPE
 Tubicacion=OBJECT
 x,y,z:integer;
 PROCEDURE Iniciar (ix,iy,iz:integer);
 END;
 Tsolido=OBJECT(Tubicacion)
 CONSTRUCTOR Iniciar (ix,iy,iz:integer);
 FUNCTION Volumen:real;VIRTUAL;
 FUNCTION Costo(costeBase,costeUvolumen:real):real;VIRTUAL;
 END;
 Tesfera=OBJECT(Tsolido)
 radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

```

## POLIMORFISMO

```
Ttoroide=OBJECT(Tsolido)
 radio_central, radio_seccion:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; radio_c, radio_s:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;
Tcilindro=OBJECT(Tsolido)
 longitud,radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; l,r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

PROCEDURE Tubicacion.Iniciar;
BEGIN
x:=ix; y:=iy; z:=iz;
END;

CONSTRUCTOR Tsolido.Iniciar (ix,iy,iz:integer);
BEGIN
 INHERITED Iniciar (ix,iy,iz);
END;

FUNCTION Tsolido.Volumen;
BEGIN
END;

FUNCTION Tsolido.Costo;
BEGIN
 Costo := costeBase + Volumen * costeUvolumen;
END;

CONSTRUCTOR Tesfera.Iniciar (ix,iy,iz:integer; r:real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio:=r;
END;

FUNCTION Tesfera.Volumen;
BEGIN
 Volumen:=4/3*Pi*radio*radio*radio;
END;

CONSTRUCTOR Ttoroide.Iniciar;
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio_central:=radio_c;
 radio_seccion:=radio_s;
END;

FUNCTION Ttoroide.Volumen;
BEGIN
 Volumen:=2*pi*radio_central*Pi*radio_seccion*radio_seccion;
END;

CONSTRUCTOR Tcilindro.Iniciar (ix,iy,iz:integer; l,r :real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 longitud:=l;
 radio:=r;
END;

FUNCTION Tcilindro.Volumen;
BEGIN
 Volumen:=2*Pi*radio*longitud;
END;

(***** PROGRAMA PRINCIPAL *****)
```

## PROGRAMACION ORIENTADA A OBJETOS

```
VAR queso_bola:Tesfera;
 queso_rosca:Ttoroide;
 queso_barra:Tcilindro;
BEGIN
 queso_bola.Iniciar(10,10,10,25);
 queso_rosca.Iniciar(20,20,20,100,20);
 queso_barra.Iniciar(30,30,30,100,20);
 Writeln(' bola -> ',queso_bola.Costo(100,0.0025):7:2,' pts');
 Writeln(' rosca -> ',queso_rosca.Costo(200,0.0025):7:2,' pts');
 Writeln(' barra -> ',queso_barra.Costo(100,0.0025):7:2,' pts');
END.
```

La ejecución del programa anterior produce la siguiente salida:

```
bola -> 263.62 pts
rosca -> 2173.92 pts
barra -> 131.42 pts
```

## CONSTRUCTORES

Un *constructor* es un método utilizado para inicializar objetos que contienen métodos virtuales. Además los constructores establecen un enlace entre la instancia que llama al constructor y una estructura de datos interna denominada *Tabla de Métodos Virtuales (TMV)*<sup>27</sup> que se crea para cada tipo objeto. Normalmente la inicialización usa valores pasados como parámetros al constructor. Como el resto de los métodos los constructores pueden heredarse, y un tipo objeto puede tener varios constructores.

Los constructores no pueden ser virtuales porque el mecanismo de servicio de los métodos virtuales depende de que antes un constructor haya inicializado el objeto.

Para crear un método constructor, simplemente se sustituyen las palabras reservadas *procedure* o *function* por **CONSTRUCTOR**, tanto en la declaración del tipo objeto como en la definición del método, así en el ejemplo 13.10 mostrado anteriormente se crea el constructor *Iniciar* en el tipo objeto *Tsolido*.

```
TYPE
...
Tsolido=OBJECT(Tubicacion)
 CONSTRUCTOR Iniciar (ix,iy,iz:integer);
 FUNCTION Volumen:real;VIRTUAL 10;
 FUNCTION Costo(costeBase,costeUvolumen:real):real;VIRTUAL 20;
 END;
...
CONSTRUCTOR Tsolido.Iniciar (ix,iy,iz:integer);
BEGIN
 INHERITED Iniciar (ix,iy,iz);
END;
```

---

<sup>27</sup> Se explicará posteriormente en el apartado: *Representación interna de los tipos objeto*.



## POLIMORFISMO

Como ejemplo de herencia de constructores pueden verse los constructores *Iniciar* de *Tesfera*, *Toroide*, y *Tcilindro* del ejemplo 13.10. Puede observarse en el programa principal del ejemplo 13.10, como se debe usar en primer lugar el constructor del objeto, antes de llamar a ningún método de dicho objeto.

### REGLAS PARA EL MANEJO DE METODOS VIRTUALES

- 1ª *Es necesario llamar al método constructor del objeto, antes que a ningún método virtual.* Por lo tanto todo tipo objeto con métodos virtuales debe tener al menos un constructor.
- 2ª Una vez que un método se ha declarado virtual en un tipo objeto, si se redefine dicho método en un tipo objeto descendiente del anterior también debe ser declarado virtual. Es decir *una vez virtual, siempre virtual*. En otras palabras, un método estático nunca puede redefinir a un método virtual. Si se intenta hacer, se obtiene un error de compilación.
- 3ª *Una vez que se ha declarado un método como virtual, su cabecera no puede ser modificada por ninguno de sus descendientes.* Esto significa que no se pueden añadir, cambiar ni eliminar parámetros, ni tampoco cambiar de procedimiento a función y viceversa. Esta regla se puede comprender pensando que la definición de un método virtual es una puerta a todos sus descendientes, por esta razón, las cabeceras de todas las implementaciones de un mismo método virtual, deben ser idénticas, tanto en número como en tipo de parámetros. Esto no sucede en los métodos estáticos. Un método estático puede redefinir a otro que tenga distinto número y tipo de parámetros.
- 4ª *Cada instancia individual de un objeto debe inicializarse con una llamada distinta al constructor*<sup>28</sup>. No es suficiente inicializar un objeto y después asignar esta instancia a otras instancias adicionales. Las instancias adicionales, aunque contengan datos correctos, no son inicializadas por las sentencias de asignación, produciéndose un error en ejecución si se llama a sus métodos virtuales. Por ejemplo usando las declaraciones del ejemplo 13.13:

```
...
VAR a,b:Tesfera;
BEGIN
 a.Iniciar(10,10,10,25); (* Llamada al constructor *)
 b:=a; (* Error: b no está inicializada por un constructor *)
...
```

---

**28** Si está activada la opción *\$R* de comprobación de rangos, es decir *{\$R+}*, todas las llamadas a métodos virtuales comprueban el estado de inicialización de la instancia que usa el método virtual. Si la instancia que utiliza el método virtual no ha sido inicializada por un constructor, se produce un error de comprobación de rango en tiempo de ejecución.

## METODOS DINAMICOS

Los *métodos dinámicos* son un caso particular de los métodos virtuales, solamente difieren de los virtuales en la forma en que se realizan las llamadas a métodos dinámicos en tiempo de ejecución, dado que construyen la denominada *Tabla de Métodos Dinámicos (TMD)*, en vez de la Tabla de Métodos Virtuales (TMV).

Los métodos dinámicos son útiles cuando los tipos ascendientes definen un gran número de métodos virtuales, y el proceso de creación de tipos descendientes puede usar mucha memoria, especialmente si se crean muchos tipos objeto descendientes, dado que la TMV de un tipo objeto contiene una entrada (con un puntero a método) para cada método virtual declarado en el tipo objeto y cualquiera de sus ascendientes. Incluso aunque los tipos derivados redefinan sólo unos pocos de los métodos heredados, la TMV de cada tipo descendiente contiene punteros a método para todos los métodos virtuales heredados, aunque no hayan sido cambiados. Los métodos dinámicos son la solución para este tipo de situaciones.

Las tablas de métodos dinámicos (TMD) en lugar de representar un puntero para todos los métodos de enlace tardío (*late bound*) en un tipo objeto, tan sólo representan los métodos redefinidos en el tipo objeto.

Cuando los tipos descendientes redefinan sólo unos pocos de un número grande de los métodos de enlace tardío (*late bound*) heredados, el formato de las TMD usa menos espacio que el formato de las TMV<sup>29</sup>. Para el resto de los propósitos, el método dinámico puede considerarse equivalente al virtual.

La declaración de un método dinámico es como la de un método virtual, excepto en que la declaración de un método dinámico debe incluir un *índice de método dinámico* justo después de la palabra reservada *virtual*. El índice de método dinámico debe ser una constante entera en el rango 1..65535 y debe ser única entre los índices del método dinámico de cualquier otro método dinámico contenido en el tipo *object* y sus ascendientes. Por ejemplo:

```
PROCEDURE mensajero(VAR unMensaje:Tmensaje); VIRTUAL 333;
```

Una redefinición (*overriden*) de un método dinámico debe hacer coincidir exactamente con el orden, tipos y nombres de los parámetros del método antepasado y el tipo del resultado en el caso de que el método sea una función. La redefinición debe incluir también una directiva *virtual* seguida del mismo índice de método dinámico que fue especificado en el tipo *object* previo.

### Ejemplo 13.11

Se repite el ejemplo 13.10, pero en este caso los métodos virtuales son dinámicos.

```
PROGRAM CostoDeCuerpos_Dinamico (Output);
```

---

<sup>29</sup> El formato de la Tabla de Métodos Virtuales (TMV) y de la Tabla de Métodos Dinámicos (TMD) se define en el apartado de este capítulo: *Representación interna de los tipos objeto*.

## POLIMORFISMO

```
TYPE
 Tubicacion=OBJECT
 x,y,z:integer;
 PROCEDURE Iniciar (ix,iy,iz:integer);
 END;

 Tsolido=OBJECT(Tubicacion)
 CONSTRUCTOR Iniciar (ix,iy,iz:integer);
 FUNCTION Volumen:real;VIRTUAL 10;
 FUNCTION Costo(costeBase,costeUvolumen:real):real;VIRTUAL 20;
 END;

 Tesfera=OBJECT(Tsolido)
 radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; r:real);
 FUNCTION Volumen:real;VIRTUAL 10;
 END;

 Ttoroide=OBJECT(Tsolido)
 radio_central, radio_seccion:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; radio_c, radio_s:real);
 FUNCTION Volumen:real;VIRTUAL 10;
 END;

 Tcilindro=OBJECT(Tsolido)
 longitud,radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; l,r:real);
 FUNCTION Volumen:real;VIRTUAL 10;
 END;

PROCEDURE Tubicacion.Iniciar;
BEGIN
x:=ix; y:=iy; z:=iz;
END;

CONSTRUCTOR Tsolido.Iniciar (ix,iy,iz:integer);
BEGIN
 INHERITED Iniciar (ix,iy,iz);
END;

FUNCTION Tsolido.Volumen;
BEGIN
END;

FUNCTION Tsolido.Costo;
BEGIN
 Costo := costeBase + Volumen * costeUvolumen;
END;

CONSTRUCTOR Tesfera.Iniciar (ix,iy,iz:integer; r:real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio:=r;
END;

FUNCTION Tesfera.Volumen;
BEGIN
 Volumen:=4/3*Pi*radio*radio*radio;
END;

CONSTRUCTOR Ttoroide.Iniciar;
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio_central:=radio_c;
 radio_seccion:=radio_s;
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```
FUNCTION Ttoroide.Volumen;
BEGIN
 Volumen:=2*pi*radio_central*Pi*radio_seccion*radio_seccion;
END;

CONSTRUCTOR Tcilindro.Iniciar (ix,iy,iz:integer; l,r :real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 longitud:=l;
 radio:=r;
END;

FUNCTION Tcilindro.Volumen;
BEGIN
 Volumen:=2*Pi*radio*longitud;
END;

(***** PROGRAMA PRINCIPAL *****)

VAR queso_bola:Tesfera;
 queso_rosca:Ttoroide;
 queso_barra:Tcilindro;

BEGIN
 queso_bola.Iniciar(10,10,10,25);
 queso_rosca.Iniciar(20,20,20,100,20);
 queso_barra.Iniciar(30,30,30,100,20);
 Writeln(' bola -> ',queso_bola.Costo(100,0.0025):7:2,' pts');
 Writeln(' rosca -> ',queso_rosca.Costo(200,0.0025):7:2,' pts');
 Writeln(' barra -> ',queso_barra.Costo(100,0.0025):7:2,' pts');
END.
```

## METODOS ESTATICOS VERSUS METODOS VIRTUALES

Aunque el uso exclusivo de métodos estáticos mutila gravemente una de las principales características de la POO: el polimorfismo, en algunos casos se utilizan métodos estáticos para optimizar velocidad y uso de memoria. En este apartado se van a comparar los métodos estáticos y los métodos virtuales.

Si un tipo objeto tiene un método virtual, cuando el compilador detecta la definición del tipo objeto se crea una Tabla de Métodos Virtuales (TMV) en el segmento de datos. Cuando se inicializan las instancias de cada objeto por medio de los constructores se crea un enlace a la TMV. Cada llamada a un método virtual debe consultar la TMV.

Sin embargo si los tipos objeto no tienen métodos virtuales, no se crea la TMV en el segmento de datos, y las llamadas a los métodos estáticos se realizan directamente. Aunque las operaciones de consulta de la TMV son muy eficientes, todavía es más rápido llamar a un método estático, que a uno virtual.

La velocidad y el ahorro de memoria de los métodos estáticos se paga con la mutilación de las posibilidades de *extensibilidad* que ofrecen los métodos virtuales.

## OBJETOS DINAMICOS

### 13.8. OBJETOS DINAMICOS

Los objetos o instancias de los tipos objeto pueden ser manipulados con punteros, al igual que otras variables dinámicas del lenguaje Pascal. Es decir los objetos también pueden ser variables dinámicas que se almacenan en la memoria *heap*. Por ejemplo si se tienen las declaraciones siguientes:

```
TYPE
 PanguloSexagesimal = ^TanguloSexagesimal;
 TanguloSexagesimal = OBJECT
 PRIVATE
 angulo:real;
 PUBLIC
 PROCEDURE crear (a:real);
 FUNCTION seno:real;
 FUNCTION coseno:real;
 FUNCTION tangente:real;
 PROCEDURE mostrar;
 END;

VAR
 p:PanguloSexagesimal;
```

Es posible manejar los objetos como cualquier otra variable dinámica del lenguaje Pascal estándar.

```
New(p);
p^.crear(45);
Writeln(p^.seno:5:3);
Writeln(p^.coseno:5:3);
Writeln(p^.tangente:5:3);
p^.mostrar;
```

El programa completo se muestra en el ejemplo 13.12.

#### Ejemplo 13.12

Se muestra el código completo de un programa que maneja objetos como variables dinámicas.

```
PROGRAM SexagesimalDinamico (output);

CONST
 gradosAradianes=Pi/180;
TYPE
 PanguloSexagesimal = ^TanguloSexagesimal;
 TanguloSexagesimal = OBJECT
 PRIVATE
 angulo:real;
 PUBLIC
 PROCEDURE crear (a:real);
 FUNCTION seno:real;
 FUNCTION coseno:real;
 FUNCTION tangente:real;
 PROCEDURE mostrar;
 END;

PROCEDURE TanguloSexagesimal.crear;
BEGIN
 angulo:=a*gradosAradianes;
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```
FUNCTION TanguloSexagesimal.seno;
BEGIN
 seno:=Sin(angulo);
END;

FUNCTION TanguloSexagesimal.coseno;
BEGIN
 coseno:=Cos(angulo);
END;

FUNCTION TanguloSexagesimal.tangente;
BEGIN
 tangente:=Sin(angulo)/Cos(angulo)
END;

PROCEDURE TanguloSexagesimal.mostrar;
BEGIN
 Writeln(angulo*180/Pi:5:3);
END;

VAR
 p:PanguloSexagesimal;

BEGIN
 New(p);
 p^.crear(45);
 Writeln(p^.seno:5:3);
 Writeln(p^.coseno:5:3);
 Writeln(p^.tangente:5:3);
 p^.mostrar;
 Dispose(p);
END.
```

### AMPLIACION DEL PROCEDIMIENTO *NEW*

Si los objetos dinámicos tienen métodos virtuales, deben ser inicializados con una llamada al *constructor* antes de hacer ninguna llamada a sus métodos. El compilador Turbo Pascal extiende la sintaxis del procedimiento estándar *New* para permitir simultáneamente reservar espacio en la memoria *heap* e inicializar con un *constructor* el objeto.

El nuevo procedimiento *New* se puede llamar ahora con dos parámetros: una variable de tipo puntero y la llamada al constructor como segundo parámetro. Es decir la llamada al procedimiento *New* ampliado es la siguiente:

```
New(variable_puntero, llamada_constructor);
```

Así si se tienen las declaraciones siguientes:

```
TYPE
 Tubicacion=OBJECT
 x,y,z:integer;
 PROCEDURE Iniciar (ix,iy,iz:integer);
 END;

 Tsolido=OBJECT(Tubicacion)
 CONSTRUCTOR Iniciar (ix,iy,iz:integer);
 FUNCTION Volumen:real;VIRTUAL;
 FUNCTION Costo(costeBase,costeUvolumen:real):real;VIRTUAL;
 END;
```

## OBJETOS DINAMICOS

```
Pesfera=^Tesfera;
Tesfera=OBJECT(Tsolido)
 radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;
...
VAR pQueso_bola:Pesfera;
...
```

Se puede utilizar la nueva extensión del procedimiento *New* de la forma siguiente. El programa completo se muestra en el ejemplo 13.13.

```
New(pQueso_bola,Iniciar(10,10,10,25));
...
Writeln(' bola -> ',pQueso_bola^.Costo(100,0.0025):7:2 , ' pts');
...
```

### Ejemplo 13.13

Se escribe una nueva versión del ejemplo 13.10 utilizando objetos dinámicos. Uno de sus objetivos es ilustrar la nueva sintaxis extendida del procedimiento estándar *New*, con dos parámetros: el puntero al objeto y la llamada al constructor. En este ejemplo los constructores están redefinidos dentro de la jerarquía de tipos objeto, el compilador identifica el método *Iniciar* que debe utilizar a través del tipo del puntero que se le paso como primer parámetro al procedimiento *New*.

```
PROGRAM CostoDeCuerposConPunteros (Output);
TYPE
 Tubicacion=OBJECT
 x,y,z:integer;
 PROCEDURE Iniciar (ix,iy,iz:integer);
 END;

 Tsolido=OBJECT(Tubicacion)
 CONSTRUCTOR Iniciar (ix,iy,iz:integer);
 FUNCTION Volumen:real;VIRTUAL;
 FUNCTION Costo(costeBase,costeUvolumen:real):real;VIRTUAL;
 END;

 Pesfera=^Tesfera;
 Tesfera=OBJECT(Tsolido)
 radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

 Ptoroide=^Ttoroide;
 Ttoroide=OBJECT(Tsolido)
 radio_central, radio_seccion:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; radio_c, radio_s:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```

Pcilindro:=^Tcilindro;
Tcilindro=OBJECT(Tsolido)
 longitud,radio:real;
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; l,r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

PROCEDURE Tubicacion.Iniciar;
BEGIN
x:=ix; y:=iy; z:=iz;
END;

CONSTRUCTOR Tsolido.Iniciar (ix,iy,iz:integer);
BEGIN
 INHERITED Iniciar (ix,iy,iz);
END;

FUNCTION Tsolido.Volumen;
BEGIN
END;

FUNCTION Tsolido.Costo;
BEGIN
 Costo := costeBase + Volumen * costeUvolumen;
END;

CONSTRUCTOR Tesfera.Iniciar (ix,iy,iz:integer; r:real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio:=r;
END;

FUNCTION Tesfera.Volumen;
BEGIN
 Volumen:=4/3*Pi*radio*radio*radio;
END;

CONSTRUCTOR Ttoroide.Iniciar;
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio_central:=radio_c;
 radio_seccion:=radio_s;
END;

FUNCTION Ttoroide.Volumen;
BEGIN
 Volumen:=2*pi*radio_central*Pi*radio_seccion*radio_seccion;
END;

CONSTRUCTOR Tcilindro.Iniciar (ix,iy,iz:integer; l,r :real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 longitud:=l;
 radio:=r;
END;

FUNCTION Tcilindro.Volumen;
BEGIN
 Volumen:=2*Pi*radio*longitud;
END;

(***** PROGRAMA PRINCIPAL *****)

VAR pQueso_bola:Pesfera;
 pQueso_rosca:Ptoroide;
 pQueso_barra:Pcilindro;

```



## OBJETOS DINAMICOS

```
BEGIN
New(pQueso_bola,Iniciar(10,10,10,25));
New(pQueso_rosca,Iniciar(20,20,20,100,20));
New(pQueso_barra,Iniciar(30,30,30,100,20));
Writeln(' bola -> ',pQueso_bola^.Costo(100,0.0025):7:2 ,' pts');
Writeln(' rosca -> ',pQueso_rosca^.Costo(200,0.0025):7:2,' pts');
Writeln(' barra -> ',pQueso_barra^.Costo(100,0.0025):7:2,' pts');
Dispose(pQueso_bola);
Dispose(pQueso_rosca);
Dispose(pQueso_barra);
END.
```

## FUNCION NEW

El procedimiento *New* también se ha extendido para permitirle actuar como una función también denominada *New*. La función *New* se puede utilizar con uno o dos parámetros al igual que el procedimiento *New*.

### • **p := New(tipoPuntero)**

El parámetro pasado a *New* es el tipo del puntero que señala al objeto para el que se quiere reservar memoria *heap*. La función devuelve un puntero a la posición de memoria reservada para el objeto. Esta forma de uso de *New* como función también es válida para *todos* los tipos de datos, y no sólo para los tipos objeto. Si se utilizan las declaraciones del ejemplo 13.13, la función *New* se usaría de la siguiente forma:

```
...
pQueso_bola:=New(Pesfera);
pQueso_bola^.Iniciar(10,10,10,25);
...
Writeln(' bola -> ',pQueso_bola^.Costo(100,0.0025):7:2 ,' pts');
...
```

### • **p:= New(tipoPuntero, llamadaConstructor)**

La forma de la función *New* con dos parámetros es similar al procedimiento *New* con dos parámetros, pero con el primer parámetro con el tipo puntero al objeto que se quiere reservar memoria *heap*. El segundo parámetro es la llamada al *constructor*. La función devuelve un puntero a la posición de memoria reservada para el objeto. Si se utilizan las declaraciones del ejemplo 13.13, la función *New* se usaría de la siguiente forma:

```
...
pQueso_bola:=New(Pesfera, Iniciar(10,10,10,25));
...
Writeln(' bola -> ',pQueso_bola^.Costo(100,0.0025):7:2 ,' pts');
...
```

## LIBERACION DE OBJETOS DINAMICOS

Los objetos dinámicos, al igual que el resto de las variables dinámicas pueden ser liberados con el procedimiento *Dispose* cuando ya no se necesitan. Sin embargo puede ocurrir que para liberar el espacio asignado en la memoria *heap* a un objeto dinámico, sea necesario realizar ciertas operaciones previas, sobre todo si el objeto contiene punteros a estructuras dinámicas complejas. Entonces se debe agrupar todo lo necesario para eliminar el objeto dinámico en un único método. Este método encapsula todos los detalles necesarios para eliminar los objetos dinámicos y las estructuras dinámicas de datos anidadas dentro de ellos. Así con una única llamada al método se elimina el objeto dinámico completamente.

Se pueden definir varios métodos para liberar la memoria *heap* de un objeto dinámico determinado. Los objetos dinámicos complejos pueden necesitar liberarse de distintas formas dependiendo de como se asignaron o se usaron, o en función del estado del objeto dinámico en el momento que se desea su liberación.

El compilador Turbo Pascal proporciona unos tipos especiales de métodos denominados *destructores* para liberar la memoria *heap* asignada a los objetos dinámicos.

## DESTRUCTORES

Un destructor es un método especial que además de liberar los objetos dinámicos de la memoria *heap* realiza otras tareas necesarias para un tipo objeto dado.

Para crear un método destructor, simplemente se sustituyen las palabras reservadas *procedure* o *function* por **DESTRUCTOR**, tanto en la declaración del tipo objeto como en la definición del método. A continuación se muestra la declaración de un tipo objeto con un método destructor (el código completo se muestra en el ejemplo 13.15).

```
Plista=^Tlista;
Tlista=OBJECT
 PRIVATE
 cabeza:Pnodo;
 PUBLIC
 CONSTRUCTOR Iniciar;
 DESTRUCTOR Destruir;VIRTUAL;
 PROCEDURE Insertar(unElemento:Psolido);
 PROCEDURE Mostrar;
END;
```

La implementación del método destructor *Destruir* se presenta a continuación:

```
DESTRUCTOR Tlista.Destruir;
VAR
 p:Pnodo;
BEGIN
 WHILE cabeza<>NIL DO
 BEGIN
 p:=cabeza;
 cabeza:=p^.sig;
 Dispose(p^.info, eliminar);
```

## OBJETOS DINAMICOS

```
Dispose(p);
END;
END;
```

A continuación se describen algunas características de los destructores:

- Se pueden definir *múltiples destructores* para un único tipo objeto. Al igual que se pueden definir cualquier número de métodos en un tipo objeto.
- Los destructores *pueden heredarse*.
- Los destructores pueden ser *estáticos o virtuales*. En general los destructores suelen definirse *virtuales*, para que se ejecute en cada caso el destructor adecuado a cada tipo de objeto.
- Los destructores sólo operan realmente sobre los *objetos dinámicos*, aunque no hay problemas por usar destructores con *objetos estáticos*.
- Los destructores son necesarios realmente cuando se desea liberar la memoria *heap* que ocupan los *objetos polimórficos*. Recuérdese que cuando en tiempo de compilación el programa está manejando un objeto polimórfico, no conoce cual es el tipo exacto de dicho objeto, tan sólo sabe que el tipo del objeto es uno de la jerarquía de objetos descendientes del tipo especificado. Sin embargo en tiempo de ejecución el destructor conoce el tamaño exacto en bytes de cada tipo objeto, consultando la Tabla de Métodos Virtuales (TMV). La TMV de cualquier tipo objeto está disponible a través del parámetro invisible *Self*, que se pasa en la llamada a cualquier método. Un destructor es un tipo especial de método, y recibe una copia de *Self* en la *stack* cuando lo llama un objeto. Para que se realice esta liberación de memoria de enlace tardío, el destructor debe llamarse como parte de la sintaxis extendida del procedimiento *Dispose*, que se explica en el epígrafe siguiente.
- Un método destructor puede estar vacío y ser útil. La utilidad no viene dada por el cuerpo del método, sino por el código que añade el compilador al encontrar la palabra reservada *DESTRUCTOR*. Así en el ejemplo 13.14 aparece el siguiente destructor:

```
DESTRUCTOR Tsolido.Eliminar;
BEGIN
END;
```

- Habitualmente en inglés se utiliza el identificador *Done* para los destructores.

### AMPLIACION DEL PROCEDIMIENTO *DISPOSE*

El nuevo procedimiento *Dispose* se puede llamar ahora con dos parámetros: una variable de tipo puntero y la llamada al destructor como segundo parámetro.

```
Dispose(variable_puntero, llamada_destructor);
```

Así en el ejemplo 13.15, se crea una lista de objetos. Para liberar cada elemento de la lista es necesario llamar previamente al destructor del objeto:

## PROGRAMACION ORIENTADA A OBJETOS

```
Dispose(p^.info, eliminar);
```

### Ejemplo 13.14: Unit Costes

Se introducen los tipos objeto utilizados en el ejemplo 13.10 en una *unit*. Se han añadido: las declaraciones de los punteros a los tipos objeto, un destructor, y un método para visualizar los datos contenidos en los objetos. Esta *unit* se utilizará en el ejemplo 13.15 para crear listas de objetos polimórficos.

```
UNIT Costes;
INTERFACE
TYPE
 Tubicacion=OBJECT
 PRIVATE
 x,y,z:integer;
 PUBLIC
 PROCEDURE Iniciar (ix,iy,iz:integer);
 END;

 Psolido=^Tsolido;
 Tsolido=OBJECT(Tubicacion)
 CONSTRUCTOR Iniciar (ix,iy,iz:integer);
 DESTRUCTOR Eliminar;VIRTUAL;
 FUNCTION Volumen:real;VIRTUAL;
 FUNCTION Costo(costeBase,costeUvolumen:real):real;VIRTUAL;
 PROCEDURE Visualizar(costeBase, costeUvolumen:real); VIRTUAL;
 END;

 Pesfera=^Tesfera;
 Tesfera=OBJECT(Tsolido)
 PRIVATE
 radio:real;
 PUBLIC
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

 Ptoroide=^Ttoroide;
 Ttoroide=OBJECT(Tsolido)
 PRIVATE
 radio_central, radio_seccion:real;
 PUBLIC
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; radio_c, radio_s:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

 Pcilindro=^Tcilindro;
 Tcilindro=OBJECT(Tsolido)
 PRIVATE
 longitud,radio:real;
 PUBLIC
 CONSTRUCTOR Iniciar (ix,iy,iz:integer; l,r:real);
 FUNCTION Volumen:real;VIRTUAL;
 END;

IMPLEMENTATION

 PROCEDURE Tubicacion.Iniciar;
 BEGIN
 x:=ix; y:=iy; z:=iz;
 END;
```

## OBJETOS DINAMICOS

```
CONSTRUCTOR Tsolido.Iniciar (ix,iy,iz:integer);
BEGIN
 INHERITED Iniciar (ix,iy,iz);
END;

FUNCTION Tsolido.Volumen;
BEGIN
 RunError(211); (* Este es un método abstracto *)
END;

FUNCTION Tsolido.Costo;
BEGIN
 Costo := costeBase + Volumen * costeUvolumen;
END;

DESTRUCTOR Tsolido.Eliminar;
BEGIN
END;

PROCEDURE Tsolido.Visualizar;
BEGIN
 Writeln(x:3,y:3,z:3, volumen:11:2, costo(costeBase, costeUvolumen):10:2);
END;

CONSTRUCTOR Tesfera.Iniciar (ix,iy,iz:integer; r:real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio:=r;
END;

FUNCTION Tesfera.Volumen;
BEGIN
 Volumen:=4/3*Pi*radio*radio*radio;
END;

CONSTRUCTOR Ttoroide.Iniciar;
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 radio_central:=radio_c;
 radio_seccion:=radio_s;
END;

FUNCTION Ttoroide.Volumen;
BEGIN
 Volumen:=2*pi*radio_central*Pi*radio_seccion*radio_seccion;
END;

CONSTRUCTOR Tcilindro.Iniciar (ix,iy,iz:integer; l,r :real);
BEGIN
 INHERITED Iniciar(ix,iy,iz);
 longitud:=l;
 radio:=r;
END;

FUNCTION Tcilindro.Volumen;
BEGIN
 Volumen:=2*Pi*radio*longitud;
END;

END.
```

### Ejemplo 13.15: Lista de objetos

Se crea una lista de objetos usando la *unit* del ejemplo 13.14.

## PROGRAMACION ORIENTADA A OBJETOS

```
PROGRAM ListaCuerpos(Output);

USES Costes;

TYPE
 Pnodo= ^Tnodo;
 Tnodo=RECORD
 info:Psolido;
 sig:Pnodo;
 END;

 Plista=^Tlista;
 Tlista=OBJECT
 PRIVATE
 cabeza:Pnodo;
 PUBLIC
 CONSTRUCTOR Iniciar;
 DESTRUCTOR Destruir;VIRTUAL;
 PROCEDURE Insertar(unElemento:Psolido);
 PROCEDURE Mostrar(costeBase, costeUvolumen:real);
 END;

CONSTRUCTOR Tlista.Iniciar;
BEGIN
 cabeza:=NIL;
END;

DESTRUCTOR Tlista.Destruir;
VAR
 p:Pnodo;
BEGIN
 WHILE cabeza<>NIL DO
 BEGIN
 p:=cabeza;
 cabeza:=p^.sig;
 Dispose(p^.info, eliminar);
 Dispose(p);
 END;
END;

PROCEDURE Tlista.Insertar;
VAR
 p:Pnodo;
BEGIN
 New(p);
 p^.info:=unElemento;
 p^.sig:=cabeza;
 cabeza:=p;
END;

PROCEDURE Tlista.mostrar;
VAR
 p:Pnodo;
BEGIN
 p:=cabeza;
 Writeln ('Situación Volumen Coste');
 IF p=NIL THEN Writeln('Lista vacia');
 WHILE p<>NIL DO
 BEGIN
 p^.info^.Visualizar(costeBase, costeUvolumen);
 p:=p^.sig;
 END;
END;
```

## OBJETOS DINAMICOS

```
VAR
 unaListaExistencias:Tlista;

BEGIN
 WITH unaListaExistencias DO
 BEGIN
 Iniciar;
 Insertar(New(Pesfera, Iniciar(1,0,0,25)));
 Insertar(New(Pcilindro, Iniciar(2,0,0,100,20)));
 Insertar(New(Ptoroide, Iniciar(3,0,0,100,20)));
 Mostrar(100,0.0025);
 Destruir;
 END;
 END.
```

La ejecución de este programa produce la siguiente salida:

| Situación | Volumen   | Coste   |
|-----------|-----------|---------|
| 3 0 0     | 789568.35 | 2073.92 |
| 2 0 0     | 12566.37  | 131.42  |
| 1 0 0     | 65449.85  | 263.62  |

## TRATAMIENTO DE ERRORES DE MEMORIA HEAP

El compilador Turbo Pascal permite instalar una función propia de error de *heap* mediante la variable *HeapError* de la *unit System*. Esta función debe tener la siguiente cabecera:

```
FUNCTION MiFuncionErrorHeap (tamagno:word): integer; FAR;
```

La función de error de *heap* se llama siempre que una llamada a *New* o *GetMem* no puede completar la petición. El parámetro *tamagno* contiene la dirección del bloque que no se pudo asignar, y la función error del *heap* debería intentar liberar un bloque de al menos ese tamaño. Obsérvese que la directiva *far* obliga al compilador a usar el modelo lejano (FAR) para la función de error de *heap*.

Según ha tenido éxito o no la asignación de memoria *heap* la función devuelve los valores siguientes:

- 0 Indica error, generando un error en tiempo de ejecución.
- 1 Indica error, pero en lugar de generar un error en tiempo de ejecución, hace que *New* o *GetMem* devuelvan un puntero a *NIL*.
- 2 Indica éxito.

La función error estándar siempre devuelve 0 y genera un error en tiempo de ejecución siempre que no se pueda completar una llamada a *New* o *GetMem*. Sin embargo para muchas aplicaciones se puede construir la función de error propia siguiente:

```
FUNCTION MiFuncionErrorHeap (tamagno:word): integer; FAR;
BEGIN
 MiFuncionErrorHeap:=1;
END;
```

Esta función de error se instala asignando su dirección a la variable *heapError*, de la siguiente forma:

```
HeapError:=@MiFuncionErrorHeap;
```

Una vez instalada esta función hace que *New* o *GetMem* devuelvan *NIL* cuando no pueden completar la petición, en lugar de interrumpir bruscamente la ejecución del programa.

Cuando se llama a un constructor se realiza automáticamente la asignación e inicialización de los campos de la Tabla de Métodos Virtuales (TMV). Todo esto ocurre antes de que se llegue al primer *BEGIN* del constructor. Si la asignación falla y la función de error de heap devuelve 1, el constructor se salta la ejecución de la parte de sentencias y devuelve un puntero *NIL*. Pero si no fallase, y el control del programa llega al primer *BEGIN*, el constructor podría intentar asignar variables dinámicas a campos puntero de la instancia, y se puede producir un fallo. En este caso un constructor debería deshacer todas las asignaciones realizadas correctamente y liberar la instancia del tipo objeto, devolviendo un puntero *NIL* (y no dejando las variables dinámicas colgadas). Para hacer posible tal retroceso el compilador Turbo Pascal incorpora el procedimiento estándar *Fail*.

### PROCEDIMIENTO *FAIL*

El procedimiento *Fail* no tiene parámetros y tan sólo puede llamarse desde dentro de un constructor. Una llamada a *Fail* provoca en el constructor la liberación de la instancia dinámica que fue asignada a la entrada al constructor y causa la devolución de un puntero *NIL* para indicar su fallo.

Cuando se utiliza la sintaxis extendida de *New*, un valor resultante de *NIL* en la variable puntero especificada indica que la operación ha fallado. Desafortunadamente esto no es siempre posible, para esos casos el compilador Turbo Pascal permite utilizar un constructor como función booleana en una expresión: un valor de retorno *True* indica éxito, y un valor de retorno *False* indica fallo debido a la llamada a *Fail* dentro del constructor.

## 13.9. ABSTRACCION

La abstracción es uno de los caminos fundamentales que tiene el ser humano para reconocer las similitudes entre ciertos objetos, situaciones y procesos del mundo real, y concentrarlos remarcando las similitudes e ignorando las diferencias no sustanciales. En POO la abstracción se centra en la parte de como actúa un objeto, y sirve para separar lo esencial del objeto de su implementación. Es decir la abstracción permite concentrarse en lo que ocurre en un sistema determinado, e ignorar los detalles interiores del sistema.



## GENERICIDAD

Los métodos de diseño y análisis orientados a objetos usan la abstracción como un camino efectivo para el estudio del dominio del problema y examinar las acciones que ocurren sobre los tipos objeto. En este apartado se presentarán los *tipos objeto abstractos* y su papel en el diseño de la jerarquía de tipos objeto.

Cuando se diseña una jerarquía de tipos objeto, se pueden agrupar las operaciones comunes en la jerarquía utilizando un tipo objeto abstracto. El tipo objeto abstracto especificará lo que ocurre en las instancias de varios tipos objeto descendientes. Estos a su vez completarán los detalles necesarios de como llevar a cabo las operaciones concretas sobre cada tipo objeto.

Los tipos objeto abstractos pueden clasificarse en dos categorías:

- *Tipos objeto abstractos puros*: especifican los métodos públicos y privados comunes a los descendientes de la jerarquía de tipos objeto. La implementación de estos métodos no contienen sentencias. Así los tipos abstractos puros, son completamente no funcionales.
- *Tipos objeto abstractos parcialmente funcionales*: especifican los métodos públicos y privados, así como los campos de datos que son comunes a todos o a la mayor parte de los descendientes. Además implementan algunos de los métodos comunes de los tipos objetos descendientes.

Se recomienda para ambos tipos de objetos abstractos el uso de las siguientes reglas generales:

- 1ª Se deben declarar *todos los métodos y campos de datos* del tipo objeto abstracto como *privados*. Con esto se consigue prevenir el uso de instancias no permitidas a los tipos objeto abstractos por parte de programas clientes de dichos tipos objeto.
- 2ª Los métodos no implementados por el tipo objeto abstracto deben declararse *virtuales*. Este tipo de declaración asegura que los descendientes del tipo objeto abstracto también serán virtuales y tendrán la misma lista de parámetros. El beneficio de esto es el soporte del polimorfismo.

Los tipos objeto abstractos están situados habitualmente en la raíz de la jerarquía de tipos objetos, aunque también puede haber varios tipos objeto abstractos definiéndose subjerarquías. Esto último es frecuente en jerarquías complejas de tipos objetos, como por ejemplo las suministradas por *Turbo Vision* y *Object Windows*.

### 13.10. GENERICIDAD

La *genericidad*, la *extensibilidad* y la *reutilización de código* son algunos de los principales objetivos de la POO, con estas características se pueden construir módulos (*unit* en Turbo Pascal) que se pueden compilar y distribuir sin el código fuente, de tal forma que por medio de la utilización

de la herencia y el polimorfismo los usuarios pueden crear sus propios objetos usando todas las características de los módulos compilados y añadiéndoles sus adaptaciones a los problemas concretos.

Un primer paso hacia la genericidad es el uso de *estructuras de datos genéricas*. Otro paso sería el uso de marcos de trabajo como *Turbo Vision* u *Object Windows*.

Los ingredientes básicos para crear estructuras de datos genéricas son los siguientes:

- Un puntero a la dirección base de comienzo de la estructura. El puntero es de tipo genérico *Pointer*.
- El tamaño de cada elemento de la estructura de datos.
- Funciones de comparación.
- El tamaño de la estructura.
- La asignación de datos a los elementos de la estructura genérica se realiza mediante el procedimiento estándar de Turbo Pascal *Move*, cuya sintaxis general es de la forma:

```
Move(PunteroOrigen^,PunteroDestino^, TamagnoElemento);
```

### Ejemplo 13.16: unit pila

En este ejemplo se desea construir un tipo pila genérico, que pueda ser empleado como pila de enteros, de reales, o de cualquier tipo. Para lograr este objetivo en el campo *info* de *Tnodo* es de tipo *pointer*, puntero genérico que tiene Turbo Pascal y que puede apuntar a cualquier tipo de datos (véase fig. 13.16). Además la pila genérica podrá trabajar sobre la memoria principal (*heap*) o sobre memoria secundaria (*fichero en disco*). Esta *unit* tiene un tipo objeto abstracto denominado *Tpila*, y dos tipos descendientes *TpilaHeap* y *TpilaFichero*.

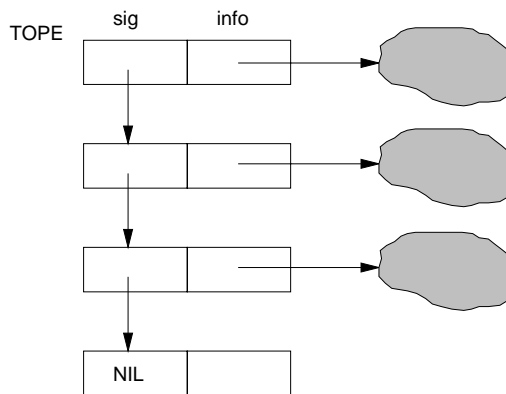


Figura 13.16: Pila genérica

## GENERICIDAD

```
UNIT Pila;

INTERFACE

TYPE
 Pnodo = ^Tnodo;
 Tnodo = RECORD
 info : pointer;
 sig : Pnodo
 END;

 Tpila = OBJECT
 CONSTRUCTOR inicializar(unTamagnoElemento : word);
 DESTRUCTOR destruir; VIRTUAL;
 FUNCTION estaVacía : boolean;
 PROCEDURE meter(x : pointer); VIRTUAL; (* push *)
 FUNCTION sacar(x : pointer) : boolean; VIRTUAL;
 PROCEDURE borrar; VIRTUAL;
 PRIVATE
 tamagnoElemento, (* Tamaño en bytes de cada elemento *)
 numElementos: word; (* N° de elementos actualmente en la pila *)
 END;

 TpilaHeap = OBJECT(Tpila)
 CONSTRUCTOR inicializar(unTamagnoElemento : word);
 DESTRUCTOR destruir; VIRTUAL;
 PROCEDURE meter(x : pointer); VIRTUAL;
 FUNCTION sacar(x : pointer) : boolean; VIRTUAL;
 PROCEDURE borrar; VIRTUAL;
 PRIVATE
 tope : Pnodo; (* puntero al tope de la pila *)
 END;

 TpilaFichero = OBJECT(Tpila)
 CONSTRUCTOR inicializar(unTamagnoElemento : word;
 nombreFichero : STRING);
 DESTRUCTOR destruir; VIRTUAL;
 PROCEDURE meter(x : pointer); VIRTUAL;
 FUNCTION sacar(x : pointer) : boolean; VIRTUAL;
 PROCEDURE borrar; VIRTUAL;
 PRIVATE
 bufferDeDatos : pointer; (* puntero al buffer *)
 nombreFicheroMV : STRING; (* nombre del fichero de
 memoria virtual *)
 ficheroMV : FILE;
 END;

IMPLEMENTATION

CONSTRUCTOR Tpila.inicializar(unTamagnoElemento : word);
BEGIN
 TamagnoElemento := unTamagnoElemento;
 numElementos := 0;
END;

DESTRUCTOR Tpila.destruir;
BEGIN
 borrar;
END;

FUNCTION Tpila.estaVacía : boolean;
BEGIN
 estaVacía := numElementos = 0
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```

PROCEDURE Tpila.meter(x : pointer);
BEGIN
END;

FUNCTION Tpila.sacar(x : pointer) : boolean;
BEGIN
END;
PROCEDURE Tpila.borrar;
BEGIN
END;

(*****)

CONSTRUCTOR TpilaHeap.inicializar(unTamagnoElemento : WORD);
BEGIN
 INHERITED inicializar(unTamagnoElemento);
 tope := NIL
END;

DESTRUCTOR TpilaHeap.destruir;
BEGIN
 borrar;
END;

PROCEDURE TpilaHeap.meter(x : pointer);
VAR p : Pnodo;
BEGIN
 IF tope <> NIL
 THEN
 BEGIN
 New(p);
 IF p = NIL THEN Exit;
 GetMem(p^.info, TamagnoElemento);
 IF p^.info = NIL THEN Exit;
 Move(x^, p^.info^, TamagnoElemento);
 p^.sig := tope;
 tope := p
 END
 ELSE
 BEGIN
 New(tope);
 IF tope = NIL THEN Exit;
 GetMem(tope^.info, TamagnoElemento);
 IF tope^.info = NIL THEN Exit;
 Move(x^, tope^.info^, TamagnoElemento);
 tope^.sig := NIL
 END;
 INC(numElementos)
 END;
END;

FUNCTION TpilaHeap.sacar(x : pointer) : boolean;
VAR p : Pnodo;
BEGIN
 IF numElementos > 0
 THEN
 BEGIN
 Move(tope^.info^, x^, TamagnoElemento);
 FreeMem(tope^.info, TamagnoElemento);
 p := tope;
 tope := tope^.sig;
 Dispose(p);
 DEC(numElementos);
 sacar := TRUE;
 END
 ELSE
 sacar := FALSE;
 END;
END;

```

## GENERICIDAD

```
PROCEDURE TpilaHeap.borrar;
VAR x : pointer;
BEGIN
 GetMem(x, TamagnoElemento);
 WHILE sacar(x) DO (* bucle vacio *);
 FreeMem(x, TamagnoElemento);
END;

(*****)

CONSTRUCTOR TpilaFichero.inicializar(unTamagnoElemento:WORD;
 nombreFichero : STRING);
BEGIN
 INHERITED inicializar(unTamagnoElemento);
 nombreFicheroMV := nombreFichero;
 Assign(ficheroMV, nombreFicheroMV);
 {$I-} Rewrite(ficheroMV, TamagnoElemento); {$I+}
 IF IOresult <> 0 THEN Exit;
 GetMem(bufferDeDatos, TamagnoElemento);
END;

DESTRUCTOR TpilaFichero.destruir;
BEGIN
 borrar;
 FreeMem(bufferDeDatos, TamagnoElemento);
END;

PROCEDURE TpilaFichero.meter(x : pointer);
BEGIN
 INC(numElementos);
 Seek(ficheroMV, numElementos-1);
 BlockWrite(ficheroMV, x^, 1);
END;

FUNCTION TpilaFichero.sacar(x : pointer) : boolean;
BEGIN
 IF numElementos > 0
 THEN
 BEGIN
 DEC(numElementos);
 Seek(ficheroMV, numElementos);
 BlockRead(ficheroMV, x^, 1);
 sacar := TRUE;
 END
 ELSE
 sacar := FALSE;
END;

PROCEDURE TpilaFichero.borrar;
BEGIN
 numElementos := 0;
 {$I-}
 Close(ficheroMV);
 Erase(ficheroMV);
 {$I+}
END;

END.
```

**Ejemplo 13.17: uso de la unit pila**

A continuación se muestra un programa que utiliza la *unit pila* para la construcción de pilas genéricas.

```

PROGRAM PruebaPilas (Output);
USES pila;

TYPE
 Pinteger=^integer;
 Preal=^real;
 Pchar=^char;
 Palumno=^Talumno;
 Talumno=RECORD
 nombre:STRING;
 DNI:longint;
 nota:real;
 END;

VAR
 unaPila, otraPila:TpilaHeap;
 unaPilaVirtual:TpilaFichero;
 p:Pinteger;
 q:Preal;
 r:Pchar;
 s:Palumno;
 nombreFich:STRING;
 nElementos,i:integer;

BEGIN
 unaPila.inicializar(SizeOf(integer));

 p^:=10;
 unaPila.meter(p);

 p^:=20;
 unaPila.meter(p);

 p^:=30;
 unaPila.meter(p);

 unaPila.sacar(p);
 Writeln('Sacando: ', p^); (* 30 *)

 unaPila.sacar(p);
 Writeln('Sacando: ', p^); (* 20 *)

 unaPila.sacar(p);
 Writeln('Sacando: ', p^); (* 10 *)

 unaPila.destruir;

 unaPila.inicializar(SizeOf(char));

 r^:='A';
 unaPila.meter(r);

 r^:='B';
 unaPila.meter(r);

 r^:='C';
 unaPila.meter(r);

```

## GENERICIDAD

```
REPEAT
 unaPila.sacar(r);
 Writeln('Sacando: ', r^);
UNTIL unaPila.estaVacía;

unaPila.destruir;

unaPila.inicializar(SizeOf(real));

q^:=111.11;
unaPila.meter(q);

q^:=222.22;
unaPila.meter(q);

q^:=333.33;
unaPila.meter(q);

REPEAT
 unaPila.sacar(q);
 Writeln('Sacando: ', q^:6:2);
UNTIL unaPila.estaVacía;

unaPila.destruir;

otraPila.inicializar(SizeOf(Talumno));

s^.nombre:='Guillermo Cueva';
s^.DNI:=10599955;
s^.nota:=9.5;

otraPila.meter(s);

s^.nombre:='Antonio Cueva';
s^.DNI:=10577722;
s^.nota:=8.5;
otraPila.meter(s);

s^.nombre:='Paloma Cueva';
s^.DNI:=10533344;
s^.nota:=7.5;
otraPila.meter(s);

REPEAT
 otraPila.sacar(s);
 Writeln(s^.nombre:30, s^.DNI:10, s^.nota:6:2);
UNTIL otraPila.estaVacía;

otraPila.destruir;

Write('Deme el nombre del fichero temporal: ');
Readln(nombreFich);
Write('Deme el número de elementos a meter en la pila: ');
Readln(nElementos);

unaPilaVirtual.inicializar(Sizeof(real),nombreFich);

Randomize; (* inicializa el generador de números aleatorios *)
FOR i:=1 TO nElementos DO
 BEGIN
 q^:=Random(1000)/100; (* valores aleatorios de tipo real entre 0 y 10 *)
 unaPilaVirtual.meter(q);
 END;
```

```

REPEAT
 unaPilaVirtual.sacar(q);
 Writeln('Sacando: ', q^:6:2);
UNTIL unaPilaVirtual.estaVacía;

unaPilaVirtual.destruir;

END.

```

### 13.11. REPRESENTACION INTERNA DE LOS TIPOS OBJETO

El formato interno de los datos de una variable de tipo objeto es similar al de una variable de tipo registro. Los campos de una variable de tipo objeto se almacenan en el orden de la declaración, como una secuencia contigua de variables. Cualquier campo heredado de un tipo ascendiente se almacena antes que un campo nuevo definido en el tipo descendiente.

Además si el tipo objeto de la instancia contiene métodos virtuales, constructores o destructores, el compilador reserva un campo extra para almacenar la dirección (desplazamiento) de la *Tabla de Métodos Virtuales (TMV)* del tipo objeto dentro del segmento de datos. El campo con la dirección a la TMV aparece inmediatamente después de los campos ordinarios del tipo objeto. Cuando un tipo objeto hereda métodos virtuales, constructores o destructores también hereda el campo con la dirección a la TMV, por lo tanto no se reserva otro campo adicional.

La inicialización del campo con la dirección (desplazamiento) de la TMV de una instancia lo realiza el constructor(es) del tipo objeto. Un programa nunca inicializa o accede explícitamente al campo que contiene la dirección de la TMV.

#### TABLA DE METODOS VIRTUALES

La *tabla de métodos virtuales TMV* (en inglés *virtual method table*, VMT) es la estructura de datos interna que soporta los métodos virtuales. Cada tipo objeto que contiene o hereda métodos virtuales, constructores o destructores tiene asociado una TMV, que se almacena en la parte inicializada del segmento de datos del programa. Sólo hay una TMV por cada tipo objeto (no una por cada instancia), sin embargo dos tipos objeto distintos nunca comparten una TMV, independientemente de lo idénticos que parezcan ser. Las TMV las construye automáticamente el compilador, y nunca son manipuladas directamente por un programa.

La estructura de la TMV es la siguiente:

- La primera palabra (*word*) contiene el tamaño de las instancias del tipo objeto asociado.
- La segunda palabra (*word*) contiene el tamaño negativo de las instancias del tipo objeto asociado. Se usa como mecanismo de validación con la directiva *\$R* (comprobación de rango).



## EJERCICIOS RESUELTOS

- La tercera palabra (*word*) contiene la dirección (desplazamiento) en el segmento de datos de la Tabla de Métodos Dinámicos del tipo objeto, o cero si no contiene métodos dinámicos.
- La cuarta palabra está reservada y siempre contiene cero.
- Por último hay una lista de punteros a los métodos virtuales del tipo objeto en el orden de la declaración.

## TABLA DE METODOS DINAMICOS

Un tipo objeto sólo construirá una *Tabla de Métodos Dinámicos TMD* si tiene o redefine métodos dinámicos. La estructura interna de la TMD es la siguiente:

- La primera palabra (*word*) contiene la dirección (desplazamiento) de la TMD padre, o cero si no la hay.
- Las palabras segunda y tercera de una TMD se utilizan como almacenamiento para acelerar la búsqueda de métodos dinámicos.
- La cuarta palabra contiene el contador de entradas de la TMD.
- Por último hay una lista de palabras, cada una de las cuales contiene un índice de método dinámico. Y a continuación una lista de punteros a los métodos dinámicos correspondientes con los índices anteriores, y en el mismo orden.

## 13.12. EJERCICIOS RESUELTOS

- 13.1** Construir un programa que sea una versión propia y simplificada de la orden *copy* del sistema operativo DOS, utilizando métodos estáticos.

### Solución

```
PROGRAM Copia_ficheros;

(* Forma de uso:
 COPIA f_fuente f_destino *)

TYPE
 accion =(lectura,escritura);
 bloque_datos= ARRAY [1..512] OF byte;

 fichero = OBJECT
 fp: FILE;
 PROCEDURE Abre_fichero (nombre:string; accion_f:accion);
 PROCEDURE Lee_bloque (VAR fb:bloque_datos;VAR tamagno:integer);
 PROCEDURE Escribe_bloque (fb:bloque_datos; tamagno:integer);
 PROCEDURE Cierra_fichero;
 END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```
PROCEDURE fichero.Abre_fichero;
BEGIN
 Assign(fp,nombre);
 CASE accion_f OF
 lectura: BEGIN
 {$I-}
 Reset(fp,1);
 {$I+}
 IF IoResult<>0 THEN
 BEGIN
 Writeln(nombre, ' ; no lo encontré ! ');
 Writeln('Forma de uso:');
 Writeln('COPIA fich_fuente fich_destino');
 Halt(1);
 END;
 Writeln(nombre, ' abierto para lectura ...');
 END;
 escritura: BEGIN
 Rewrite(fp,1);
 Writeln(nombre, ' abierto para escritura ...');
 END;
 END; { fin del CASE }
END;
```

```
PROCEDURE fichero.Cierra_fichero;
BEGIN
 Close(fp);
 Writeln('Fichero cerrado');
END;
```

```
PROCEDURE fichero.Lee_bloque;
BEGIN
 BlockRead(fp,fb,Sizeof(fb),tamagno);
 Writeln('Leyendo ', tamagno, ' bytes ...');
END;
```

```
PROCEDURE fichero.Escribe_bloque;
BEGIN
 BlockWrite(fp,fb,tamagno);
 Writeln('Escribiendo ', tamagno, ' bytes ...');
END;
```

```
(***** Programa Principal *****)
```

```
VAR
 f_entrada, f_salida :fichero;
 datos: bloque_datos;
 tamagno:integer;
```

```
BEGIN
 IF ParamCount<>2 THEN
 BEGIN
 Writeln('Forma de uso: COPIA fich_fuente fich_destino');
 Halt(1);
 END;
 f_entrada.Abre_fichero(ParamStr(1),lectura);
 f_salida.Abre_fichero(ParamStr(2),escritura);
 REPEAT
 f_entrada.Lee_bloque(datos,tamagno);
 f_salida.Escribe_bloque(datos,tamagno);
 UNTIL tamagno<>Sizeof(bloque_datos);
```

## EJERCICIOS RESUELTOS

```
f_entrada.Cierra_fichero;
f_salida.Cierra_fichero;
END.
```

La ejecución de este programa en línea de comandos:

```
c:> copia copia.exe copia.bin
copia.exe abierto para lectura ...
copia.bin abierto para escritura ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 512 bytes ...
Escribiendo 512 bytes ...
Leyendo 160 bytes ...
Escribiendo 160 bytes ...
Fichero cerrado
Fichero cerrado
```

- 13.2** Escribir un programa que maneje un tipo objeto ventana simple, con métodos estáticos. El programa moverá aleatoriamente las ventanas por la pantalla. Se utiliza dicho programa para ilustrar el uso de arrays de tipos objeto.

### Solución

```
PROGRAM Manejo_de_ventanas (output);

USES Crt,Dos;

TYPE
 str10=STRING[10];
 ventana=OBJECT
 xpos,ypos,
 xlong,ylong:integer;
 titulo:str10;
 PROCEDURE Crea(x1,y1,x2,y2:integer;cadena:str10);
 PROCEDURE Dibuja;
 PROCEDURE Mueve(x,y:integer);
 PROCEDURE Borra;
 FUNCTION CogeX:integer;
 FUNCTION CogeY:integer;
 END;

PROCEDURE ventana.Crea;
 BEGIN
 xpos:=x1;
 ypos:=y1;
 xlong:=x2;
 ylong:=y2;
 IF xlong<4 THEN xlong:=4;
 IF ylong<3 THEN ylong:=3;
```

## PROGRAMACION ORIENTADA A OBJETOS

```

titulo:=cadena;
Dibuja;
END;

PROCEDURE ventana.Dibuja;
CONST
 caja: ARRAY[1..6] OF STRING[1]=(Chr(201),Chr(205),Chr(187),
Chr(186),Chr(200),Chr(188));
VAR
 esquinal,esquina2:word;
 i:integer;
BEGIN
 esquinal:=WindMin; (* Función incluida en Crt *)
 esquina2:=WindMax; (* Función incluida en Crt *)
 Window(xpos,ypos,xpos+xlong,ypos+ylong);
 ClrScr;
 (* Dibujo del contorno de la ventana *)
 GotoXY(1,1);Write(caja[1]);
 FOR i:=2 TO xlong-1 DO Write(caja[2]);
 Write(caja[3]);
 FOR i:=2 TO ylong-1 DO
 BEGIN
 GotoXY(1,i);
 Write(caja[4]);
 GotoXY(xlong,i);
 Write(caja[4]);
 END;
 GotoXY(1,ylong);Write(caja[5]);
 FOR i:=2 TO xlong-1 DO Write(caja[2]);
 Write(caja[6]);
 GotoXY(1,1);Write(titulo);
 Window(Lo(esquinal),Hi(esquinal),Lo(esquina2),Hi(esquina2));
END;

PROCEDURE Ventana.mueve;
BEGIN
 Borra;
 IF (x>1) AND (x+xlong<80) THEN xpos:=x;
 IF (y>1) AND (y+ylong<25) THEN ypos:=y;
 Dibuja;
END;

PROCEDURE Ventana.Borra;
VAR
 esquinal,esquina2: word;
BEGIN
 esquinal:=WindMin;
 esquina2:=WindMax;
 Window(xpos,ypos,xpos+xlong,ypos+ylong);
 ClrScr;
 Window(lo(esquinal),hi(esquinal),lo(esquina2),hi(esquina2));
END;

FUNCTION Ventana.CogeX;
BEGIN
 CogeX:=xpos;
END;

FUNCTION Ventana.CogeY;
BEGIN
 CogeY:=ypos;
END;

(* FIN DE IMPLEMENTACION DE LOS METODOS DEL TIPO OBJETO: Ventana *)

```

## EJERCICIOS RESUELTOS

```
PROCEDURE Oculta_cursor;
VAR regs:registers;
BEGIN
 regs.CH:=$20;
 regs.AH:=$01;
 Intr($10,regs);
END;
FUNCTION Modo_pantalla:integer;
VAR regs:registers;
BEGIN
 regs.AH:=15;
 Intr($10,regs);
 Modo_pantalla:=regs.AL;
END;

PROCEDURE Muestra_cursor;
VAR regs:registers;
 modo_valido:boolean;
BEGIN
 modo_valido:=FALSE;
 CASE Modo_pantalla OF
 3 : BEGIN (* Modo color *)
 Modo_valido:=TRUE;
 regs.AH:=1;
 regs.CH:=6;
 regs.CL:=7;
 END;
 7 : BEGIN (* Modo monocromo *)
 Modo_valido:=TRUE;
 regs.AH:=1;
 regs.CH:=12;
 regs.CL:=13;
 END;
 END;
 IF Modo_valido THEN Intr($10,regs);
END;

VAR
 pantalla:ARRAY [1..3] OF Ventana;
 i,x,y:integer;

(**** Programa Principal ****)
BEGIN
 Window(1,1,80,25);
 ClrScr;
 Oculta_cursor;
 pantalla[1].crea(10, 1,15, 5,'P1');
 pantalla[2].crea(40,10, 5,10,'P2');
 pantalla[3].crea(70,20, 5, 2,'P3');

 (* Se mueven las ventanas aleatoriamente por la pantalla *)

 WHILE NOT KeyPressed DO
 FOR i:=1 TO 3 DO
 BEGIN
 x:=random(3)-1;
 y:=random(3)-1;
 WITH pantalla[i] DO Mueve(CogeX+x,CogeY+y);
 Delay(30);
 END;
 Readln;
 Muestra_cursor;
 END.
```

- 13.3** Construir una *unit* denominada *Figuras*, que contenga una jerarquía de tipos objeto sobre puntos y círculos, de tal forma que se puedan extender a otras figuras. Las operaciones a implementar serán: *Mostrar* (dibuja la figura), *Ocultar* (borra la figura), *Arrastrar* (se mueve la figura con las flechas del teclado), *Expandir* (amplia la figura), y *Contraer* (comprime la figura).

### Solución

```

UNIT Figuras;

INTERFACE

USES Graph, Crt;
TYPE
 Ubicacion = OBJECT
 x,y : integer;
 PROCEDURE Iniciar(inicialX, inicialY : integer);
 FUNCTION ObtenerX : integer;
 FUNCTION ObtenerY : integer;
 END;

 puntoPtr = ^punto;
 punto = OBJECT (Ubicacion)
 visible : boolean;
 CONSTRUCTOR Iniciar(InicialX, InicialY : integer);
 DESTRUCTOR Eliminar; VIRTUAL;
 PROCEDURE Mostrar; VIRTUAL;
 PROCEDURE Ocultar; VIRTUAL;
 FUNCTION EsVisible : boolean;
 PROCEDURE MoverA(NuevaX, NuevaY : integer);
 PROCEDURE Arrastrar(ArrastrarPor : integer); VIRTUAL;
 END;

 CirculoPtr = ^Circulo;
 Circulo = OBJECT (punto)
 Radio : integer;
 CONSTRUCTOR Iniciar(InicialX, InicialY : integer;
 RadioInicial : integer);
 PROCEDURE Mostrar; VIRTUAL;
 PROCEDURE Ocultar; VIRTUAL;
 PROCEDURE Expandir(ExpandirPor : integer); VIRTUAL;
 PROCEDURE Contraer(ContraerPor : integer); VIRTUAL;
 END;

IMPLEMENTATION
{-----}
{ Métodos del tipo objeto Ubicacion }
{-----}
PROCEDURE Ubicacion.Iniciar(InicialX, InicialY : integer);
BEGIN
 X := InicialX;
 Y := InicialY;
END;

FUNCTION Ubicacion.ObtenerX : integer;
BEGIN
 ObtenerX := X;
END;

FUNCTION Ubicacion.ObtenerY : integer;
BEGIN
 ObtenerY := Y;
END;

```

## EJERCICIOS RESUELTOS

```
{-----}
{ Métodos del tipo objeto punto }
{-----}

CONSTRUCTOR punto.Iniciar(inicialX, inicialY : integer);
BEGIN
 INHERITED Iniciar(inicialX, inicialY);
 visible := FALSE;
END;

DESTRUCTOR punto.Eliminar;
BEGIN
 Ocultar;
END;

PROCEDURE punto.Mostrar;
BEGIN
 visible := TRUE;
 PutPixel(X, Y, GetColor);
END;

PROCEDURE punto.Ocultar;
BEGIN
 visible := FALSE;
 PutPixel(X, Y, GetBkColor);
END;

FUNCTION punto.EsVisible : boolean;
BEGIN
 EsVisible := visible;
END;

PROCEDURE punto.MoverA(NuevaX, NuevaY : integer);
BEGIN
 Ocultar;
 X := NuevaX;
 Y := NuevaY;
 Mostrar;
END;

FUNCTION ObtenerDelta(VAR deltaX : integer;
 VAR deltaY : integer) : boolean;
{ Función auxiliar para el método arrastrar }
VAR
 tecla : char;
 salir : boolean;
BEGIN
 deltaX := 0; deltaY := 0; { 0 permite no cambiar de posición }
 ObtenerDelta := TRUE; { permite devolver un incremento Delta }
 REPEAT
 tecla := ReadKey;
 salir := TRUE;
 CASE Ord(tecla) OF
 0: BEGIN
 tecla := ReadKey; { 0 permite una tecla de 2 bytes }
 { Lee el segundo byte de la tecla }
 CASE Ord(tecla) OF
 72: deltaY := -1; { Flecha hacia arriba: decrementa Y }
 80: deltaY := 1; { Flecha hacia abajo : incrementa Y }
 75: deltaX := -1; { Flecha hacia izda. : decrementa X }
 77: deltaX := 1; { Flecha hacia dcha. : incrementa X }
 ELSE salir := FALSE; { Ignora cualquier otra tecla }
 END; { fin de CASE }
 END;
 13: ObtenerDelta := FALSE; { Tecla INTRO o CR }
 ELSE salir := FALSE; { Ignora cualquier otra tecla }
 END;
 UNTIL salir;
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```

 END; { fin de CASE }
 UNTIL salir;
END;

PROCEDURE punto.Arrastrar(ArrastrarPor : integer);
VAR
 deltaX, deltaY : integer;
 figuraX, figuraY : integer;
BEGIN
 Mostrar; { Muestra la figura que será arrastrada }
 figuraX := ObtenerX; { Obtiene la posición inicial de la figura }
 figuraY := ObtenerY;
 WHILE ObtenerDelta(deltaX, deltaY) DO
 BEGIN
 figuraX := figuraX + (deltaX * ArrastrarPor);
 figuraY := figuraY + (deltaY * ArrastrarPor);
 MoverA(figuraX, figuraY);
 END;
END;

{-----}
{ Métodos del tipo objeto círculo }
{-----}
CONSTRUCTOR Circulo.Iniciar(InicialX, InicialY : integer;
 RadioInicial : integer);
BEGIN
 INHERITED Iniciar(InicialX, InicialY);
 Radio := RadioInicial;
END;

PROCEDURE Circulo.Mostrar;
BEGIN
 visible := TRUE;
 Circle(X, Y, Radio);
END;

PROCEDURE Circulo.Ocultar;
VAR
 TempColor : Word;
BEGIN
 TempColor := GetColor;
 SetColor(GetBkColor);
 visible := FALSE;
 Circle(X, Y, Radio);
 SetColor(TempColor);
END;

PROCEDURE Circulo.Expander(ExpanderPor : integer);
BEGIN
 Ocultar;
 Radio := Radio + ExpanderPor;
 if Radio < 0 then Radio := 0;
 Mostrar;
END;

PROCEDURE Circulo.Contraer(ContraerPor : integer);
BEGIN
 Expander(-ContraerPor);
END;

END.

```



## EJERCICIOS RESUELTOS

- 13.4** Construir un programa que use la *unit* del ejercicio anterior para verificar su funcionamiento, y que además amplie la jerarquía con el tipo arco, que describe las figuras en forma de arco.

### Solución

```
PROGRAM Ejemplo_de_manejo_de_figuras (Input,Output);

USES Crt, DOS, Graph, Figuras;
TYPE
 arco = OBJECT (circulo)
 anguloInicial, anguloFinal : integer;
 CONSTRUCTOR Iniciar(Xinicial, Yinicial : integer;
 radioInicial : integer;
 angulo0inicial, angulo0final : integer);
 PROCEDURE Mostrar; VIRTUAL;
 PROCEDURE Ocultar; VIRTUAL;
 END;

VAR
 GraphDriver : integer;
 GraphMode : integer;
 ErrorCode : integer;
 UnArco : arco;
 UnCirculo : circulo;
 i: integer;

{-----}
{ Declaraciones de los métodos del tipo objeto arco }
{-----}
CONSTRUCTOR arco.Iniciar;
BEGIN
 INHERITED Iniciar(Xinicial, Yinicial, RadioInicial);
 anguloInicial := angulo0inicial;
 anguloFinal := angulo0final;
END;

PROCEDURE arco.Mostrar;
BEGIN
 visible:=true;
 Arc(X, Y, anguloInicial, anguloFinal, Radio);
END;

PROCEDURE arco.Ocultar;
var
 TempColor : Word;
BEGIN
 visible:=false;
 TempColor := GetColor;
 SetColor(GetBkColor);
 { Dibuja el arco con el color de fondo, ocultándolo }
 Arc(X, Y, anguloInicial, anguloFinal, Radio);
 SetColor(TempColor);
END;

{-----}
{ Programa principal }
{-----}
BEGIN
 GraphDriver := Detect;
 DetectGraph(GraphDriver, GraphMode);
 InitGraph(GraphDriver, GraphMode, 'c:\tp\bgi');
 if GraphResult <> GrOK then
 BEGIN
 WriteLn('Error en dispositivo gráfico:');
 END;
 END;
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```

 GraphErrorMsg(GraphDriver));
 Halt(1)
END;

UnCirculo.Iniciar(151, 82, 50);
 { X,Y inicialmente en 151,82 }
 { Radio inicial de 50 pixels }
UnCirculo.Arrastrar(5);
 { El parametro 5 indica que el círculo
 se arrastra de 5 en 5 pixels. }
UnCirculo.Expander(50);
UnCirculo.Arrastrar(3);
UnCirculo.Ocultar;

UnArco.Iniciar(151, 82, 25, 0, 90);
 { X,Y inicialmente en 151,82 }
 { Radio inicial de 50 pixels }
 { Angulo inicial: 0; Angulo final: 90 }
UnArco.Arrastrar(10);
UnArco.Expander(100);
UnArco.Arrastrar(3);
UnArco.Ocultar;

CloseGraph;
RestoreCRTMode;
END.

```

- 13.5** Construir un programa que use la *unit* del ejercicio 13.3, y el tipo objeto arco del ejercicio 13.4 para construir una lista de objetos. Se deberán usar *destructores*. Para verificar el correcto comportamiento de los destructores mostrar en pantalla la memoria *heap* libre antes de construir la lista y la memoria *heap* que queda después de eliminar la lista (si coinciden indicará que el *destructor* gestiona correctamente la memoria).

### Solución

```

PROGRAM ManejaListasDeFiguras (Input, Output);
USES Graph, Figuras;

TYPE
 arcoPtr = ^arco;
 arco = OBJECT(circulo)
 anguloInicial, anguloFinal : Integer;
 CONSTRUCTOR Iniciar(Xinicial, Yinicial : Integer;
 RadioInicial : Integer;
 unAnguloInicial, unAnguloFinal : Integer);
 PROCEDURE Mostrar; VIRTUAL;
 PROCEDURE Ocultar; VIRTUAL;
 END;

 NodoPtr = ^Nodo;
 Nodo = RECORD
 Info : puntoPtr;
 Sig : NodoPtr;
 END;

```

## EJERCICIOS RESUELTOS

```
ListaPtr = ^Lista;
Lista = OBJECT
 Nodos: NodoPtr;
 CONSTRUCTOR Iniciar;
 DESTRUCTOR Eliminar; VIRTUAL;
 PROCEDURE Insertar(Info : puntoPtr);
 PROCEDURE Mostrar;
END;

VAR
 GraphDriver : Integer;
 GraphMode : Integer;
 Temp:STRING;
 unaLista : Lista;

{-----}
{ Procedimientos que no son métodos }
{-----}

PROCEDURE OutTextLn(ElTexto : String);
BEGIN
 OutText(ElTexto);
 MoveTo(0, GetY+12);
END;

PROCEDURE HeapStatus(Mensaje : String);
BEGIN
 Str(MemAvail : 6, Temp);
 OutTextLn(Mensaje+Temp);
END;

{-----}
{ Implementación de los métodos del tipo objeto arco }
{-----}

CONSTRUCTOR arco.Iniciar;
BEGIN
 INHERITED Iniciar(Xinicial, Yinicial, RadioInicial);
 anguloInicial := UnAnguloInicial;
 anguloFinal := UnAnguloFinal;
END;

PROCEDURE arco.Mostrar;
BEGIN
 Visible := True;
 Arc(X, Y, anguloInicial, anguloFinal, radio);
END;

PROCEDURE arco.Ocultar;
VAR
 TempColor : Word;
BEGIN
 TempColor := GetColor;
 SetColor(GetBkColor);
 Visible := False;
 Arc(X, Y, anguloInicial, anguloFinal, radio);
 SetColor(TempColor);
END;

{-----}
{ Implementación de los métodos del tipo objeto Lista }
{-----}

CONSTRUCTOR Lista.Iniciar;
BEGIN
 Nodos := nil;
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```

DESTRUCTOR Lista.Eliminar;
VAR
 p : NodoPtr;
BEGIN
 while Nodos <> nil do
 BEGIN
 p := Nodos;
 Nodos := p^.Sig;
 Dispose(p^.Info, Eliminar);
 Dispose(p);
 END;
END;

PROCEDURE Lista.Insertar(Info : puntoPtr);
var
 p : NodoPtr;
BEGIN
 New(p);
 p^.Info := Info;
 p^.Sig := Nodos;
 Nodos := p;
END;

PROCEDURE Lista.Mostrar;
{Muestra los objetos y permite moverlos}
VAR
 Pactual : NodoPtr;
BEGIN
 Pactual := Nodos;
 WHILE Pactual <> NIL DO
 BEGIN
 Str(Pactual^.Info^.ObtenerX : 3, Temp);
 OutTextLn('X = '+Temp);
 Str(Pactual^.Info^.ObtenerY : 3, Temp);
 OutTextLn('Y = '+Temp);
 Pactual^.Info^.Arrastrar(5);
 Pactual := Pactual^.Sig;
 END;
END;

{-----}
{ Programa Principal }
{-----}

BEGIN
 DetectGraph(GraphDriver, GraphMode);
 InitGraph(GraphDriver, GraphMode, 'd:\tp\bgi');
 IF GraphResult <> GrOK THEN
 BEGIN
 WriteLn('Error en dispositivo gráfico: ',
 GraphErrorMsg(GraphDriver));
 Halt(1);
 END;

 HeapStatus('Espacio libre Heap antes de construir la Lista: ');

 unaLista.Iniciar;
 unaLista.Insertar(New(arcoPtr, Iniciar(151, 82, 25, 200, 330)));
 unaLista.Insertar(New(CirculoPtr, Iniciar(400, 100, 40)));
 unaLista.Insertar(New(CirculoPtr, Iniciar(305, 136, 5)));
 unaLista.Mostrar;

 HeapStatus('Espacio Heap libre después de construir la Lista: ');

 unaLista.Eliminar;

 HeapStatus('Espacio Heap libre después de borrar la Lista: ');

```

## EJERCICIOS RESUELTOS

```
 OutText('Pulse INTRO para finalizar el programa: ');
 Readln;
 CloseGraph;
 RestoreCRTmode;
END.
```

**13.6** Combinar la pila genérica (*unit Pila*), construida en el ejemplo 13.16, con la jerarquía de tipos objeto del ejercicio anterior para crear una pila de objetos gráficos.

### Solución

```
PROGRAM ManejaListasDeFiguras (Input, Output);
USES Graph, Figuras, Pila;

TYPE
 arcoPtr = ^arco;
 arco = OBJECT(circulo)
 anguloInicial, anguloFinal : Integer;
 CONSTRUCTOR Iniciar(Xinicial, Yinicial : Integer;
 RadioInicial : Integer;
 unAnguloInicial, unAnguloFinal : Integer);
 PROCEDURE Mostrar; VIRTUAL;
 PROCEDURE Ocultar; VIRTUAL;
 END;

VAR
 GraphDriver : Integer;
 GraphMode : Integer;
 Temp:STRING;
 unaPila : TpilaHeap;
 p:puntoPtr;

PROCEDURE OutTextLn(ElTexto : String);
BEGIN
 OutText(ElTexto);
 MoveTo(0, GetY+12);
END;

{-----}
{ Implementación de los métodos del tipo objeto arco }
{-----}

CONSTRUCTOR arco.Iniciar;
BEGIN
 INHERITED Iniciar(Xinicial, Yinicial, RadioInicial);
 anguloInicial := UnAnguloInicial;
 anguloFinal := UnAnguloFinal;
END;

PROCEDURE arco.Mostrar;
BEGIN
 Visible := True;
 Arc(X, Y, anguloInicial, anguloFinal, radio);
END;

PROCEDURE arco.Ocultar;
VAR
 TempColor : Word;
BEGIN
 TempColor := Graph.GetColor;
 SetColor(GetBkColor);
 Visible := False;
END;
```

## PROGRAMACION ORIENTADA A OBJETOS

```
Arc(X, Y, anguloInicial, anguloFinal, radio);
SetColor(TempColor);
END;

{-----}
{ Programa Principal }
{-----}

BEGIN
 DetectGraph(GraphDriver, GraphMode);
 InitGraph(GraphDriver, GraphMode, 'd:\tp\bgi');
 IF GraphResult <> GrOK THEN
 BEGIN
 WriteLn('Error en dispositivo gráfico: ',
 GraphErrorMsg(GraphDriver));
 Halt(1);
 END;
 unaPila.Inicializar(SizeOf(puntoPtr));

 unaPila.Meter(New(arcoPtr, Iniciar(151, 82, 25, 200, 330)));
 unaPila.Meter(New(CirculoPtr, Iniciar(400, 100, 40)));
 unaPila.Meter(New(CirculoPtr, Iniciar(305, 136, 5)));
 unaPila.Meter(New(arcoPtr, Iniciar(100, 50, 10, 0, 180)));
 unaPila.Meter(New(puntoPtr, Iniciar(333, 666)));

 REPEAT
 unaPila.Sacar(p);
 Str(p^.ObtenerX:3,Temp);
 OutTextLn('X = '+Temp);
 Str(p^.ObtenerY:3,Temp);
 OutTextLn('Y = '+Temp);
 UNTIL unaPila.estaVacía;

 unaPila.Destruir;

 OutText('Pulse INTRO para finalizar el programa: ');
 Readln;
 CloseGraph;
 RestoreCRTmode;
END.
```

### 13.13. EJERCICIOS PROPUESTOS

- 13.7** Diseñar una *unit* denominada *vectores* que contenga una jerarquía de tipos objeto que permita manejar las operaciones básicas de los vectores empleados habitualmente en Matemáticas, Física, etc... Ha de tenerse en cuenta que los vectores pueden ser bidimensionales o tridimensionales. Las operaciones habituales son: suma, diferencia, producto escalar, producto vectorial, y producto de un vector por una constante. Usar la *unit* en un programa para verificar su correcto funcionamiento.
- 13.8** Diseñar una *unit* denominada *matrices* que contenga una jerarquía de tipos objeto que permita manejar las operaciones básicas de las matrices de números reales empleadas habitualmente en Matemáticas. Ha de tenerse en cuenta que el constructor de la matriz ha de recibir como parámetros las dimensiones de la matriz. La matriz

## EJERCICIOS PROPUESTOS

se tiene que implementar internamente como una estructura dinámica de datos y como un fichero, al estilo del ejemplo 13.16. Las operaciones que deben hacerse sobre el tipo matriz son: inicializar toda la matriz a un valor determinado, introducir un valor en una posición determinada de la matriz, escribir la matriz, calcular la matriz traspuesta, producto de una constante por la matriz, si la matriz es cuadrada calcular el determinante, calcular la inversa cuando sea posible, calcular el producto de matrices, y resolver un sistema de ecuaciones representado por la matriz, añadiendo el vector de términos independientes. Usar la *unit* en un programa para verificar su correcto funcionamiento.

- 13.9** Escribir una *unit* para implementar árboles binarios genéricos. Utilizarla desde un programa para realizar búsquedas de fichas de alumnos.
- 13.10** Escribir una *unit* denominada *calcula*, que implemente las operaciones habituales de una calculadora científica sobre números reales. Usarla desde un programa para comprobar su funcionamiento.
- 13.11** Modificar el ejercicio anterior para que la calculadora opere tanto sobre reales, como sobre números complejos.
- 13.12** Mejorar el tipo objeto *ventana* del ejercicio 13.2, para que admita colores, fondos, y se pueda guardar en memoria la ventana y se pueda también restaurar en cualquier momento.
- 13.13** Diseñar una *unit* con una jerarquía de clases para almacenar las propiedades de objetos tridimensionales (esferas, cilindros, toroides, cubos, tetraedros, conos, etc...). Escribir un programa que la use para probarla.
- 13.14** Diseñar una *unit* para realizar operaciones de bits sobre enteros. Deberan implementarse métodos para todas las operaciones de bits, y para las operaciones de lectura y escritura de bits.
- 13.15** Diseñar con metodología orientada a objetos una aplicación de reservas hoteleras de la forma más genérica posible. Programarla en *units*. Para probarla se debe de crear un programa principal que usa dichas *units*.

### 13.14. AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Las principales ideas de la programación orientada a objetos provienen del proyecto Simula. *Nydgaard* y *Dahl* relatan la historia del proyecto *Simula* en la obra de *R.L. Wexelblat (ed.)* titulada *History of programming languages*, editada por *Academic Press (1981)*. *Simula I* fue un lenguaje para describir y programar simulaciones. La experiencia con *Simula I* llevó a la conclusión de que los datos y las operaciones sobre ellos son una misma cosa, y que podrían ahorrarse esfuerzos si se programaran con anterioridad las propiedades comunes de los objetos. Las clases de objetos surgieron como el concepto central de un nuevo lenguaje de propósito general, *Simula 67*, diseñado en 1967. Como texto de *Simula* puede consultarse la obra de *B. Kirkerud* titulado *Object-Oriented Programming with SIMULA* publicado por *Addison-Wesley (1989)*.

Sobre el lenguaje *Smalltalk* puede consultarse la obra *Smalltalk-80, the language* de *A. Goldberg* y *D. Robson*, publicada por *Addison-Wesley (1989)*.

Las principales referencias sobre el lenguaje *Eiffel* son las obras de *B. Meyer* tituladas: *Object-Oriented software construction (Prentice Hall, 1988)* y *Eiffel, the language (Prentice Hall, 1992)*.

El Turbo Pascal en su versión 7.0 carece de algunas características que incorporan otros lenguajes orientados a objetos, y que es posible que en un futuro se le incorporen siguiendo la línea del lenguaje C++. Así por ejemplo C++ incorpora *herencia múltiple*, *sobrecarga de operadores* y *funciones*, *templates* (que implementan directamente la genericidad), y *manejo de excepciones*. Las principales referencias de C++ son las obras de *Ellis M.A.* y *Stroustrup B.* titulada *The annotated C++ reference manual, ANSI base document* publicada por *Addison-Wesley (1990)*; y la segunda edición de la definición del lenguaje C++ de *B. Stroustrup* con el título en castellano *El lenguaje de programación C++*, publicado por *Addison-Wesley/Díaz de Santos (1993)*.

Respecto a bibliografía específica de POO con Turbo Pascal no es amplia, y en general profundiza poco, se pueden citar las obras: *Object-Oriented programming with Borland Pascal 7* de *N.M. Shammis* editado por *SAMS (1993)*; *Object-Oriented programming with Turbo Pascal* del mismo autor en la editorial *Wiley (1990)*; *Object-Oriented programming with Turbo Pascal 5.5* de *B. Ezzel* en la editorial *Addison-Wesley (1989)*. También incluye capítulos dedicados a la POO el libro *Borland Pascal developer's guide* de *E. Mitchell* publicado en la editorial *QUE (1993)*.

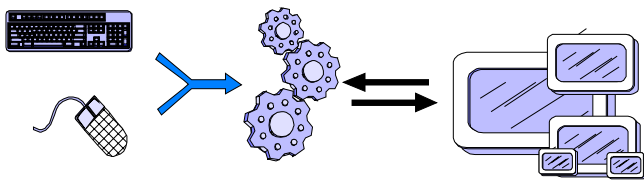
También existen revistas especializadas en POO como la *Journal of Object-Oriented Programming* publicada por *SIGS*, y que trata de los distintos aspectos de la POO desde diferentes ópticas. También se pueden citar las revistas: *OOPS Messenger* publicada por *ACM (Association for Computing Machinery)*; y *Object Magazine* publicada por *SIGS*.

Otra fuente de información sobre POO son los congresos y las publicaciones con las comunicaciones presentadas a dichos congresos. En España se celebró *INFOOP'93 (I Congreso español de POO y C++)*, cuyo libro de sesiones está publicado por *R. Devis (INFO+, 1993)*. A nivel



#### AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

internacional ACM organiza el congreso anual denominado *OOPSLA (Conference on Object-Oriented Programming Systems, Languages and Applications)*, cuyas comunicaciones publica también ACM.



## CAPITULO 14

### MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

#### CONTENIDOS

- 14.1 Introducción
- 14.2 Marcos de aplicación
- 14.3 Programación dirigida por eventos
- 14.4 Turbo Vision: un marco de aplicación en modo texto
- 14.5 Las vistas en Turbo Vision
- 14.6 Los grupos en Turbo Vision
- 14.7 Eventos en Turbo Vision
- 14.8 Utilización de los tipos objeto de Turbo Vision
- 14.9 Ejercicios propuestos
- 14.10 Ampliaciones y notas bibliográficas

## 14.1 INTRODUCCION

La programación de interface de usuario adquiere día a día mayor cantidad de seguidores a partir del éxito comercial alcanzado por el entorno *MS Windows*. En este capítulo se presenta el concepto de *marcos de aplicación* y de *programación dirigida por eventos*, como pilares fundamentales para la programación de interfaces de usuario. Los marcos de aplicación permiten demostrar la potencia que la programación orientada a objetos puede poner a disposición de los programadores y diseñadores de software al aportar la reutilización del código.

Como exposición práctica de los nuevos conceptos explicados se analizará un caso concreto de marco de aplicación como es *Turbo Vision*<sup>®</sup>. Para ello se presentará su jerarquía de clases, como utilizarlas para desarrollar aplicaciones a través de pequeños ejemplos que ilustren su manejo y finalmente se desarrollara un caso complejo implementándolo con este marco de aplicación. Este marco de aplicación, a diferencia de las aplicaciones generadas para Windows con una interface gráfica, utiliza la programación de interfaces para usuario en modo texto.

Los *marcos de aplicación* (*application frameworks*) son tipos de *librerías*<sup>30</sup> de clases que facilitan la tediosa, monótona y engorrosa tarea de desarrollar aplicaciones que requieren una interface de usuario sofisticada. Estrictamente hablando, los marcos de aplicación ponen a disposición de las aplicaciones un soporte en tiempo de ejecución. Este soporte normalmente incluye

---

**30** Una librería o también llamada biblioteca es una colección de módulos, subprogramas y/o utilidades disponibles para uso común dentro de unas condiciones de explotación; los elementos individuales no necesitan estar relacionados. Por regla general, únicamente es necesario referenciar el elemento de la librería para que se incorpore de forma automática a un programa de usuarios.

En una librería de clases los elementos que la forman son clases que encapsulan los datos y métodos miembros que definen su funcionalidad. Por lo tanto el acceso a estos se realizará indicando a la clase que pertenecen que se encuentra dentro de la librería.

La programación orientada a objetos (POO) se caracteriza por la programación mediante el uso de librerías de clases. Una de las ventajas más palpables de la POO reside en el aprovechamiento del trabajo realizado por otros programadores. Se utilizan clases ya desarrolladas cuyo código ya ha sido desarrollado, probado y depurado antes de ponerlas dentro de la librería de clases.

Las librerías de clases podrían verse como librerías estándar, pero difieren significativamente de estas. Las librerías estándar para lenguajes procedurales tipo **C** o **Pascal** no tienen la flexibilidad de las librerías de los lenguajes orientados a objetos. Si a un usuario no le satisface el comportamiento de una función o procedimiento de una librería estándar, no lo podrá modificar. Si por ejemplo el 95 por ciento de un procedimiento se adecúa a las necesidades de un determinado usuario, este no podrá aprovechar ese código y deberá reescribir la totalidad del mismo, con la consiguiente pérdida de tiempo y de código. Esto no ocurre en las librerías de clases, al poder utilizar la herencia para crear nuevas clases a partir de clases ya definidas y además el enlace dinámico, al permitir extender la funcionalidad de una clase sin tener acceso al código fuente de la misma.

algo más que los recursos típicos de una librería para el manejo de *ventanas*<sup>31</sup>. Los marcos de aplicación están diseñados para generalizar la entrada y salida de los programas, al liberar de la tarea de controlar la entrada a través del ratón y teclado, y ofrecer un soporte para la salida por medio de una variedad de tipos de ventanas

Los componentes de un marco de aplicación que se estudiarán más adelante se pueden clasificar de una forma general en *vistas*, *controles* y *modelos*. Las **vistas** son la interface de la aplicación (parte visible); también se les da el nombre de *ventanas*. Los **controles** incluyen objetos visibles e invisibles que manejan la entrada de usuario y la traducen en acciones a través del envío de *mensajes* a los componentes del modelo apropiado. Los **modelos** se encargan de manipular los datos cuando se le es ordenado por los controles. Cada modelo componente de un programa normalmente tienen una vista asociada con él. La vista utiliza los datos del modelo para presentar la información al usuario. Los modelos son las partes del marco de aplicación que deben ser suministradas por el programador. Las vistas pueden ser de texto o gráficas según sea el dispositivo de salida (pantalla) que se utilice.

Los marcos de aplicación facilitan la tarea de comunicación con el usuario al hacer más sencilla la tarea de construir ventanas, menús y cajas de diálogo, a la vez que generalizan la manera en que se realiza el control del programa. Esto libera al programador de tener que definir explícitamente como los componentes de un programa deben monitorizar y responder a las entradas de ratón y teclado. Los marcos de aplicación implementan un mecanismo de control a través del cual todas las entradas son encaminadas a un distribuidor o planificador (*scheduler, dispatcher*) central. Todas las entradas procedentes de cualquier dispositivo de E/S (teclado, ratón, puerto de comunicaciones, controlador de disco, ...) son clasificadas como **eventos** o **sucesos**. El distribuidor central monitoriza todos los eventos. A los sistemas que poseen este mecanismo de control se les denomina *sistemas dirigidos por eventos*.

La forma de realizar el control del programa en un sistema dirigido por eventos es diferente a como se realiza en una aplicación tradicional. En vez de especificar la secuencia de acciones a seguir por el programa, el programador especifica los diferentes tipos de eventos a los cuales puede responder el programa y las acciones a realizar para los distintos sucesos que puedan ocurrir a lo largo de la ejecución del programa. Las acciones se realizan por medio de los *métodos* implementados por el programador para responder o manejar los distintos eventos que se produzcan.

## 14.2 MARCOS DE APLICACION

La mayoría de los marcos de aplicación permiten el manejo de objetos polimórficos como listas, conjuntos, diccionarios, pilas, colas, strings y flujos de datos o ficheros por medio de unas

---

**31** *Area rectangular en una pantalla dentro de la cual se representa parte de una imagen, de un fichero o información de cualquier tipo. La ventana puede ocupar el tamaño de la pantalla, pudiéndose visualizar más de una simultáneamente (superpuestas o en áreas distintas de la pantalla).*

## MARCOS DE APLICACION

clases básicas. No todos los marcos de aplicación incluyen estos tipos de objetos, pero si al menos un subconjunto de ellos y probablemente otros. Además, la mayoría de los marcos proporcionan clases para la visualización de datos en un entorno de ventanas. Unos soportan un sistema de ventanas en modo texto (**COWS** *character-oriented window systems*) y otros una interface gráfica de usuario (**GUI** *graphical user interface*), para lo cual, en cualquiera de los casos, implicará dar soporte para la creación y manipulación de objetos como por ejemplo *punto* o *rectángulo*.

A modo de ejemplo, dado su éxito comercial, podíamos citar el entorno *Microsoft Windows*<sup>®</sup> como un marco de aplicación, aunque normalmente no se describe de este modo. Este entorno funciona por medio de un esquema de envío de mensajes: *los mensajes se envían a objetos sin conocer su tipo exacto, esperando que estos respondan apropiadamente a los primeros*. Dado que *Microsoft Windows* está desarrollado básicamente en C y ensamblador, no es realmente orientado a objetos, pero para suplir esta cuestionable deficiencia se utilizan algunos *trucos* para que se comporte de una forma compatible con la programación orientada a objetos. La solución consiste en colocar una concha (*shell*) sobre el entorno *Windows* que permita desarrollar aplicaciones para este entorno totalmente orientadas a objetos. Con este objetivo existen librerías de clases para C++ y Pascal orientado a objetos que simplifican la interface con las funciones de *Windows*, facilitando enormemente la construcción de software para ejecutarse bajo este entorno. Para darnos una idea de la complejidad de programación en entorno *Windows* baste citar que las herramientas suministradas para la programación de interfaces que componen la gigantesca Interface para la Programación de Aplicaciones (*API*) tiene cerca de 1000 funciones<sup>32</sup> que a su vez agrupan subfunciones determinadas por combinaciones particulares de sus parámetros. El *API* de *Windows* resulta un medio muy potente para la programación de aplicaciones y particularmente para la programación de GUIs, pero resulta bastante compleja y fatigosa la asimilación y adquisición de habilidad en su utilización.

Las características comunes de los marcos de aplicación podríamos resumirlas en dos:

- Estructura de los programas orientada a objetos
- Sistema de control dirigido por eventos.

La programación orientada a objetos la hemos estudiado en el capítulo anterior. Analizaremos la programación dirigida por eventos en la sección 14.3.

Los marcos de aplicación son librerías de clases diseñadas específicamente para facilitar la creación y mantenimiento del software. Por ello, estas librerías incluyen las plantillas para la construcción de los objetos más comunes en programas de aplicaciones orientados a ventanas, permitiendo desarrollar interfaces de usuario similares a los de las aplicaciones comerciales más actuales como por ejemplo el entorno de desarrollo integrado (IDE) de lenguajes *Turbo* de *Borland*, hojas de cálculo, procesadores de texto y gestores de bases de datos. Normalmente estos programas tienen una barra de menús en la parte superior de la pantalla y una línea de estado en la parte

---

<sup>32</sup> Aproximadamente 600 funciones en la versión de *Windows 3.0*, y 1000 en la versión *3.1*

inferior. La barra de menús permite acceder a menús desplegados, al seleccionarlos con el ratón o una combinación de teclas, que presentan una lista de posibles opciones de igual forma seleccionables para realizar determinadas acciones del programa. Las combinaciones de teclas que permiten activar una acción de forma alternativa a la selección por ratón se indican en las propias opciones de los menús desplegados o en la línea de estado. También resultará familiar en estos entornos el uso de cajas de diálogo para la entrada y salida de datos. En cualquier caso los estudiaremos más adelante dentro de este mismo apartado en el epígrafe *Componentes de un programa*.

## ESTRUCTURA Y ORGANIZACION DE UN PROGRAMA

### PARADIGMA Modelo/Vista/Control

Hasta aquí hemos utilizado los términos *programa*, *aplicación* y *modelo* de una forma confusa, ambigua e incluso indistintamente en contextos similares. Estos y otros términos vamos a tratar de aclararlos a continuación, para explicar el *paradigma*<sup>33</sup> Modelo/Vista/Control (MVC).

En el contexto de los marcos de aplicación se suele usar los términos *programa*, *aplicación* y *modelo* de forma indistinta. En la programación estructurada tradicional, el término **programa** hace referencia a una unidad ejecutable de software que puede utilizarse para dirigir el comportamiento de un ordenador. Con el fin de facilitar la reutilización de código y minimizar las repercusiones de posibles modificaciones del software, la programación orientada a objetos facilita el descomponer un programa en partes o capas de un nivel conceptual superior (menor abstracción). Una parte del programa se encarga de manipular y mantener los datos (p.ej. el texto que se edita en un procesador de textos), otra de su presentación al usuario de una forma comprensible y correcta, y una tercera que interactúa con el usuario para aceptar las ordenes de éste, a través de los dispositivos de entrada, y pasarlas a la aplicación.

La parte del programa (como sistema o unidad software) que mantiene y manipula los datos se denomina **modelo**. A esta parte se le denomina también **aplicación** (dado que el *modelo* es el que define la funcionalidad de la aplicación) y, aunque puede resultar confuso, a veces también se le llama *programa*.

A la parte que presenta la información se la suele denominar **vista**. Una *ventana* es la forma más habitual de una vista.

La parte del programa que interactúa con el usuario se le llama **controlador**<sup>34</sup> (*controller*). Un programa normalmente contiene numerosos componentes de control como *botones*, *barras de desplazamiento*, ... El manejo de todos estos componentes se realiza por medio de un mecanismo de control centralizado. El término *controlador* hace referencia a este mecanismo y a la parte del software que implementa los componentes de control individuales.

---

<sup>33</sup> Método o conjunto de reglas para resolver un problema. Así por ejemplo el paradigma de la programación estructurada es una metodología desarrollada por varios autores (N. Wirth, E.W. Dijkstra, ...) a lo largo de la década de los 70 para diseñar y construir aplicaciones informáticas.

<sup>34</sup> Normalmente nos referiremos al **controlador** con la palabra "**control**".

## MARCOS DE APLICACION

Podemos resumir que el **paradigma MVC** involucra la separación de una aplicación en tres partes a distinguir:

- *Modelo* Representa la funcionalidad de la aplicación, es decir, la parte que manipula y mantiene los datos.  
Los objetos de la capa modelo implementan los distintos componentes que configuran el dominio del problema de la cada aplicación (*p.ej.* manejo de datos, base de datos o proceso/simulación).
- *Vista* Forma en que se presenta el *modelo* al usuario.  
Los objetos de la capa vista implementa los elementos gráficos de la aplicación (si el dispositivo de salida es gráfico) o elementos de presentación de forma más general (para cualquier tipo de salida: gráfica o de texto). Esta clase de objetos obtienen la información de la capa *modelo* y la presentan por medio de distintos mecanismo de visualización como por ejemplo gráficas, diagramas, editores, listas, etc.  
Este tipo de objetos tienen la capacidad de realizar transformaciones (*p.ej.* para mover o redimensionar una vista), recortes (*clipping*) en la salida (*p.ej.* presentar parte de una subvista que visualiza un fichero dentro de una vista de nivel superior) y otro tipo de acciones de control sobre la presentación. Esta capa podría contener subvistas que permitan subdividir la presentación del modelo de la aplicación. De esta forma, la aplicación tendría una vista que sirve de organizador, gestor y encargado de invocar a los métodos de las subvistas. Esta vista se le llama *vista principal o superior (top view)*.
- *Control* Modo en el que interactúa el usuario con el *modelo*. Recoge las ordenes y las envía al módulo apropiado de la aplicación.  
Esta capa constituye la interface entre los dispositivos de entrada y el *modelo* y la *vista*. Los objetos de esta parte de la aplicación no sólo manipulan dispositivos, sino que pueden realizar tareas de un nivel más alto como seguimiento de los desplazamientos del ratón o controlar las peticiones para redimensionar o mover ventanas. El *control* se encarga de la comunicación usuario-aplicación, así como de comunicar los eventos externos a los elementos de visualización del modelo de la aplicación.

La idea fundamental de este paradigma es la descomposición de una aplicación en capas o partes modificables de forma independiente. Esto permite cambiar la forma en que el usuario interactúa y ve una aplicación sin necesidad de reprogramar su estructura central. La flexibilidad de modificar las capas aporta mayor portabilidad entre diferentes entornos (tanto de hardware como de sistema operativo). La posibilidad de modificar la *vista*, o la *capa de control*, permite adaptar un programa a nuevos o diferentes dispositivos de visualización (*p.ej.* monitores de mayores prestaciones) o de selección (*p.ej.*, ratones de distinto tipo).

En la figura 14.1 se pueden observar las relaciones entre las tres partes de la aplicación y las direcciones de las llamadas de los métodos/procedimientos o envío de mensajes.

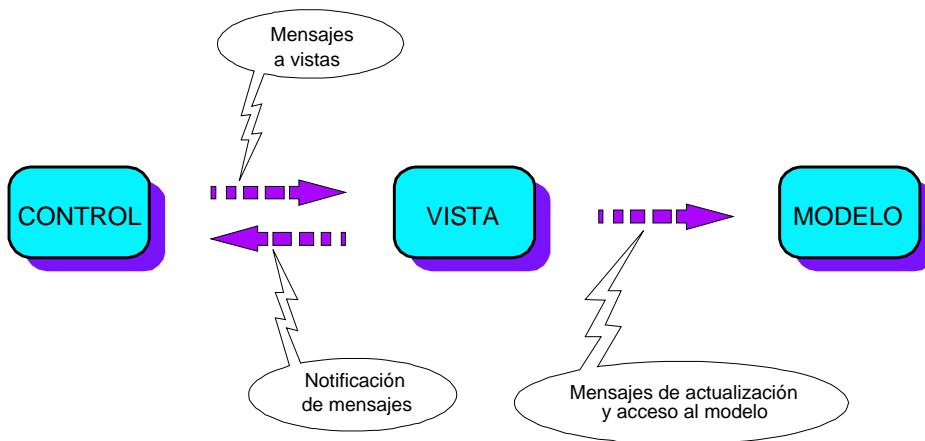


Fig. 14.1 Implementación del paradigma MVC

El *modelo* no contiene información sobre la *vista*. De esta manera, la *vista* puede ser reemplazada en cualquier momento sin repercusiones sobre el código de la aplicación. Los modelos siempre son referenciados por las vistas. Las vistas tienen las referencias de los objetos de sus modelos para poder invocar a los métodos que obtienen los datos a presentar o se encargan de modificarlos. Los objetos del modelo son totalmente pasivos y no contienen información que les haga depender de la *vista*. Dado que el *modelo* depende completamente de las especificaciones de cada aplicación, los marcos de aplicación no suelen aportar un soporte específico para su implementación.

La *vista* es el componente central, a la vez que el más sensible a cambios. Mantiene relación con el *modelo* y el *control*. El usuario no interactúa directamente con la *vista*; esta sólo presenta los datos al usuario que obtiene del modelo, y responde a los eventos generados por el control. El usuario interactúa directamente con el *control*, o mejor dicho, los componentes del control. A estos componentes se les denomina *vistas de control*. Estas vistas de control se programan para generar cierto tipo de mensajes cuando son activadas (bien por un *click* de ratón o una tecla aceleradora<sup>35</sup>). La *vista* también tiene a su vez subcomponentes. Por este motivo se establece una jerarquía de vistas y controles. La vista de nivel superior se denomina *vista principal* (*top-level view*, *main view* o *top view*). Y de igual manera al control principal se le llama *gestor de eventos de programa principal* o *bucle de eventos principal* (*main event loop*).

**35 Tecla aceleradora o atajo (hot-key).** Combinación de teclas (Ctrl+A, Alt+F, F1, ...) que al ser pulsadas desencadenan una acción sin acceder por una secuencia de menús.



## MARCOS DE APLICACION

El control principal siempre existe en un programa. Sin embargo los controles de nivel inferior, son creados por las vista principal o subvistas. Por ejemplo, cuando el usuario abre una vista de edición, esta vista crea automáticamente *controles de desplazamiento e iconos de control de la ventana* para moverla, redimensionarla o cerrarla. El control principal es un bucle que recibe todos los eventos de entrada, los procesa y envía a la vista principal. Este es el mecanismo principal de control del programa. Pero el programador también necesita mecanismo de control en forma de vistas para que el usuario pueda indicar que desea realizar cierto tipo de acciones. Por medio de un menú, un botón o una caja de verificación dentro de una vista, se genera un evento si el usuario activa esa vista de control. Una vez generado el evento el programador debe cerrar el ciclo de control mediante la definición de vistas y subvistas que contengan los manejadores de evento que respondan a las peticiones del usuario. Los manejadores de evento podrían actualizar vistas, cerrarlas o crear otras nuevas, además de poder modificar los datos del *modelo* subyacente o a sí mismas con información del modelo.

Es posible realizar otras organizaciones de un programa, pero la organización *Modelo/Vista/Control* es la más típica de los sistemas de software orientados a objetos.

La tarea más complicada para desarrollar un programa es el diseño e implementación del interface de usuario. Según las estimaciones de algunos expertos el 80 por ciento del código de una aplicación típica es el encargado de la interface de usuario. La interface de usuario incluye los componentes de la aplicación que se encargan de la presentación de los datos de usuario y de recoger las ordenes de usuario y enviarlas al módulo de la aplicación que las lleve a cabo.

Estos componentes son los más fácilmente generalizables, y esta es la tarea que desempeña un marco de aplicación por medio de una librería de clases reutilizables, que a menudo es portable entre distintos sistemas operativos y dispositivos de entrada y salida. Como consecuencia de la utilización de un marco de aplicación la carga de trabajo y tiempo de desarrollo de una aplicación se ve acortado considerablemente .

El inconveniente de los marcos de aplicación es el tiempo que se requiere dedicar hasta aprender a utilizarlos. Inicialmente parecen complicados, pero una vez se domina una de estas herramientas de desarrollo se da uno cuenta de que el tiempo invertido en su aprendizaje es rentable. Quizá sea un problema más complejo la propia decisión de qué marco de aplicación elegir para el desarrollo de nuestras aplicaciones. Algunos de los marcos de aplicación más conocidos comercialmente son:

Turbo Vision    C++ Views    Smalltalk    Actor    ObjectWindows

*Turbo Vision* es un marco de aplicación orientado a caracteres que permite la programación de interfaces de usuario en modo texto para el sistema operativo MS-DOS. Esta biblioteca de Borland se puede utilizar tanto desde el lenguaje *C++* como desde *Pascal*. *C++ Views* es un marco de aplicaciones desarrollado por CNS, Inc. con una estructura Modelo/Vista/Control, que permite

escribir programas con el lenguaje C++ para correr sobre el entorno Microsoft Windows. *Smalltalk* y *Actor* aparte de tener un lenguaje de programación y un entorno de desarrollo, son marcos de aplicación gracias a las sofisticadas librerías de clases que aportan ambos sistemas. *ObjectWindows* es la respuesta de la compañía Borland a C++ Views. Con *ObjectWindows* se generan aplicaciones para ejecutarse en Microsoft Windows y usa una jerarquía de clases similar a la de Turbo Vision. Además de utilizarse desde Borland Pascal y Borland C++, también trabaja con *Actor*.

## COMPONENTES DE UN PROGRAMA

Los componentes básicos de un programa aportados por un marco de aplicación son las *vistas*. Las vistas son objetos visibles utilizados para comunicar información al usuario a través de la pantalla. Las formas más comunes de las vistas son ventanas, vistas de control y vistas de datos. Una *ventana* es la forma más general de presentar un objeto, y pueden ser desde una región rectangular estática de la pantalla donde se presenta texto o gráficos, a objetos movibles con bordes, barras de desplazamiento, e iconos para mover, cerrar y redimensionar el marco de la ventana. Las ventanas normalmente presentan texto en pantalla, pero a menudo contienen otro tipo de vistas, como por ejemplo botones. Un botón (*pushbutton*) es un tipo de vista de control. Las vistas de control no se pueden mover, están fijadas en un aposición dentro de una ventana movible. Otro tipo de vista son las *vistas de datos* utilizadas para presentar estructuras de datos comunes como por ejemplo listas. Veamos distintos tipos de vistas.

### • Vistas y ventanas

Todos los objetos que se visualizan en los marcos de aplicación son vistas. El tipo más común de vista es las ventana. En la figura 14.2 se muestra una ventana de caracteres típica y una ventana gráfica de una aplicación Windows.

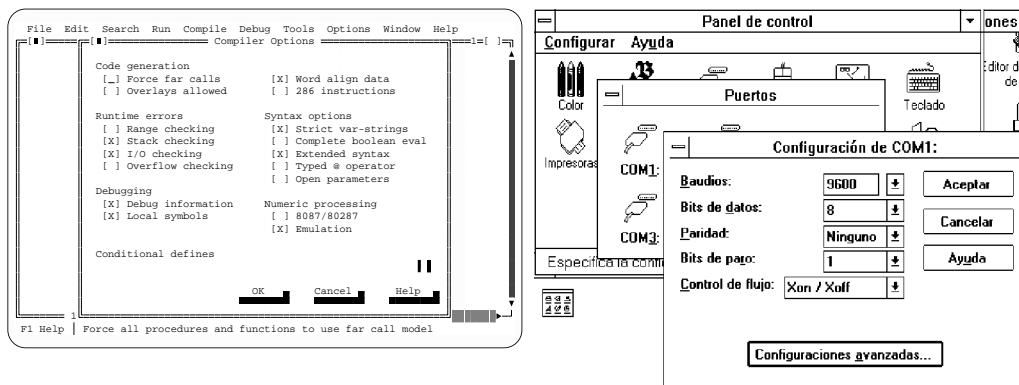


Fig. 14.2 Ventanas de texto y gráficas.

## MARCOS DE APLICACION

La mayoría de las ventanas pueden moverse, cambiarse su tamaño, o cerrarse, permitiendo presentar información —como un mensaje al usuario, o datos para que sean modificados por el usuario —como una línea de entrada.

La mayoría de los marcos de aplicación usan una *vista principal (top view)* denominada *display* o *desktop* (escritorio). En la figura 14.3 se muestra un desktop típico que contiene tres subvistas: una barra de menú, una línea de estado y una ventana de texto. Cada una de estas subvistas son vistas por sí mismas, y de hecho los componentes de la barra de menú en la parte superior de la pantalla o de la línea de estado en la parte inferior, también son vistas. Todo botón o dispositivo de control que aparece en la pantalla es una vista. Son vistas los botones **OK**, **Si**, **No** y **Cancelar** que aparecen en las cajas de diálogo así como iconos de control de las ventanas, como barras de desplazamiento, manejadores de tamaño, e iconos para cerrar, ampliar, maximizar y minimizar<sup>36</sup> ventanas.

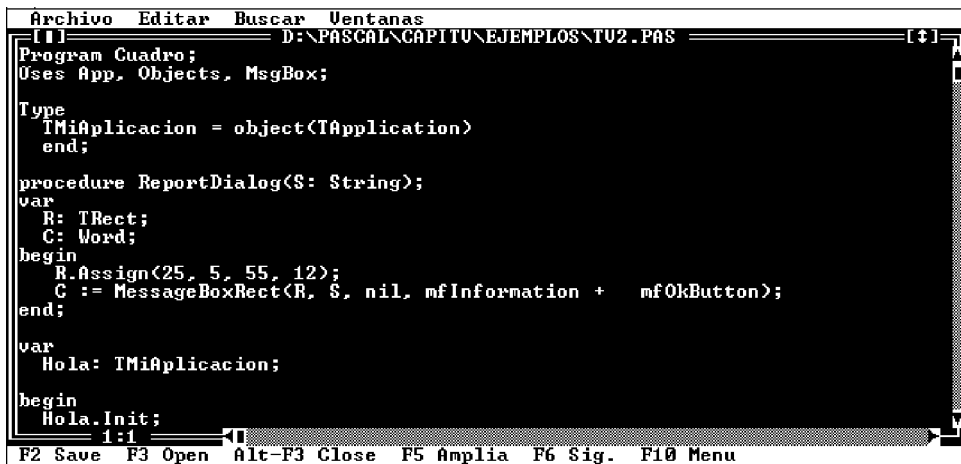
The image shows a screenshot of the Turbo Vision desktop environment. At the top, there is a menu bar with the following items: 'Archivo', 'Editar', 'Buscar', and 'Ventanas'. Below the menu bar is a status line that reads 'D:\PASCAL\CAPITULO\EJEMPLOS\TU2.PAS'. The main area of the desktop is a text window containing Pascal code. The code includes a 'Program Cuadro;' declaration, a 'Uses App, Objects, MsgBox;' statement, a 'Type' declaration for 'TMiAplicacion = object(TApplication)', and two procedures: 'ReportDialog(\$: String);' and 'Hola.Init;'. The 'ReportDialog' procedure defines a 'TRect' variable 'R' and a 'Word' variable 'C', and calls 'R.Assign(25, 5, 55, 12);' followed by 'C := MessageBoxRect(R, \$, nil, mfInformation + mfOkButton);'. The 'Hola.Init' procedure simply calls 'Hola.Init;'. At the bottom of the window, there is a status bar with keyboard shortcuts: 'F2 Save', 'F3 Open', 'Alt-F3 Close', 'F5 Amplia', 'F6 Sig.', and 'F10 Menu'.

Fig. 14.3. Desktop de Turbo Vision con tres subvistas: barra de menú, línea de estado y una ventana de texto

### • Menús

Un menú es un tipo especial de vista para presentar listas de elementos de menú. Un elemento de menú contiene a su vez otros objetos como subcomponentes. Un caso típico de componente de menú (figura 14.4) contiene una cadena de caracteres o etiqueta de la opción de menú, una especificación de una *hot-key* o acelerador, y una acción que será invocada al seleccionar el elemento de menú. Unos sistemas usan un puntero a una función, almacenado en el objeto elemento

<sup>36</sup> **Maximizar** una ventana consiste en hacer que esta ocupe la totalidad de la pantalla, de forma que si es la ventana activa no se pueda ver nada de ninguna otra. **Minimizar** consiste en reducir la ventana a su mínima expresión: un icono (en entornos gráficos) que se asocia a dicha ventana.

de menú; otros sistemas utilizan un entero asociado con un comando para indicar en cada elemento de menú la acción a realizar. En estos últimos cuando un elemento de menú se selecciona, se pasa el entero que referencia un único comando al mecanismo de control de eventos central, el cual en orden secuencial de llegada los va pasando al programa. El programa debe por tanto definir métodos para manejar cada comando específico.

Con este sistema de funcionamiento de los menús es posible diseñar e implementar un menú completo sin tener que definir las acciones que se realizarán cuando el menú funcione en la realidad. Esto facilita la construcción de aplicaciones al permitir desarrollar e implementar una parte de las mismas de manera totalmente independiente del resto.

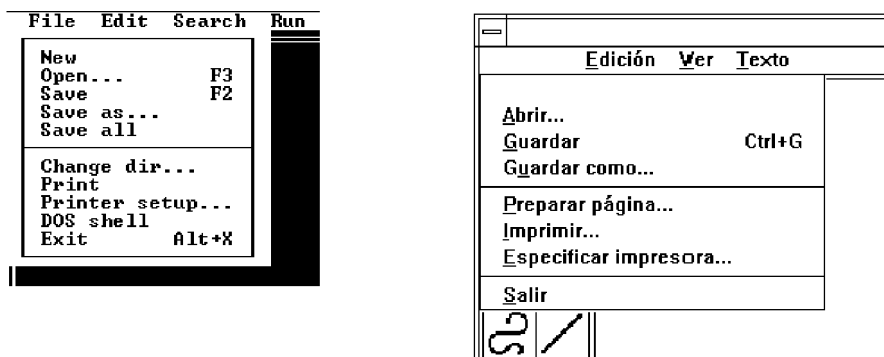


Fig. 14.4. Menú en modo texto y en modo gráfico.

### • Cajas de diálogo

Ofrecen un modo sencillo de especificar los parámetros de una aplicación controlados por el usuario (*p.ej.* nombre de un fichero a editar, el tipo de pantalla en el que se desea trabajar, posiciones de los tabuladores de un editor, parámetros de un programa de comunicaciones, modelo de memoria a utilizar por un compilador en la generación de código, ... ).

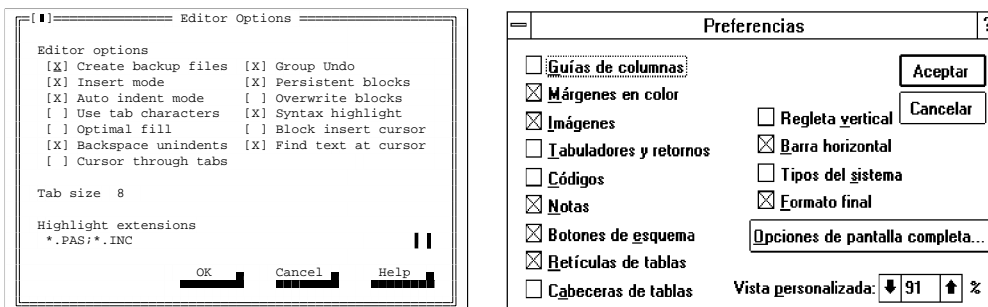


Fig. 14.5. Ejemplos de cajas de diálogo.

## MARCOS DE APLICACION

Las cajas de diálogo normalmente constan de una ventana movable que contiene uno o varios objetos botones (etiquetados, por ejemplo, con **OK** o **Aceptar** y **Cancel** o **Cancelar** para aceptar o rechazar las modificaciones realizadas en los parámetros controlados por la caja de diálogo). Esta ventana puede contener también *casillas de verificación* (*check boxes*) para activar/desactivar opciones (*binary switches*) y *botones de radio* (*radio buttons*) para seleccionar una de varias opciones excluyentes entre sí. Otro tipo de objetos vista que pueden contener las cajas de diálogo son etiquetas de texto o campos de entrada de datos. En la figura 14.5 se pueden observar ejemplos en modo texto y gráfico de este tipo de vistas.

### • Vistas de texto

Las vistas de texto son rectángulos de salida para texto dentro de la pantalla. Se utilizan para presentar grandes estructuras de datos como *buffers* de edición de texto, mensajes de información, campos de entrada de texto en una caja de diálogo, y cajas de selección de lista que permiten escoger una cadena de caracteres dentro de una lista de cadenas presentadas (figura 14.6).

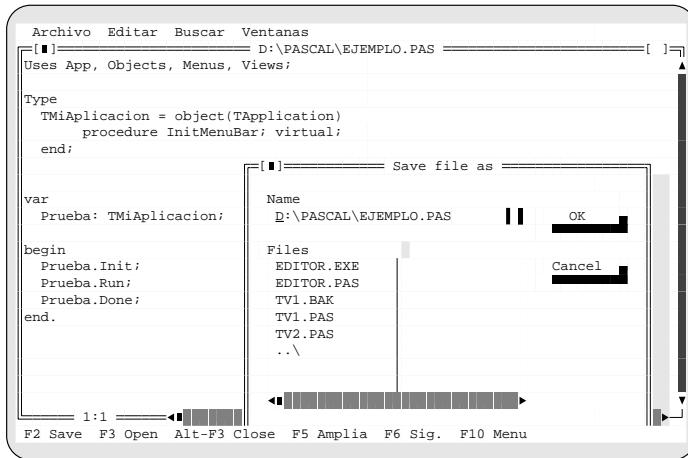


Fig. 14.6. Vista de texto para la edición de un fichero.

### • Vistas gráficas

Las vistas gráficas no son posibles en sistemas de ventanas en modo carácter. Sólo las interfaces gráficas de usuario permiten visualizar gráficos (*p.ej.* iconos gráficos, figuras geométricas, ...), siendo posible mezclar texto y gráficos en la misma ventana como se puede apreciar en la figura 14.7. El texto en este tipo de ventanas se simula mediante una representación gráfica del carácter, lo que permite utilizar distintos tipos de *fuentes de letra* en una misma pantalla. Este es necesario cuando se implementa, por ejemplo, un procesador de textos WYSIWYG en el que la imagen de lo visualizado en pantalla debe ser una copia (de menor calidad) de lo que se va a obtener en una impresora laser.

Las vistas gráficas se construyen utilizando distintos objetos gráficos (plumas, paletas, herramientas de dibujo, ...). La mayoría de los marcos de aplicación incluyen recursos para implementar estos objetos.

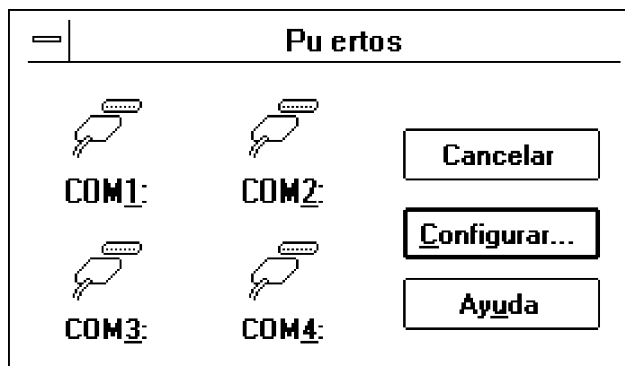


Fig. 14.7. Vista gráfica con texto e imágenes.

#### • Vistas de control

Son un tipo de vistas especiales que responden a los eventos de entrada generados por el usuario de una aplicación (*p.ej.* un botón —vista de control— puede ser seleccionada por el usuario al señalarlo con el ratón y pulsar su botón izquierdo —eventos de entrada). La diferencia entre una vista de control y una vista normal, que sólo presenta información, está en que la primera, cuando es activada, envía mensajes de control al gestor de eventos, véase figura 14.8 (*p.ej.* el botón de **Aceptar** una ventana para introducir datos cuando es seleccionado envía el mensaje de que la ventana ya se puede cerrar y realizar las operaciones oportunas con los datos que contiene).

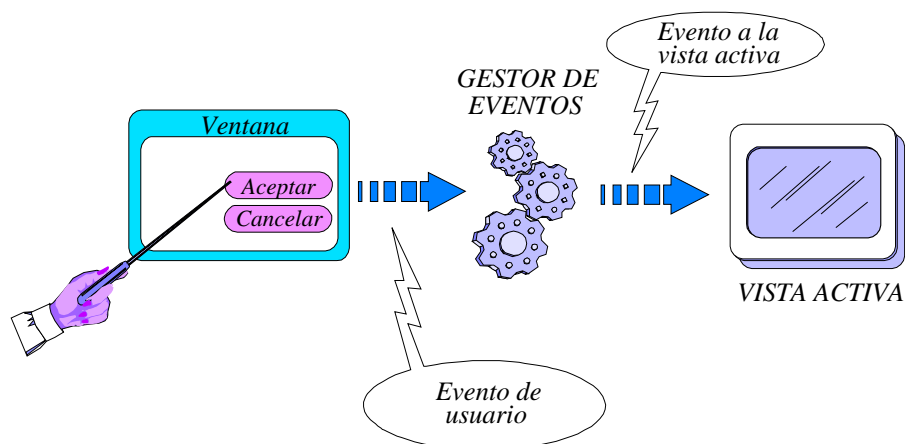


Fig. 14.8 Funcionamiento de una vista de control

## MARCOS DE APLICACION

Las vistas de control tienen las características comunes a los objetos vistas y a los objetos de control. Para que se pueda activar una vista de control debe ser visible en la pantalla. Una ventana de edición de textos o un campo de entrada de datos son vistas de control ya que responden a eventos de entrada generados por el usuario como las pulsaciones de teclado. Cada tecla pulsada causa que un carácter sea presentado en la vista, y normalmente, además, ese carácter se añade a un buffer manejado por un *modelo* determinado. Las vistas de edición deben monitorizar y responder a los eventos de usuario, lo cual las diferencia de una vista que únicamente se ocupa de visualizar texto. Para éstas, las teclas pulsadas no provocan ninguna acción (ignoran los eventos de entrada exceptuando los relacionados con cerrar, mover o redimensionar la vista).

Las vistas de control típicas en los marcos de aplicación son los botones, grupos de botones, barras de desplazamiento, selección en listas y editores de texto. Estos últimos están formados por un conjunto de elementos de control como son campos de entrada de línea, objetos de control para editar conjuntos de datos en memoria (*buffer*) o ficheros de disco, y en algunos casos se incluyen subobjetos que facilitan realizar acciones típicas en un editor como son la búsqueda y sustitución de texto, almacenar y recuperar un fichero, etc.

### Botones y grupos de botones

Existen tres tipos de *botones (buttons)*: botones individuales, casillas de verificación y botones de radio.

- ✘ Los *botones individuales*<sup>37</sup> o *botones de acción* permiten activar directamente comandos o acciones de la aplicación, como por ejemplo los botones: , ,  y  de la figura 14.9.
- ✘ Las *casillas de verificación* permiten activar o desactivar parámetros de valor binario o biestado. Así en la figura 14.9 son casillas de verificación las mostradas en el grupo denominado *Visualizar*.
- ✘ Los *botones de radio*<sup>38</sup> permiten especificar una opción de un conjunto de opciones excluyentes entre sí. Así en la figura 14.9 tienen botones de radio los grupos denominados: *Deshacer*, *Tamaño cursor*, *Replicar*, *Guardar*, *Mostrar bordes página* y *Utensilio de dibujo*.

Los botones se reúnen en *grupos*, a los cuales se les asigna un nombre de grupo. De esta forma se facilita al usuario la tarea de elegir las opciones.

---

<sup>37</sup> Que denominaremos simplemente botones (*pushbutton*).

<sup>38</sup> Se les denomina de esta forma por su semejanza en el funcionamiento a los botones de selección de emisoras en equipo de recepción de radio: sólo puede seleccionarse una emisora en un momento dado, no puede estar más de un botón pulsado simultáneamente.

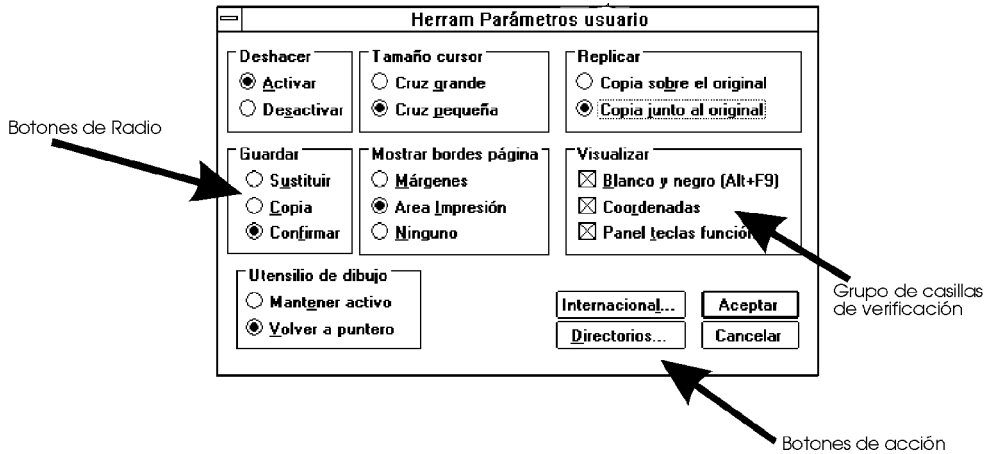


Fig. 14.9 Botones de control

En la figura 14.9 se puede observar una caja de diálogo, para controlar los parámetros de personalización de un entorno de trabajo, en la que se utilizan varios tipos de botones de control.

Al colocar los botones dentro de un *grupo* con un título de grupo de referencia, como en el caso de la velocidad de transmisión o la paridad, permite pasar el control de un grupo a otro mediante la pulsación de las teclas de tabulación *TAB* o *Shift+TAB*. Estas teclas son usadas por el usuario cuando quiere pasar el control al siguiente o anterior campo de la caja de diálogo. Cuando existan muchas casillas de verificación o botones de radio en la caja de diálogo, es aconsejable reunirlos en grupos o *clusters* que faciliten el movimiento entre las vistas de control del diálogo.

### Barras de desplazamiento

Las barras de desplazamiento permiten especificar un porcentaje de un rango de posibles valores (*p.ej.* que parte de la totalidad de un fichero de texto se desea visualizar en la ventana de un editor). Las barras de desplazamiento pueden ser tanto horizontales como verticales. De esta forma, en el caso del editor, las barras verticales nos permiten visualizar el texto que no entra a lo ancho de la ventana, tanto a la izquierda como a la derecha (para lo que normalmente utilizaríamos las teclas **End** **Home** **←** y **→**); de igual forma las barras verticales nos permitirían pasar a paginas anteriores o posteriores del texto (similar a **↑** **↓** **PgUp** y **PgDn**)

Las barras de desplazamiento también se pueden usar para controlar el volumen de un programa que genera sonido, la intensidad del monitor o la sensibilidad del ratón en aquellos programas que permiten al usuario modificar estos parámetros.



## MARCOS DE APLICACION

En la figura 14.10 se puede observar que una barra de desplazamiento consta de una barra en cuyo interior se sitúa un cuadrado desplazable con el ratón o las teclas de movimiento citadas anteriormente, y que representa el valor del parámetro controlado por la barra de desplazamiento (a la izquierda o arriba valor mínimo, a la derecha o abajo valor máximo). A ambos lados de la barra se encuentran dos botones de flecha que permiten modificar el valor del parámetro controlado con el incremento mínimo (*p.ej.* una línea para una ventana de edición).

Las barras verticales se utilizan con frecuencia para recorrer listas de opciones en una caja de selección cuando estas superan una determinada longitud de opciones (*p.ej.* una lista de ficheros que se presenta para seleccionar un fichero a editar).

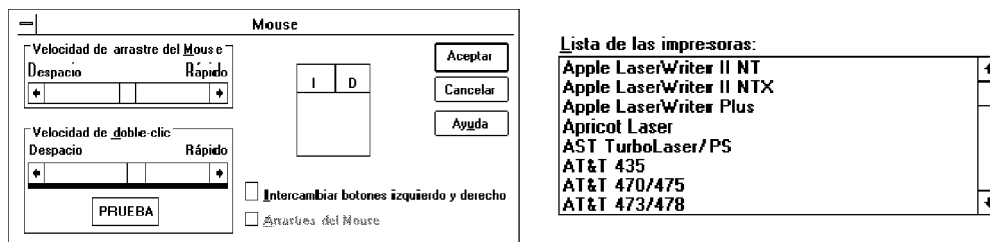


Fig. 14.10 Barras de desplazamiento verticales y horizontales

### Listas de selección

Las listas de selección permiten al usuario escoger una cadena de caracteres (*string*) de una lista de cadenas. Cada cadena está asociada con un objeto, que de esta forma puede ser seleccionado. Así por ejemplo, cada fichero de un directorio está asociado con un nombre o cadena de caracteres; el usuario puede indicar a que fichero desea acceder con sólo apuntar a una de las cadenas, resaltarla y pulsar el botón **Aceptar** (también se podría seleccionar haciendo *doble clic*<sup>39</sup> sobre la cadena).

Cuando la lista de selección tiene más elementos que el número de líneas del cuadro de diálogo que se utiliza para presentarla, aparecerá una barra de desplazamiento vertical (al lado derecho normalmente) que se usará para desplazarse a lo largo de la lista.

En la figura 14.11 se muestran dos listas de selección típicas, una entorno gráfico y otra en modo texto.

---

**39** Hacer *doble clic* consiste en pulsar dos veces seguidas el botón del ratón (normalmente el derecho) de forma mas o menos rápida (según la sensibilidad del ratón) para seleccionar un objeto representado en pantalla (a través de un icono o una cadena de caracteres) y actuar sobre él si tiene alguna acción asociada (como editar un objeto fichero de texto, ejecutar un fichero ejecutable, ...)

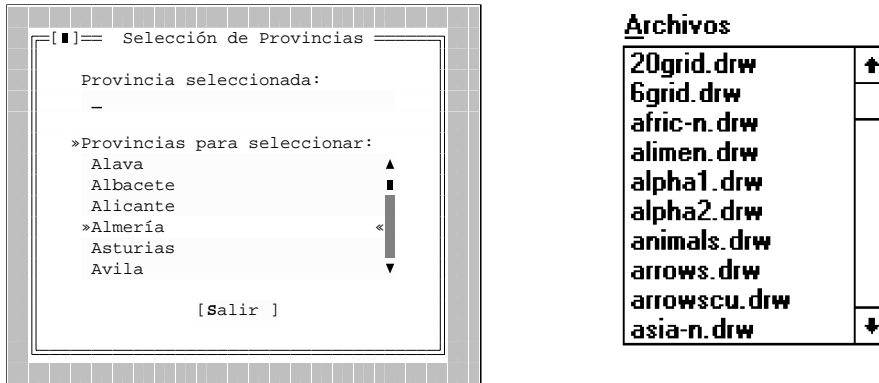


Fig. 14.11. Listas de selección

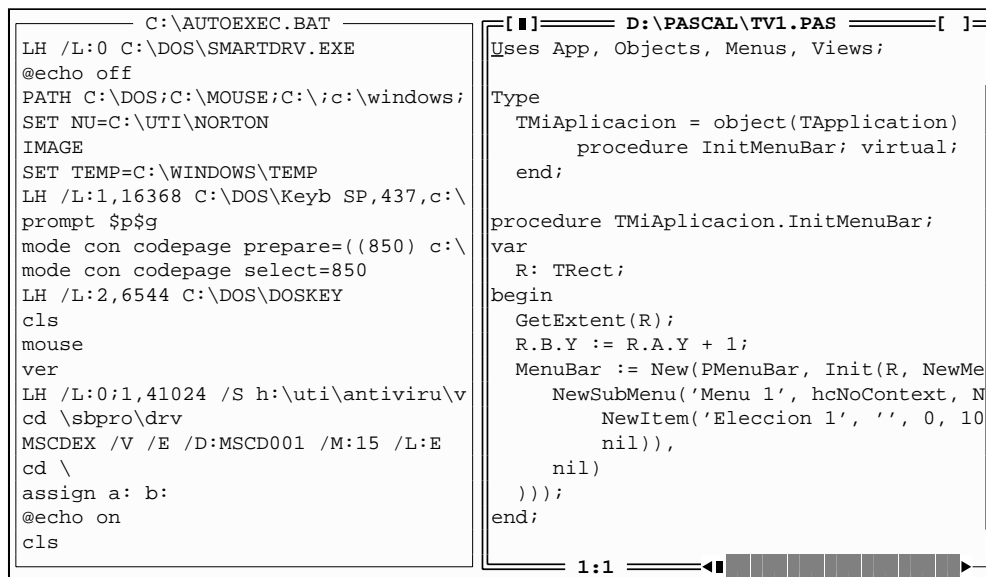


Fig. 14.12 Vista de un editor de texto creada con Turbo Vision

### Vistas de texto y editores de texto

Permiten visualizar y editar texto. Son claros ejemplos de vistas de control que están diseñadas para responder a las entradas de usuario. Normalmente tienen barras de desplazamiento horizontal y vertical, iconos o símbolos para cerrar, mover, ampliar/reducir o redimensionar la vista (figura 14.12). Las vistas de texto tienen métodos para visualizar o leer datos de la vista, trabajando con objetos de tipo *string* que pueden ocupar toda la memoria disponible

## PROGRAMACION DIRIGIDA POR EVENTOS

para mantener la información (texto) de la vista si así se requiere. En algunos casos los objetos string son más sofisticados para permitir manipular una información que requeriría una mayor capacidad de almacenamiento que la proporcionada por la memoria real del ordenador. Este tipo de *string virtual* añade la funcionalidad necesaria para poder almacenar la información en disco o recuperarla de éste, manteniendo en memoria la parte del string necesaria para un acceso inmediato (*p.ej.* páginas o líneas próximas a la página/línea activa en un editor de texto).

Un caso especial de editores de texto son aquellos que permiten editar una línea de entrada de datos (figura 14.13). Estos editores de línea se utilizan frecuentemente en las cajas o cuadros de diálogo como campos de entrada.

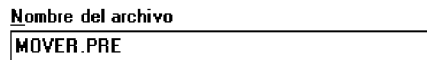


Fig. 14.13 Editor de línea para entrada de datos

### • Vistas de datos

Permiten presentar al usuario los distintos tipos de datos que este puede necesitar en una aplicación típica. Aunque aparentemente no existe ninguna distinción entre este tipo de vistas y una ventana normal, las vistas de datos se usan para visualizar datos exclusivamente y no para manipularlos. Los cuadros de mensaje y de diálogo que no necesitan entradas de usuario, pueden ser considerados vistas de datos (figura 14.14).

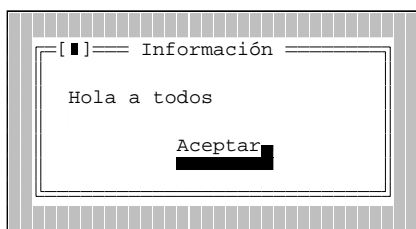


Fig. 14.14 Cuadro de diálogo

Aunque las cajas de diálogo contienen normalmente vistas de control, por sí mismas se consideran vistas de datos ya que, técnicamente, no son los diálogos quienes responden a los eventos generados por el usuario sino las vistas de control que están dentro de los cuadros de diálogo. Aunque la diferencia puede parecer un poco arbitraria, está se verá más claramente cuando se comprenda como un programa manipula los eventos.

## 14.3 PROGRAMACION DIRIGIDA POR EVENTOS

La programación en marcos de aplicación es similar entre todos ellos. EL primer obstáculo que encuentra un programador clásico es que ya no dispone de la semántica del terminal de caracteres. Este presupone que hay un cursor, a partir del cual se escribe el texto que se desee. Sin embargo ahora estamos en un entorno de trabajo, y podemos mostrar texto (o dibujos en entornos gráficos) en cualquier lugar de la superficie de presentación. Como consecuencia además de informar, por ejemplo, de que texto queremos escribir habrá que dar un punto de comienzo (ya no hay cursor, aunque puede haber una posición activa por defecto) y en que ventana se desea escribir.

Por otro lado, no se podrán usar las primitivas de salida clásicas (*write*, *writeln*), sino que habrá que utilizar los métodos que se ofertan en cada marco para generar salidas sobre los distintos tipos de vistas. Sólo de esta forma la presentación será coherente y podremos utilizar la funcionalidad de las vistas (cerrarlas, abrirlas, redibujarlas, ampliarlas, etc) sin perder la información que se haya enviado al usuario.

Sin embargo el mayor problema que encuentra un programador es la *programación por eventos*. Este es un paradigma de programación que supone un cambio radical en la forma de desarrollar software. Hasta ahora la programación era dirigida por el programador. Esta forma de programar presupone el orden en el que el usuario realiza las operaciones. Es una programación básicamente secuencial, mientras no usemos sentencias de control de flujo, de forma que el punto en el que nos encontremos refleja la historia del proceso hasta ese momento y por lo tanto su estado. En la programación por eventos, el programa es dirigido por el usuario. El programador ofrece al usuario un conjunto de acciones a realizar, y este las realiza en el orden que desee o incluso no las realiza. Esto supone que dividimos nuestro programa en un conjunto de pequeños módulos cada uno de los cuales implementa un servicio que se ofrece al usuario. La programación no es lineal, y es más complejo saber el estado del proceso.

La *programación dirigida por eventos*<sup>40</sup> es una consecuencia natural de la organización de un programa según una estructura *modelo-vista-control*. En lugar de tener varias subrutinas del programa responsables de monitorizar los distintos dispositivos que pueden producir potenciales entradas en la aplicación, el mecanismo de control central se encarga de monitorizar todos los dispositivos de entrada. El *mecanismo de control central* o *gestor de eventos* (*event manager*) envía todos los eventos a la *vista principal*, que se encargará de enviarlos a su vez a las subvistas apropiadas. Las subvistas, de acuerdo a un orden preestablecido, podrían enviar los eventos a otras subvistas o consumirlos.

La programación dirigida por eventos y la programación orientada a objetos suelen ir muy ligadas. Si utilizamos la terminología de la programación orientada objetos, podríamos decir que en el espacio de trabajo de un sistema dirigido por eventos residen objetos que reciben *eventos*, o *mensajes*, y la respuesta a estos puede implicar el envío de otros eventos a otros objetos de este espacio.

Los eventos podríamos agruparlos en las siguientes categorías:

- Eventos externos (producidos directamente por el usuario)
  - Eventos de teclado
  - Eventos del ratón
- Eventos internos (producidos por el sistema o la aplicación)
  - Notificaciones crónicas (*p.ej.* envío de un carácter a un puerto serie)

---

<sup>40</sup> *Construcción de aplicaciones basadas en un esquema de control dirigido por eventos.*

## PROGRAMACION DIRIGIDA POR EVENTOS

- Mensajes de menú
- Notificaciones de controles
- Mensajes producidos por otros mensajes
- Mensajes producidos por el programador

La filosofía de un sistema basado en eventos podría resumirse con la siguiente frase:

*Los objetos no intentan recibir datos del exterior (lo que implicaría un ciclo de lectura por objeto, durante el cual ningún otro objeto podría interactuar con el usuario), sino que permanecen en el espacio de trabajo, esperando que se produzca una situación en la cual sea requerida su intervención.*

Cada objeto en este tipo de sistema pertenece a una *clase*. La clase es un mecanismo para especificar las características, atributos principales y conducta de los objetos que a ella pertenecen

## CAPTURA DE EVENTOS

Cada vez que se pulsa una tecla, llega un carácter por un puerto serie, se mueve el apuntador del ratón o se presiona un botón del ratón, el mecanismo de control registra el evento producido y lo coloca en una cola de eventos FIFO<sup>41</sup>. El control del programa a un nivel superior se implementa como un bucle que toma el siguiente evento de la cola y lo envía a la vista principal.

## BUCLE PRINCIPAL DE PROCESAMIENTO DE EVENTOS

Todos los eventos en un programa basado en una estructura *modelo-vista-control* pasan por el *bucle principal de eventos (main event loop)*, también llamado *control de nivel superior o principal (top-level controller)*, que mantiene una conexión directa con la *vista principal (top-level view)*. El bucle de procesamiento de eventos es el responsable de leer los eventos enviados a la cola FIFO por el gestor de eventos. El pseudocódigo para el bucle principal tiene una estructura muy sencilla:

```
Hacer siempre
 e = LeerSiguienteEvento
 EncaminarEvento(e)
```

El código que implementa este bucle es invisible para el programador, a menos que se implemente su propio marco de aplicación. Este código estará dentro de la librería de clases del marco de aplicación y es de poco interés para entender como funcionan los eventos. Es más importante conocer como se activa el mecanismo de lectura/gestión de eventos. Esto habitualmente

---

**41 First-In First-Out.** Estructura de datos (lista) para almacenamiento de elementos en la cual el primer elemento en llegar es el primero en salir.

se realiza con un sencilla llamada, pero antes deben definirse ciertas clases, crearse algunos objetos y efectuar algunas inicializaciones. Así por ejemplo en *Turbo Vision*<sup>®</sup> una clase aplicación se puede crear derivando una nueva clase de la clase *TAplicacion* ya existente:

**Ejemplo 14.1**

```

type
 TMiAplicacion = Object(TAplicacion)
end;

```

que llamamos *TMiAplicacion*. Después habrá que crear una instancia de esta clase (será una vista) que llamaremos *MiApl*:

```

var
 MiApl: TMiAplicacion;

```

y finalmente el cuerpo del programa principal será el siguiente, con sólo tres sentencias:

```

begin
 MiApl.Init;
 MiApl.Run;
 MiApl.Done;
end.

```

La primera sentencia realiza la inicialización necesaria de los objetos de la aplicación. La segunda activa el mecanismo de lectura/gestión de eventos. Este mecanismo permanece activo hasta que el usuario indique que el programa debe finalizar (*p.ej.* seleccionando un comando de salida en un menú o en la barra de estado de la aplicación). Entonces se ejecuta la tercera y última sentencia (*Done*) que se encargará de realizar una limpieza y liberar los recursos utilizados por la aplicación.

Realizar un programa en otro marco de aplicación como *C++ Views* es igual de sencillo como se puede ver en la siguiente función *main*<sup>42</sup> de una aplicación típica escrita en C++:

```

main()
{
 MiVistaApl *v;

 v = new MiVistaApl();
 v->show();
 notifier.start();
}

```

El bucle principal de eventos se encapsula dentro de la clase *Notifier*. Una instancia global de esta clase, llamada *notifier*<sup>43</sup>, se genera automáticamente para cada aplicación. Lo único que debe hacer el programador es crear una vista para la aplicación (`v = new MiVistaApl();`) y presentarla en pantalla (`v->show();`) ya que no se visualiza automáticamente al crearla. Por último sólo resta por activar el bucle principal (`notifier.start();`).

---

<sup>42</sup> Similar al cuerpo principal de un programa en Pascal.

<sup>43</sup> En C++ se distingue entre letras mayúsculas y minúsculas.

## PROGRAMACION DIRIGIDA POR EVENTOS

Normalmente las aplicaciones son más complicadas que lo se ha visto. El programador debe crear un modelo y hacerlo accesible desde la vista principal, y encargarse de liberar el objeto del modelo cuando la aplicación finalice.

### ENVIO DE EVENTOS

El envío de eventos al nivel superior (vista principal) se realiza automáticamente. Si sólo la vista principal necesita responder a los eventos producidos, el programador puede despreocuparse totalmente del manejo de los eventos. En cambio, si crea subvistas o añade eventos de control a las vistas para realizar algún proceso determinado, los manejadores de eventos en la vista principal se pueden programar para que sean enviados a las subvistas o a las vistas padres<sup>44</sup>.

### MANEJO DE EVENTOS

En los ejemplos anteriores no se han utilizado manejadores de eventos. Existen manejadores genéricos proporcionados por los marcos de aplicación de forma que la plantilla o esqueleto inicial de los programas pueda ser creado y probado con muy poco esfuerzo. Por supuesto, este prototipo inicial no hará nada nuevo más allá de presentar la apariencia visual inicial del programa y permitir ejecutar comandos de finalización. Para realizar tareas específicas se deben definir manejadores de eventos.

### UN EJEMPLO

Para finalizar este apartado sobre la programación dirigida por eventos ilustraremos lo explicado con un ejemplo que muestra un sencillo cuadro de información. Este ejemplo lo implementaremos utilizando dos marcos de aplicación: *Turbo Vision* y *Object Windows*, permitiendo ver un entorno de texto en un caso y gráfico en el otro.

Dado que los marcos de aplicación permiten trabajar con múltiples ventanas de tamaños variables simultáneamente, y estas pueden moverse o cambiar de tamaño, el programador no puede escribir texto en cualquier lugar de la pantalla de forma similar a como se hace con la función *writeln*. Primero se debe abrir una ventana y después escribir texto de una manera específica. Cada ventana debe tener asociado un método que le permita escribir el texto que se desea que presente cada vez que sea necesario actualizar los contenidos de la ventana (*p.ej.* cuando una ventana está oculta, parcial o totalmente por otra, y esta se cierra o se mueve). A este suceso se le denomina *evento de exposición (exposure event)*. Este tipo de eventos envían automáticamente mensajes a todas las ventanas del desktop, indicando la región que ha sido expuesta. Cada ventana puede ver entonces si necesita redibujar alguna parte de sí misma, y si es así llamará al método que realice esta operación (*draw* en *Turbo Vision* o *show* en *C++ Views*).

---

<sup>44</sup> Vista padre de una subvista es aquella vista dentro de la cual se ha insertado la subvista.

El ejemplo que se implementa presenta un cuadro de informe (*report dialog*) que permite visualizar un mensaje. Un cuadro de informe abre una vista, presenta un texto que se le pasa como argumento y espera a que el usuario active el botón **O.K.** o **Aceptar**, o se cierre la ventana por medio del icono de control en la esquina superior izquierda.

El programa del ejemplo 14.2 utiliza la librería de clases de Turbo Visión, su ejecución aparece en las figuras 14.15 y 14.16.

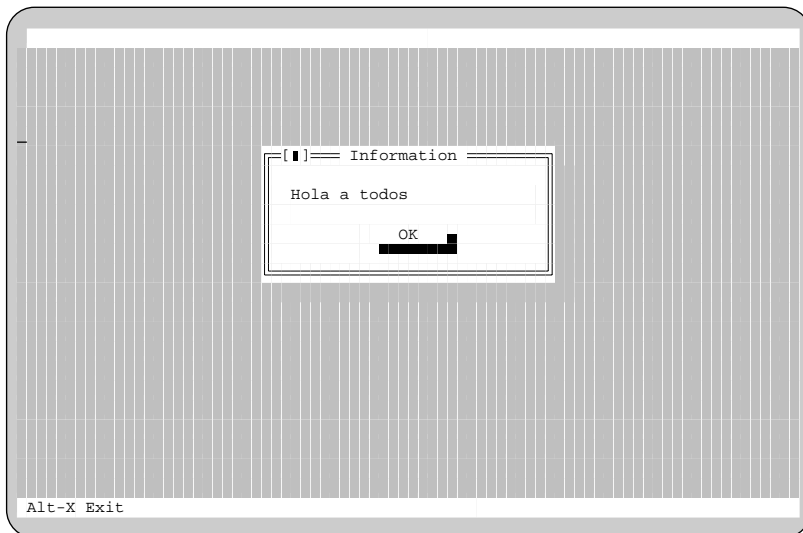


Fig. 14.15 Caja de diálogo creada con Turbo Vision

### Ejemplo 14.2.

```

Program Saludol(Output);

Uses App, Objects, MsgBox;

type
 TMiAplicacion = Object(TApplication)
 end;

procedure CuadroInforme(S: string);
var
 R: TRect;
 C: Word;
begin
 R.Assign(25, 5, 55, 12);
 C := MessageBoxRect(R, S, NIL, mfInformation + mfOkButton);
end;

var
 Hola: TMiAplicacion;

```



## PROGRAMACION DIRIGIDA POR EVENTOS

```
begin
 Hola.Init;
 CuadroInforme(' ; Hola a todos ! ');
 Hola.Run;
 Hola.Done;
end.
```

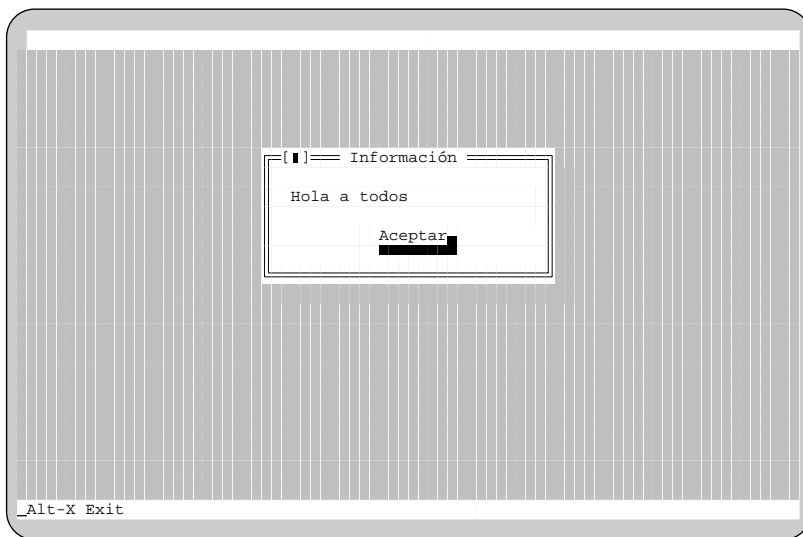


Fig. 14.16 Caja de diálogo con títulos en castellano

La única diferencia de este programa con la versión del ejemplo 14.1<sup>45</sup> está en la definición y llamada al procedimiento `CuadroDialogo`. Este procedimiento, que tiene una cadena como argumento, crea un objeto rectángulo<sup>46</sup> y le asigna sus dimensiones dentro del desktop. Después se llama al procedimiento `MessageBoxRect`<sup>47</sup> pasando como primer parámetro el rectángulo y segundo la cadena. El tercer argumento, `NIL`, es un puntero a una lista de parámetros. La variable string `S`, que se pasa a `MessageBoxRect`, es una cadena de formato muy similar al primer argumento pasado a la función de librería de C `printf`, que junto con el tercer argumento permiten generar mensajes de salida con unas potentes posibilidades de formato. El cuarto argumento contiene varios campos de bits que permiten especificar opciones de la caja de diálogo. Los campos de bits se añaden entre si por medio del operador `+`. Los argumentos de campos de bits especifican el tipo de ventana (`mfinformation`) y los botones de vista de control que se deben añadir a la ventana

---

<sup>45</sup> Todas las aplicaciones escritas con Turbo Vision deben incluir la Unit `App`, donde está definido la clase `TApplication` entre otras.

<sup>46</sup> Para usar objetos rectángulos definidos por `TRect` se debe incluir la Unit `Objects`.

<sup>47</sup> La función `MessageBoxRect` está definida en la Unit de Turbo Pascal `MsgBox` (implementada en el fichero `MSGBOX.PAS`). Si se modifica este fichero y se recompila la TPU se puede conseguir que los títulos del cuadro y los botones de control aparezcan en castellano como se muestra en la figura 14.15.

(*mfOkButton*, *mfCancelButton*, *mfYesButton*, *mfNoButton*). Se pueden incluir uno, dos, tres o los cuatro en un cuadro de dialogo con tan sólo añadir los botones deseados. La caja de diálogo se redimensionará para adaptarse a todos los botones.

### Ejemplo 14.3.

A continuación se presenta un programa escrito utilizando la librería de clases de Object Windows, para obtener un resultado similar al del ejemplo anterior pero bajo el entorno gráfico de Microsoft Windows. El resultado de su ejecución se presenta en la figura 14.17.

```

Program Saludo2(Output);

Uses OWindows, WinProcs, WinTypes;

type
 PVentanaSaludo=^TPVentanaSaludo;
 TPVentanaSaludo= Object (TWindow)
 procedure WMLButtonDown(var Msg:TMessage);
 virtual wm_First+wm_LButtonDown;
 end;

 TMiAplicacion=Object(TApplication)
 procedure InitMainWindow; virtual;
 end;

procedure TPVentanaSaludo.WMLButtonDown;
begin
 MessageBox(HWindow, '¡Hola a todos!', 'Saludo', mb_OK);
end;

procedure TMiAplicacion.InitMainWindow;
begin
 MainWindow:=New(PVentanaSaludo, Init(NIL,'Ejemplo de mensaje'));
end;

var
 MiAplica:TMiAplicacion;

begin
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
end.

```

En el capítulo 15 se explicará la programación bajo el entorno Windows utilizando la librerías de clases de Object Windows. Por ahora baste decir, como se puede apreciar comparando los dos ejemplos anteriores, que ambos marcos de aplicación de Borland poseen la misma jerarquía de clases diferenciándose su uso por las restricciones que impone los entornos en que se ejecutan las aplicaciones de los dos marcos (DOS en modo texto y Windows en modo gráfico).

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

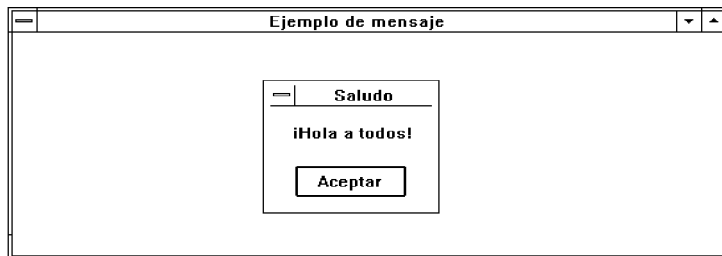


Fig. 14.17 Caja de diálogo bajo entorno Windows

### 14.4 TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

*Turbo Vision* es un marco de aplicaciones orientado a objetos para programas con una interface de usuario por medio de ventanas en modo texto. Su librería de clases incluye soporte para:

- Ventanas múltiples, redimensionables y solapables.
- Menús desplegados
- Cuadros de diálogo
- Botones, barras de desplazamiento, campos de entrada, casillas de verificación y botones de radio
- Tratamiento estandarizado de pulsaciones de teclas y clicks de ratón.
- Validación de datos
- Personalización de colores de los elementos
- Manejo de ratón

Las librerías de clases se organizan en una jerarquía y no como un conjunto de herramientas sin conexión entre ellas. La programación con Turbo Vision implica realizar una programación dirigida por eventos, permitiendo escribir programas flexibles que proporcionan al usuario control sobre la parte del programa que quieren acceder, sin que esto venga impuesto por el programa. En

definitiva, Turbo Vision proporciona los cimientos sobre los que desarrollar las aplicaciones, adoptando un enfoque estandarizado y racional del diseño de pantallas con lo que todas las aplicaciones desarrollados con este marco de aplicación adquieren un aspecto familiar, idéntico al de los propios entornos de los lenguajes *Turbo*.

En la mayor parte de la exposición utilizaremos como ejemplos partes del programa AGENDA que se encuentra en el directorio de ejercicios del capítulo catorce que lleva su mismo nombre. Este programa es un buen ejemplos de uso de Turbo Vision al utilizar gran parte de las posibilidades de su librería de clases. El programa permite editar ficheros de texto al incorporar opciones de edición; configurar el programa a través de las opciones de colores, ratón, modo de video, ...; manipular las ventanas; utilizar una calculadora y un calendario y mantener una sencilla agenda. Y todo ello mediante menús, teclas aceleradoras, línea de estado, ayuda en línea y posibilidad de uso del ratón.

La ventaja de usar el marco de aplicación es que se puede desarrollar rápidamente un prototipo del programa para comprobar su apariencia y adecuación de los menús diseñados a su futura funcionalidad, sin necesidad de implementar el modelo de la aplicación.

Cuando se construyen programas con Turbo Vision se crean muchos tipos de vistas. Las más comunes son los menús y los cuadros de diálogos. Los menús son vistas muy especiales puesto que a la vez son controles: reciben una entrada de usuario y la traducen en un evento que deberá ser manejado. Son controles visibles (aunque los controles normalmente no se ven) llamadas *vistas de control*, necesarios para que el usuario pueda indicar las acciones que desea realizar. Otro tipo de controles visibles son los botones (de radio, de verificación, ...). Todos ellos contrastan con el gestor de eventos que no es visible en ningún lugar de la pantalla. Las vistas de control deben ser siempre los primeros componentes que se deben construir cuando se desarrollan aplicaciones utilizando un marco de aplicación.

Si no se utiliza un marco de aplicaciones, el mayor esfuerzo normalmente se pierde en el desarrollo y mantenimiento del interface de usuario. Turbo Vision simplifica esta tarea. En el ejemplo 14.1 vimos el programa más simple que se puede desarrollar utilizando Turbo Vision:

```

program plantilla_TV;

uses App;

type
 TMiAplicacion = Object(TApplication)
end;

var
 MiApl: TMiAplicacion;

begin
 MiApl.Init;
 MiApl.Run;
 MiApl.Done;
end.

```

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

Se parte de una clase heredada de la clase *TApplication* de la *Unit App*, si añadir ninguna funcionalidad adicional, se instancia el objeto `MiAp1` de esta nueva clase y utiliza en el cuerpo principal del programa en la siguiente secuencia de llamadas a los métodos del objeto:

- llamada a su constructor `Init` (inicializa los objetos de la aplicación)
- al método `Run` (activa el mecanismo de lectura/gestión de eventos que permanecerá activo hasta que el usuario indique que el programa debe finalizar)
- llamada al destructor de la clase `Done` (realiza una limpieza y libera los recursos utilizados por la aplicación)

Esta estructura elemental se mantiene constante en todas las aplicaciones desarrolladas con Turbo Vision. Para poder modificar el comportamiento de la aplicación<sup>48</sup> no hay que modificar los fuentes de Turbo Vision, ni fuentes de clases desarrolladas anteriormente por el programador. Lo que se hace es extender la funcionalidad de los objetos para cada aplicación. El esqueleto de aplicación *TApplication* se mantiene invariable dentro de `APP.TPU`. Se le añaden nuevos comportamientos derivando nuevos tipos objeto que modifican lo que sea necesario redefiniendo los métodos heredados (en el caso anterior no se añade nada a `TMiAplicacion`) con métodos nuevos que se definen para los nuevos objetos. A continuación se puede observar el objeto aplicación `TGestionFichas` que se define para el programa AGENDA:

```
TGestionFichas = object(TApplication)
 VentanaClipboard: PEditWindow;
 VentanaFicha: PVentanaFicha;
 Reloj: PVistaReloj;
 Heap: PVistaDeHeap;
 constructor Init;
 destructor Done; virtual;
 procedure CancelarFicha;
 procedure CajaAbout;
 procedure MeterNuevaFicha;
 procedure GetEvent(var Evento: TEvent); virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Evento: TEvent); virtual;
 procedure Idle; virtual;
 procedure InitMenuBar; virtual;
 procedure InitStatusLine; virtual;
 procedure NuevaVentana;
 procedure AbrirVentanaFicha;
 procedure AbrirVentana;
 procedure CargaFichas;
 procedure SalvaFichas;
 procedure CargarDesktop;
 procedure SalvarDesktop;
 procedure SalvarDatosFicha;
 procedure MostrarFicha(ANumFicha: Integer);
 procedure InitDesktop; virtual;
```

---

<sup>48</sup> El programa *plantilla\_TV* únicamente crea y presenta el *Desktop*, que incluye una barra de menús, sin elementos todavía, y una barra o línea de estado en la línea inferior. Lo único que se puede hacer con este programa es salir de él seleccionando el icono de control `Alt-X` de la línea de estado y haciendo click con el ratón o pulsando esa combinación de teclas.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
 procedure OutOfMemory; virtual;
 procedure WriteShellMsg; virtual;
end;
```

Los métodos virtuales son métodos heredados de *TApplication* a los cuales se les modificará el comportamiento, en muchos casos extendiendo el que ya tenían. El resto son métodos propios de la nueva clase de aplicación. Para poder implementar los nuevos métodos se hará uso de objetos propios de la aplicación:

```
PVentanaFicha = ^TVentanaFicha;
TVentanaFicha = object(TDialog)
 ContadorFichas: PVistaContador;
 constructor Init;
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure HandleEvent(var Evento: TEvent); virtual;
 procedure Store(var S: TStream); virtual;
end;

PObjetoFicha = ^TObjetoFicha;
TObjetoFicha = object(TObject)
 RegisTransferencia: TFicha;
 constructor Load(var S: TStream);
 procedure Store(var S: TStream);
end;

PDesktopConFondo = ^TDesktopConFondo;
TDesktopConFondo = object(TDesktop)
 procedure InitBackground; virtual;
end;
```

que serán invocados desde los métodos de la clase *TGestionFichas*. A los nuevos métodos se accede al incorporar nuevos comandos al programa, de forma que para poder responder a ellos el gestor de eventos de la aplicación invocará a estos nuevos métodos.

## ELEMENTOS DE UNA APLICACION TURBO VISION

Una aplicación Turbo Vision es una sociedad de *vistas*, *eventos*, y *motores* que cooperan entre sí.

- Una **vista** es cualquier elemento del programa que es visible en la pantalla —tales elementos son objetos. En el contexto de Turbo Vision, todo lo que se puede ver es una vista. Campos, títulos de campo, bordes de ventana, barras de desplazamiento, barras de menús, y botones son vistas. Las vistas pueden combinarse para formar elementos más complejos como ventanas y cuadros de diálogo. Estas vistas colectivas se llaman *grupos*, y operan juntos como si fuesen una sola vista. Conceptualmente, los grupos pueden considerarse vistas.

Las vistas siempre son rectangulares. Esto incluye rectángulos que contienen un único carácter, o una fila que tiene un solo carácter de altura o una columna de un solo carácter de anchura.

- Un **evento** es una especie de suceso al que la aplicación debe responder. Los eventos vienen del teclado, del ratón, o de otras partes de Turbo Vision. Por ejemplo, una pulsación de tecla es un evento, como lo es un click de un botón del ratón. Los eventos son puestos en colas por el esqueleto de aplicación de Turbo Vision (bucle principal de procesamiento de eventos) a medida que ocurren, después son procesados en orden por un manejador de eventos. El objeto *TApplication*, que constituye la base de la aplicación, contiene un manejador de eventos. Mediante un mecanismo que será explicado más adelante, los eventos que no son atendidos por *TApplication* se pasan a otras vistas del programa hasta que o bien se encuentra una vista que gestione el evento, o bien ocurre un error *evento abandonado*.

Por ejemplo, pulsando **F1** se invoca al sistema de ayuda. A menos que cada vista tenga su propia entrada al sistema de ayuda (como podría ocurrir en un sistema de ayuda sensible al contexto como el del programa AGENDA), la pulsación de **F1** es tratada por el manejador de eventos del programa principal. Por contra, las teclas alfanuméricas normales o las teclas de edición, necesitan ser tratadas por la vista que en ese momento tenga el foco<sup>49</sup>.

- Los **motores**, algunas veces llamados *objetos mudos*, son cualesquiera otros objetos del programa que no son vistas. Son *mudos* porque no pueden interactuar con la pantalla directamente. Efectúan cálculos, se comunican con los periféricos, y en general hacen el trabajo de la aplicación. Cuando un motor necesita mostrar algo en la pantalla, tiene que hacerlo con la cooperación de una vista. El motor de la terminología de Turbo Vision son el equivalente de los *modelos* en el paradigma Modelo/Vista/Control.

Este concepto es muy importante para el mantenimiento del orden en una aplicación Turbo Vision: *Sólo las vistas pueden acceder a la pantalla*.

Nada impedirá a los motores escribir en la pantalla con las instrucciones del Pascal `write` o `writeln`. Sin embargo, si escriben en la pantalla *por su cuenta*, se desbaratará el texto que Turbo Vision escribe, y éste borrará el texto *intruso* (por ejemplo, al mover o redimensionar ventanas sobre la posición de ese texto).

En la figura 14.18 se muestra un conjunto de objetos comunes que podrían aparecer como parte de una aplicación Turbo Vision. El *escritorio* o *desktop* es el fondo sombreado sobre el que aparece el resto de la aplicación. Como todo en Turbo Vision, el escritorio es un objeto. También están la *barra de menús* en la parte superior de

---

<sup>49</sup> Se dice que una vista tiene el **foco** cuando esa es la vista que está interactuando con el usuario en un momento dado.

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

la pantalla y la *línea de estado* en la parte inferior. Las palabras en la barra de menús representan menús, que son desplegados al pinchar en las opciones del menú o presionar las *hot keys* o *teclas aceleradoras*.

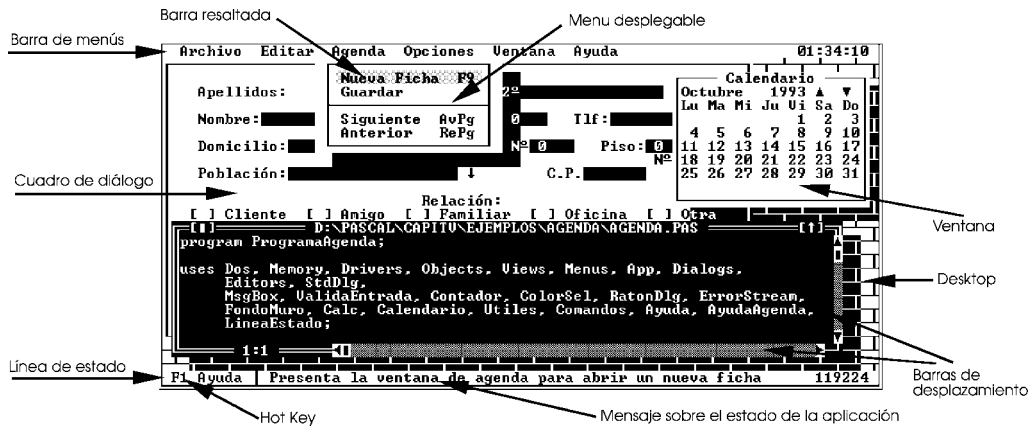


Fig. 14.18 Objetos Turbo Vision

El texto que aparece en la línea de estado lo decide el programador, pero habitualmente muestra mensajes sobre el estado actual de la aplicación, hot keys disponibles, o comandos actualmente disponibles para el usuario.

Cuando se despliega un menú, una barra resaltada se desliza arriba y abajo por la lista de *elementos de menú* (*menu items*) en respuesta a los movimientos del cursor o a las teclas de cursor. Cuando se pulsa  $\leftarrow$  o se presiona el botón izquierdo del ratón, el elemento resaltado en ese momento es seleccionado. La selección de un elemento del menú emite un comando a alguna parte de la aplicación.

Una aplicación habitualmente se comunica con el usuario a través de una o más *ventanas* o *cuadros de diálogo*, los cuales aparecen y desaparecen en el escritorio en respuesta a comandos de ratón o de teclado. Turbo Vision proporciona un gran surtido de herramientas de ventana para presentar e introducir información. Se pueden hacer *desplazamientos* (*scroll*) en el interior de una ventana, lo cual permite a las ventanas mayores presentaciones de datos (e.j. documentos) del que cabría en su tamaño real. El *scroll* a través de los datos en una ventana se hace moviendo una *barra de desplazamiento* en la parte inferior de la ventana, la parte derecha, o ambas. La barra de desplazamiento indica la posición relativa en la ventana respecto a la totalidad de los datos que se muestran.

**JERARQUIA DE TIPOS OBJETO DE TURBO VISION**

El árbol jerárquico se muestra en las figuras 14.19 (en esta figura no aparecen los objetos vistas) y 14.20 (jerarquía de los objetos vistas). Es importante conocer esta jerarquía para poder



## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

utilizarla convenientemente. El saber que, por ejemplo, *TMenuBar* tiene la cadena de ancestros o de derivación *TmenuView - TView - Tobject*, o que la de *TDialog* es *Twindow - TGroup - TView - Tobject* reduce considerablemente la curva de aprendizaje. A medida que se desarrollen aplicaciones propias con Turbo Vision, se verá que una familiaridad general con los tipos objeto estándar y sus mutuas relaciones es una enorme ayuda. Cada nuevo tipo objeto derivado que se encuentre ya tiene propiedades heredadas que son familiares. Simplemente se estudiarán los campos adicionales y las propiedades que se superpongan a las de su padre.

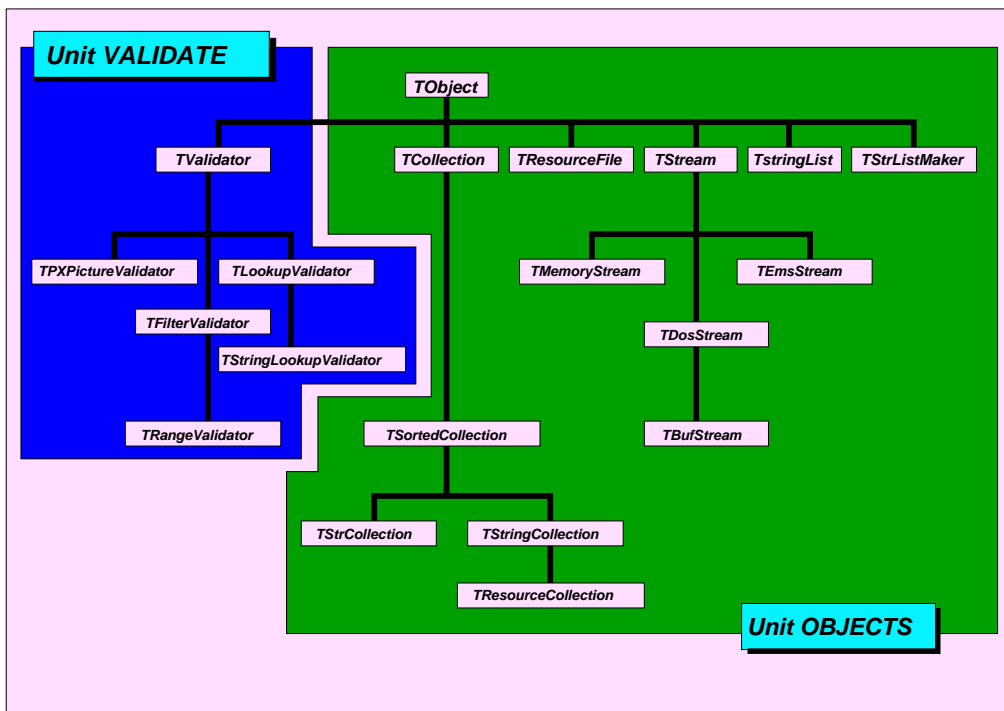


Fig. 14.19 Jerarquía de tipos objeto de Turbo Vision (I)

Algunos de los objetos de la jerarquía se pueden utilizar directamente —se pueden crear instancias de ellos y usarlas. Otros son objetos abstractos —no se pueden crear instancias de ellos —y sirven como base para objetos derivados de los que ya se pueden crear instancias.

La razón de tener tipos abstractos es en parte conceptual pero sirve al propósito práctico de reducir el esfuerzo de codificación. En general, a medida que se baja en la jerarquía, los tipos se vuelven más especializados y menos abstractos. Por ejemplo, los tipos *TRadioButtons* y *TCheckBoxes* podrían derivarse directamente de *TView* sin dificultad. Sin embargo, tienen muchas cosas en común. Ambos representan conjuntos de controles con respuestas similares. Un conjunto de botones de radio es algo muy parecido a un conjunto de casillas de verificación en las cuales

sólo se puede marcar una casilla, aunque hay algunas diferencias. Esta semejanza justifica la creación de un tipo objeto abstracto denominado *TCluster*. Los *TRadioButtons* y *TCheckBoxes* se derivan luego de *TCluster* con la adición de unos pocos métodos especializados para proporcionar sus funcionalidades individuales.

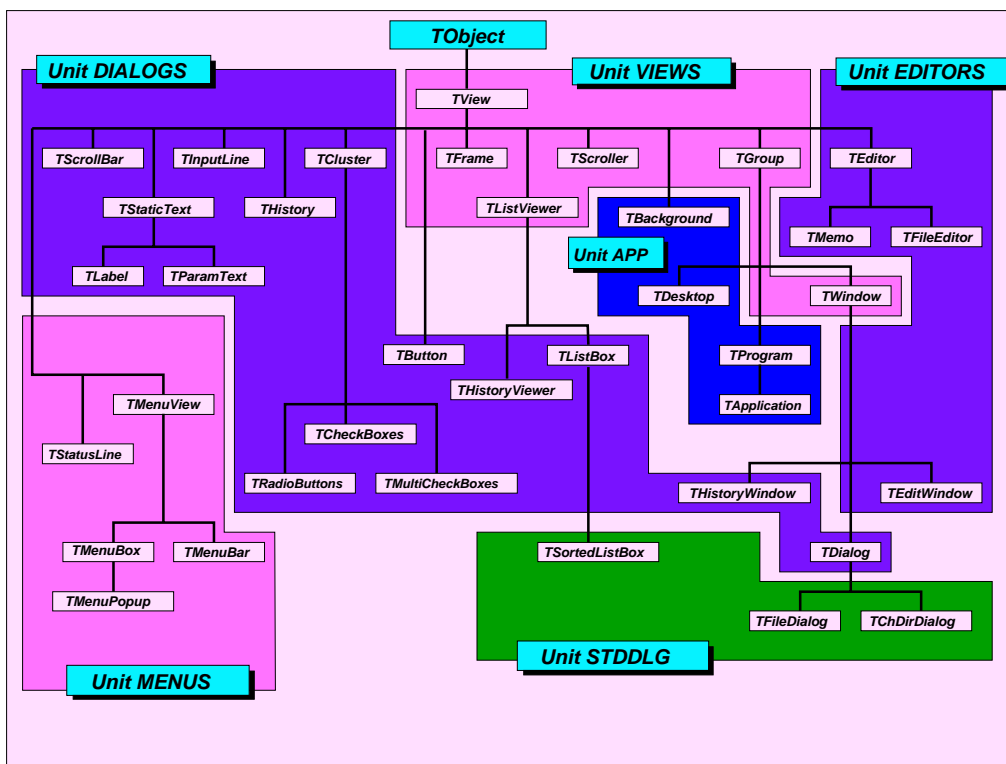


Fig. 14.20 Jerarquía de tipos objeto de Turbo Vision (II)

Nunca es útil, y a menudo es imposible, crear una instancia de un tipo objeto abstracto. Una instancia de *TCluster*, por ejemplo, no tendría un método `Draw` útil, ya que hereda *TView.Draw* sin redefiniciones y este método simplemente visualizaría un rectángulo vacío del color por defecto.

Si se quiere un cluster de controles elegante con diferentes propiedades para los botones de radio y las casillas de verificación, se podría intentar derivar un *TMiCluster* de *TCluster*, o podría ser más fácil derivar el cluster especial de *TRadioButtons* o *TCheckBoxes*, dependiendo de cuál sea más cercano a las necesidades. En cualquier caso, se añaden campos, y se añaden o redefinen métodos, con el mínimo esfuerzo posible. Si los planes incluyen una familia completa de elegantes clusters, podría ser conveniente crear un tipo objeto abstracto intermedio.

Dado un tipo objeto cualquiera se pueden realizar dos cosas básicas:

- *Derivar un tipo objeto descendiente*

Cuando se quiere extender o modificar un tipo objeto existente, se deriva un nuevo tipo objeto del existente:

```
PVentanaFicha = ^TVentanaFicha; { define puntero a nuevotipo }
TVentanaFicha = object(TDialog) { deriva tipo de uno existente }
ContadorFichas: PVistaContador; { añade nuevo campo }
constructor Init;
constructor Load(var S: TStream);
destructor Done; virtual; { redefine método existente }
procedure HandleEvent(var Evento: TEvent); virtual;
procedure Store(var S: TStream); virtual;
end;

TGestionFichas = object(TApplication) { deriva tipo }
. . .
constructor Init;
destructor Done; virtual;
procedure CancelarFicha; { Define nuevo método }
procedure CajaAbout;
procedure MeterNuevaFicha;
. . .
```

Al definir el nuevo objeto, se pueden hacer tres cosas:

- Añadir nuevos campos
- Definir nuevos métodos
- Redefinir métodos existentes

Si no se hace al menos una de esas cosas, no hay ninguna razón para crear un nuevo tipo objeto. Los métodos y campos nuevos o revisados que se definen añaden funcionalidad a *TDialog* y *TApplication*. Los tipos nuevos de objetos casi siempre redefinen el constructor *Init* para determinar los valores y propiedades por defecto.

- *Crear una instancia de ese tipo*

La creación de una instancia de un objeto generalmente es llevada a cabo por una declaración de variable, bien sea estática o dinámica:

```
var
FicheroRecursos: TResourceFile; { declara instancia estática }
FichaInfo: TFicha;
Agenda: TGestionFicha;
ColecFichas: PCollection; { declara instancia dinámica }
FichaTemporal: PObjetoFicha;
```

Agenda será inicializado por *TGestionFicha.Init* quien inicializa los campos extendidos de *TGestionFicha* respecto a su ancestro *TApplication* y invoca a *TApplication.Init* para inicializar la parte heredada con ciertos valores de campo por defecto. Se pueden ver estos consultando la *guía de programación de Turbo Vision*. Dado que *TApplication* es un descendiente de *TProgram*, *TApplication.Init* invoca a *TProgram.Init* para dar valores a los campos heredados de *TProgram*. De forma similar, *TProgram* es un descendiente de *TGroup*, por lo que *TProgram.Init* invocará

a *TGroup.Init*. *Tgroup* es descendiente de *TView* que a su vez lo es de *TObject*, luego *TGroup.Init* invocará al constructor de *TView* y este al de *TObject*. *TObject* no tiene padre, así que la cadena de llamadas se detendría en *TObject.Init*.

Los diagramas de herencia al comienzo de cada entrada de tipo objeto en el Capítulo 19 de la *guía de programación de Turbo Vision* muestran qué campos y métodos declara o redefine cada tipo objeto, tachando en los tipos ascendientes los métodos redefinidos.

El que se pueda crear una instancia utilizable de un tipo objeto depende de la clase de métodos virtuales que tenga el objeto. Muchos de los tipos estándar de Turbo Vision tienen métodos abstractos que deben ser definidos en tipos descendientes.

Si se deriva un tipo objeto descendiente, se obtiene un nuevo tipo objeto sobre el cual se pueden aplicar de nuevo las dos operaciones anteriores (derivar y crear instancias).

#### • Campos y métodos

Vamos a analizar ahora como se heredan los campos y que tipos de métodos maneja Turbo Vision para poder utilizar de forma adecuada su jerarquía de tipos objeto.

Si se toma un trío importante de tipos objeto: *TView*, *TGroup* y *TWindow*, una ojeada a sus campos revela el uso de herencia, y además refleja claramente que la funcionalidad crece a medida que se baja en la jerarquía. La tabla 14.1 muestra el diagrama de herencia de estos objetos, con los campos que tiene cada objeto, incluso aquellos que son heredados. Los métodos que aparecen en negrita son redefinidos en los tipos objeto descendiente. Es decir, que si quitamos todos los métodos en negrita tenemos los métodos que permiten manipular la clase **TWindow**.

En la tabla 14.2 se pueden ver los campos que poseen los tipos *TView*, *TGroup* y *TWindow*. *TGroup* hereda todos los campos de *TView* y añade varios más que son propios del funcionamiento del grupo, como por ejemplo punteros a las vistas actual y última del grupo. *TWindow* a su vez hereda todos los campos de *TGroup* y añade aún más que son necesarios para el funcionamiento de la ventana, como por ejemplo el título y número de la ventana.

TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

| <b>TObject</b>       | <b>TView</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | <b>TGroup</b>                                                                                                                                                                                                                                                                                                                                                                                                        | <b>TWindow</b>                                                                                                                                  |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
|                      | Cursor<br>DragMode<br>EventMask<br>GrowMode<br>HelpCtx<br>Next<br>Options<br>Origin<br>Owner<br>Size<br>State                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Buffer<br>Current<br>Last<br>Phase                                                                                                                                                                                                                                                                                                                                                                                   | Flags<br>Frame<br>Number<br>Palette<br>Title<br>ZoomRect                                                                                        |
| Init<br>Done<br>Free | <b>Init</b><br><b>Awaken</b><br>BlockCursor<br><br>CalcBounds<br><b>ChangeBounds</b><br>ClearEvent<br>CommandEnabled<br><b>DataSize</b><br>DisableCommands<br><b>Done</b><br>DragView<br><b>Draw</b><br>DrawView<br><br>EnableCommands<br><b>EndModal</b><br>EventAvail<br><b>Execute</b><br>Exposed<br>Focus<br>GetBounds<br>GetClipRect<br>GetColor<br>GetCommands<br><b>GetData</b><br>GetEvent<br>GetExtent<br><b>GetHelpCtx</b><br><b>GetPalette</b><br>GetPeerViewPtr<br>GetState<br>GrowTo<br><b>HandleEvent</b><br>Hide<br>HideCursor<br>KeyEvent<br><b>Load</b><br>Locate<br><br>MakeFirst<br>MakeGlobal<br>MakeLocal<br>MouseEvent<br>MouseInView<br>MoveTo<br>NextView<br>NormalCursor<br>Prev<br>PrevView<br><br>PutEvent<br>PutInFrontOf<br>PutPeerViewPtr<br>Select<br>SetBounds<br>SetCommands<br>SetCmdState<br>SetCursor<br><b>SetData</b><br><b>SetState</b><br>Show<br>ShowCursor<br><b>SizeLimits</b><br><b>Store</b><br>TopView<br><b>Valid</b><br>WriteBuf<br>WriteChar<br>WriteLine<br>WriteStr | <b>Init</b><br>Awaken<br>ChangeBounds<br><br>DataSize<br>Delete<br><b>Done</b><br>Draw<br>EndModal<br>EventError<br>ExecView<br>Execute<br>First<br>FirstThat<br><br>FocusNext<br>ForEach<br>GetData<br>GetHelpCtx<br>GetSubViewPtr<br><b>HandleEvent</b><br>Insert<br>InsertBefore<br><b>Load</b><br>Lock<br>PutSubViewPtr<br>Redraw<br>SelectNext<br>SetData<br><b>SetState</b><br><b>Store</b><br>Unlock<br>Valid | Init<br>Close<br>Done<br><br>GetPalette<br>GetTitle<br>HandleEvent<br>InitFrame<br>SetState<br>SizeLimits<br>StandardScrollBar<br>Store<br>Zoom |

Tabla 14.1. Campos y métodos de TObject, TView, TGroup y Twindow.

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

| Campos de<br>TView | Campos de<br>TGroup | Campos de<br>TWindow |
|--------------------|---------------------|----------------------|
| Cursor             | Cursor              | Cursor               |
| DragMode           | DragMode            | DragMode             |
| EventMask          | EventMask           | EventMask            |
| GrowMode           | GrowMode            | GrowMode             |
| HelpCtx            | HelpCtx             | HelpCtx              |
| Next               | Next                | Next                 |
| Options            | Options             | Options              |
| Origin             | Origin              | Origin               |
| Owner              | Owner               | Owner                |
| Size               | Size                | Size                 |
| State              | State               | State                |
|                    | Buffer              | Buffer               |
|                    | Current             | Current              |
|                    | Last                | Last                 |
|                    | Phase               | Phase                |
|                    |                     | Flags                |
|                    |                     | Frame                |
|                    |                     | Number               |
|                    |                     | Palette              |
|                    |                     | Title                |
|                    |                     | ZoomRect             |

Tabla 14.2. Herencia de campos de TWindow

En el listado que sigue se puede ver la definición de los tipos objeto citados:

```

 { Tipo objeto TObject }
TObject = object
 constructor Init;
 procedure Free;
 destructor Done; virtual;
end;

 { Tipo objeto TView }
TView = object(TObject)
 Owner: PGroup;
 Next: PView;
 Origin: TPoint;
 Size: TPoint;
 Cursor: TPoint;
 GrowMode: Byte;
 DragMode: Byte;
 HelpCtx: Word;
 State: Word;
 Options: Word;
 EventMask: Word;
 constructor Init(var Bounds: TRect);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Awaken; virtual;
 procedure BlockCursor;
 procedure CalcBounds(var Bounds: TRect; Delta: TPoint); virtual;
 procedure ChangeBounds(var Bounds: TRect); virtual;

```

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
procedure ClearEvent(var Event: TEvent);
function CommandEnabled(Command: Word): Boolean;
function DataSize: Word; virtual;
procedure DisableCommands(Commands: TCommandSet);
procedure DragView(Event: TEvent; Mode: Byte;
 var Limits: TRect; MinSize, MaxSize: TPoint);
procedure Draw; virtual;
procedure DrawView;
procedure EnableCommands(Commands: TCommandSet);
procedure EndModal(Command: Word); virtual;
function EventAvail: Boolean;
function Execute: Word; virtual;
function Exposed: Boolean;
function Focus: Boolean;
procedure GetBounds(var Bounds: TRect);
procedure GetClipRect(var Clip: TRect);
function GetColor(Color: Word): Word;
procedure GetCommands(var Commands: TCommandSet);
procedure GetData(var Rec); virtual;
procedure GetEvent(var Event: TEvent); virtual;
procedure GetExtent(var Extent: TRect);
function GetHelpCtx: Word; virtual;
function GetPalette: PPalette; virtual;
procedure GetPeerViewPtr(var S: TStream; var P);
function GetState(AState: Word): Boolean;
procedure GrowTo(X, Y: Integer);
procedure HandleEvent(var Event: TEvent); virtual;
procedure Hide;
procedure HideCursor;
procedure KeyEvent(var Event: TEvent);
procedure Locate(var Bounds: TRect);
procedure MakeFirst;
procedure MakeGlobal(Source: TPoint; var Dest: TPoint);
procedure MakeLocal(Source: TPoint; var Dest: TPoint);
function MouseEvent(var Event: TEvent; Mask: Word): Boolean;
function MouseInView(Mouse: TPoint): Boolean;
procedure MoveTo(X, Y: Integer);
function NextView: PView;
procedure NormalCursor;
function Prev: PView;
function PrevView: PView;
procedure PutEvent(var Event: TEvent); virtual;
procedure PutInFrontOf(Target: PView);
procedure PutPeerViewPtr(var S: TStream; P: PView);
procedure Select;
procedure SetBounds(var Bounds: TRect);
procedure SetCommands(Commands: TCommandSet);
procedure SetCmdState(Commands: TCommandSet; Enable: Boolean);
procedure SetCursor(X, Y: Integer);
procedure SetData(var Rec); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Show;
procedure ShowCursor;
procedure SizeLimits(var Min, Max: TPoint); virtual;
procedure Store(var S: TStream);
function TopView: PView;
function Valid(Command: Word): Boolean; virtual;
procedure WriteBuf(X, Y, W, H: Integer; var Buf);
procedure WriteChar(X, Y: Integer; C: Char; Color: Byte;
 Count: Integer);
procedure WriteLine(X, Y, W, H: Integer; var Buf);
procedure WriteStr(X, Y: Integer; Str: String; Color: Byte);
private
procedure DrawCursor;
procedure DrawHide>LastView: PView);
procedure DrawShow>LastView: PView);
procedure DrawUnderRect(var R: TRect; LastView: PView);
```

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

procedure DrawUnderView(DoShadow: Boolean; LastView: PView);
procedure ResetCursor; virtual;
end;

```

```

 { Tipo objeto TGroup }
TGroup = object(TView)
 Last: PView;
 Current: PView;
 Phase: (phFocused, phPreProcess, phPostProcess);
 Buffer: PVideoBuf;
 EndState: Word;
constructor Init(var Bounds: TRect);
constructor Load(var S: TStream);
destructor Done; virtual;
procedure Awaken; virtual;
procedure ChangeBounds(var Bounds: TRect); virtual;
function DataSize: Word; virtual;
procedure Delete(P: PView);
procedure Draw; virtual;
procedure EndModal(Command: Word); virtual;
procedure EventError(var Event: TEvent); virtual;
function ExecView(P: PView): Word;
function Execute: Word; virtual;
function First: PView;
function FirstThat(P: Pointer): PView;
function FocusNext(Forwards: Boolean): Boolean;
procedure ForEach(P: Pointer);
procedure GetData(var Rec); virtual;
function GetHelpCtx: Word; virtual;
procedure GetSubViewPtr(var S: TStream; var P);
procedure HandleEvent(var Event: TEvent); virtual;
procedure Insert(P: PView);
procedure InsertBefore(P, Target: PView);
procedure Lock;
procedure PutSubViewPtr(var S: TStream; P: PView);
procedure Redraw;
procedure SelectNext(Forwards: Boolean);
procedure SetData(var Rec); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Store(var S: TStream);
procedure Unlock;
function Valid(Command: Word): Boolean; virtual;
private
 Clip: TRect;
 LockFlag: Byte;
function At(Index: Integer): PView;
procedure DrawSubViews(P, Bottom: PView);
function FirstMatch(AState: Word; AOptions: Word): PView;
function FindNext(Forwards: Boolean): PView;
procedure FreeBuffer;
procedure GetBuffer;
function IndexOf(P: PView): Integer;
procedure InsertView(P, Target: PView);
procedure RemoveView(P: PView);
procedure ResetCurrent;
procedure ResetCursor; virtual;
procedure SetCurrent(P: PView; Mode: SelectMode);
end;

```

```

 { Tipo objeto TWindow }
TWindow = object(TGroup)
 Flags: Byte;
 ZoomRect: TRect;
 Number: Integer;
 Palette: Integer;
 Frame: PFrame;
 Title: PString;

```



## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr; ANumber: Integer);
constructor Load(var S: TStream);
destructor Done; virtual;
procedure Close; virtual;
function GetPalette: PPalette; virtual;
function GetTitle(MaxSize: Integer): TTitleStr; virtual;
procedure HandleEvent(var Event: TEvent); virtual;
procedure InitFrame; virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure SizeLimits(var Min, Max: TPoint); virtual;
function StandardScrollBar(AOptions: Word): PScrollBar;
procedure Store(var S: TStream);
procedure Zoom; virtual;
end;
```

A continuación vamos a pasar a analizar los tipos de métodos que maneja Turbo Vision. Se pueden clasificar en cuatro variantes, algunas de las cuales se solapan entre sí:

- *Métodos estáticos*

Un método estático no puede ser redefinido *per se* como en el caso de los métodos virtuales. Así un tipo descendiente puede definir un método con el mismo nombre utilizando argumentos y tipos de retorno totalmente diferentes, si es necesario, pero los métodos estáticos no operan polimórficamente. Esto es más grave cuando se invoca a métodos de objetos dinámicos.

Por ejemplo, si `PGenerico` es una variable puntero de tipo `PView`, se le pueden asignar punteros de cualquier tipo de la jerarquía. Sin embargo, cuando se derreferencia la variable e invoca a un método estático, el método invocado será siempre el de `TView`, dado que ése es el tipo del puntero que se determinó en tiempo de compilación. `PGenerico^.StaticMethod` es siempre equivalente a `TView.StaticMethod`, incluso si se ha asignado un puntero de algún otro tipo a `PGenerico`. Un ejemplo es `TView.Init`.

- *Métodos virtuales*

Los métodos virtuales utilizan la directiva **virtual** en sus declaraciones de prototipos. Un método virtual puede ser redefinido en descendientes pero el propio método redefinido debe ser virtual y encajar exactamente con la cabecera del método original. Los métodos virtuales no necesitan ser redefinidos obligatoriamente, pero la intención general es que sean redefinidos antes o después. Un ejemplo de esto es `TDialog.HandleEvent` en los tipos objeto `TVentanaFicha` de la aplicación *Agenda*.

- *Métodos abstractos*

Los métodos abstractos se definen siempre como métodos virtuales, tal como se aconsejó en el apartado **Abstracción** del capítulo 13. En el tipo objeto base, un método abstracto tienen un cuerpo vacío o un cuerpo conteniendo la sentencia **Abstract**<sup>50</sup> puesta para atrapar llamadas ilegales. Los métodos abstractos deben ser definidos por un descendiente antes de poder ser usados. Se debe derivar un tipo nuevo y redefinir los métodos abstractos antes de que se pueda crear una instancia utilizable de ese tipo objeto. Un ejemplo es *TStream.Read*.

- *Métodos pseudo-abstractos*

A diferencia de los métodos verdaderamente abstractos que generan un error en tiempo de ejecución, los métodos pseudo-abstractos ofrecen acciones mínimas por defecto o ninguna acción en absoluto. Sirven como recipientes, donde se puede insertar código en los objetos derivados.

Por ejemplo, el tipo *TView* introduce un método virtual denominado *Awaken*. Este no contiene ningún código:

```
procedure TView.Awaken;
begin
end;
```

Por defecto, *Awaken* está claro que no hace nada. *Awaken* es invocado cuando un objeto grupo ha finalizado de cargarse a sí mismo desde un *stream*. Una vez cargadas todas sus subvistas, el grupo invoca al método *Awaken* de cada subvista. Así que si se crea un objeto vista que necesita inicializarse a sí mismo cuando es cargado desde un stream, se puede redefinir *Awaken* para llevar a cabo esa inicialización.

Otro ejemplo de método pseudoabstracto es el método *Idle* de *TApplication*. Este método permite realizar tareas en los tiempos ociosos de un programa. En este caso este método sí realiza alguna acción por defecto.

## LOS TIPOS OBJETO DE LA JERARQUIA DE TURBO VISION

No todos los tipos objeto de Turbo Vision son iguales. Se pueden separar según sus funciones en cuatro grupos distintos:

- *Objetos primitivos*
- *Vistas*
- *Vistas de grupo*
- *Motores*

---

**50** Una llamada al procedimiento **Abstract** hace que finalice la ejecución del programa con un error en tiempo de ejecución 211. Cuando se implementa un tipo objeto abstracto, la llamada al procedimiento **Abstract** debe ser redefinida en los métodos virtuales de tipos descendientes. De esta forma se asegura que cualquier intento de crear instancias de un tipo objeto abstracto no sea posible.

### • Tipos objeto primitivos

Turbo Vision proporciona tres tipos objeto sencillos que están disponibles ante todo para ser utilizados por otros objetos o para servir como base de una jerarquía de tipos objeto más complejos:

- *TPoint*
- *TRect*
- *TObject*

*TPoint* y *TRect* son utilizados por todos los objetos visibles de la jerarquía de Turbo Vision. *TObject* es la base de la jerarquía (Figuras 14.19 y 14.20). Los objetos de estos tipos no son visualizables. *TPoint* es simplemente un objeto de posición de pantalla (coordenadas  $X, Y$ ). *TRect*, por su nombre, parecería ser un objeto visible, pero sólo suministra los límites superior izquierdo e inferior derecho de un rectángulo y varios métodos que son utilidades no visuales.

- ✎ **TPoint** representa un punto. Sus campos,  $X$  e  $Y$ , definen las coordenadas cartesianas ( $X, Y$ ) de una posición de pantalla. El punto (0,0) es la esquina superior izquierda de la pantalla.  $X$  se incrementa horizontalmente hacia la derecha;  $Y$  se incrementa verticalmente hacia abajo. *TPoint* no tiene métodos.

```
TPoint = object
 X, Y: Integer;
end;
```

- ✎ **TRect** representa un rectángulo. Sus campos,  $A$  y  $B$ , son objetos *TPoint* que definen los puntos superior izquierdo e inferior derecho del rectángulo. *TRect* tiene los métodos *Assign*, *Copy*, *Move*, *Grow*, *Intersect*, *Union*, *Contains*, *Equals* y *Empty*. Los objetos *TRect* no son vistas visibles y no pueden dibujarse por sí mismos. Sin embargo, todas las vistas son rectangulares: Todos sus constructores *Init* llevan un parámetro *Bounds* de tipo *TRect* que determina la región que van a cubrir.

```
TRect = object
 A, B: TPoint;
 procedure Assign(XA, YA, XB, YB: Integer);
 procedure Copy(R: TRect);
 procedure Move(ADX, ADY: Integer);
 procedure Grow(ADX, ADY: Integer);
 procedure Intersect(R: TRect);
 procedure Union(R: TRect);
 function Contains(P: TPoint): Boolean;
 function Equals(R: TRect): Boolean;
 function Empty: Boolean;
end;
```

- ✎ **TObject** es un tipo base abstracto sin campos. Es el ascendiente de todos los objetos de Turbo Vision excepto *TPoint* y *TRect* (figura 14.21). *TObject* proporciona tres métodos: *Init*, *Free*, y *Done*. El constructor, *Init*, forma la base para todos los constructores de Turbo Vision proporcionando la reserva de memoria. *Free* libera esta reserva. *Done* es un destructor pseudo-abstracto que debería ser redefinido por sus descendientes.

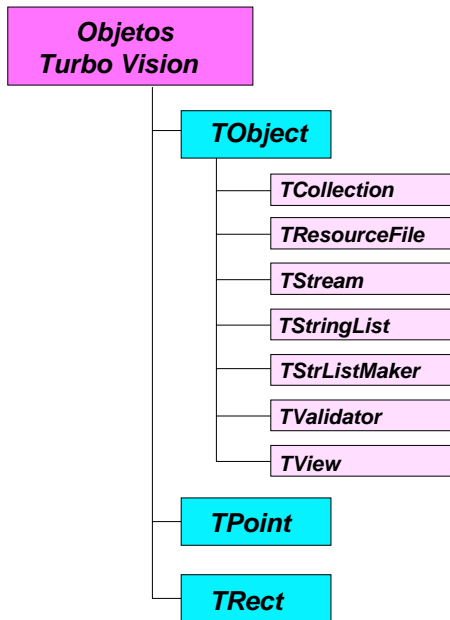


Fig. 14.21 Jerarquía de objetos primitivos

Cualquier objeto que se tenga intención de utilizar con los streams de Turbo Vision debe ser derivado de *TObject*.

Los descendientes de *TObject* pertenecen a una de dos familias posibles: *vistas* o *no-vistas*. Las **vistas** son descendientes de *TView*, que suministra propiedades especiales no compartidas por los objetos *no-vistas*. Las vistas pueden dibujarse a sí mismas y manejar los eventos que les son enviados. Los objetos **no-vista** proporcionan multitud de utilidades para el manejo de streams y colecciones de otros objetos, incluso vistas, pero no son directamente visualizables.

```

TObject = object
 constructor Init;
 procedure Free;
 destructor Done; virtual;
end;

```

## • Vistas

```

TView = object(TObject)
 Owner: PGroup;
 Next: PView;
 Origin: TPoint;
 Size: TPoint;
 Cursor: TPoint;
 GrowMode: Byte;
 DragMode: Byte;
 HelpCtx: Word;
 State: Word;
 Options: Word;
 EventMask: Word;
 constructor Init(var Bounds: TRect);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Awaken; virtual;
 procedure BlockCursor;
 procedure CalcBounds(var Bounds: TRect; Delta: TPoint); virtual;
 procedure ChangeBounds(var Bounds: TRect); virtual;
 procedure ClearEvent(var Event: TEvent);
 function CommandEnabled(Command: Word): Boolean;
 function DataSize: Word; virtual;
 procedure DisableCommands(Commands: TCommandSet);
 procedure DragView(Event: TEvent; Mode: Byte;
 var Limits: TRect; MinSize, MaxSize: TPoint);
 procedure Draw; virtual;
 procedure DrawView;
 procedure EnableCommands(Commands: TCommandSet);

```

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
procedure EndModal(Command: Word); virtual;
function EventAvail: Boolean;
function Execute: Word; virtual;
function Exposed: Boolean;
function Focus: Boolean;
procedure GetBounds(var Bounds: TRect);
procedure GetClipRect(var Clip: TRect);
function GetColor(Color: Word): Word;
procedure GetCommands(var Commands: TCommandSet);
procedure GetData(var Rec); virtual;
procedure GetEvent(var Event: TEvent); virtual;
procedure GetExtent(var Extent: TRect);
function GetHelpCtx: Word; virtual;
function GetPalette: PPalette; virtual;
procedure GetPeerViewPtr(var S: TStream; var P);
function GetState(AState: Word): Boolean;
procedure GrowTo(X, Y: Integer);
procedure HandleEvent(var Event: TEvent); virtual;
procedure Hide;
procedure HideCursor;
procedure KeyEvent(var Event: TEvent);
procedure Locate(var Bounds: TRect);
procedure MakeFirst;
procedure MakeGlobal(Source: TPoint; var Dest: TPoint);
procedure MakeLocal(Source: TPoint; var Dest: TPoint);
function MouseEvent(var Event: TEvent; Mask: Word): Boolean;
function MouseInView(Mouse: TPoint): Boolean;
procedure MoveTo(X, Y: Integer);
function NextView: PView;
procedure NormalCursor;
function Prev: PView;
function PrevView: PView;
procedure PutEvent(var Event: TEvent); virtual;
procedure PutInFrontOf(Target: PView);
procedure PutPeerViewPtr(var S: TStream; P: PView);
procedure Select;
procedure SetBounds(var Bounds: TRect);
procedure SetCommands(Commands: TCommandSet);
procedure SetCmdState(Commands: TCommandSet; Enable: Boolean);
procedure SetCursor(X, Y: Integer);
procedure SetData(var Rec); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Show;
procedure ShowCursor;
procedure SizeLimits(var Min, Max: TPoint); virtual;
procedure Store(var S: TStream);
function TopView: PView;
function Valid(Command: Word): Boolean; virtual;
procedure WriteBuf(X, Y, W, H: Integer; var Buf);
procedure WriteChar(X, Y: Integer; C: Char; Color: Byte;
 Count: Integer);
procedure WriteLine(X, Y, W, H: Integer; var Buf);
procedure WriteStr(X, Y: Integer; Str: String; Color: Byte);
private
procedure DrawCursor;
procedure DrawHide>LastView: PView);
procedure DrawShow>LastView: PView);
procedure DrawUnderRect(var R: TRect; LastView: PView);
procedure DrawUnderView(DoShadow: Boolean; LastView: PView);
procedure ResetCursor; virtual;
end;
```

El listado anterior presenta los campos y métodos de *TView*, a parte de los que hereda de *TObject*. Su elevado número puede dar una idea de la funcionalidad y complejidad de este tipo objeto.

Los descendientes visualizables<sup>51</sup> de *TObject* se conocen como *vistas*, y están derivados de *TView*, un descendiente inmediato de *TObject*.

Una vista es cualquier objeto que puede ser visualizado en una porción rectangular de pantalla. Todos los objetos vista descienden del tipo *TView*. El propio *TView* es un objeto abstracto que representa una área de pantalla rectangular vacía. Teniendo a *TView* como ascendiente, se asegura que cada vista derivada tiene al menos una porción rectangular de la pantalla y un método pseudo-abstracto *Draw* que rellena un rectángulo con un color por defecto.

En la figura 14.22 se pueden observar los tipos objeto vistas que descienden directamente de *TView*. La figura 14.23 muestra la jerarquía de tipos objeto vista de Turbo Vision completa.

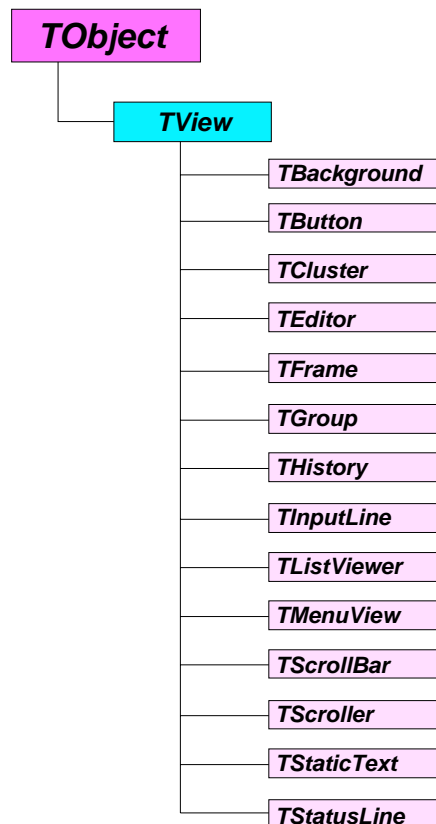


Fig. 14.22 Descendientes directos de *TView*

Turbo Vision incluye las siguientes vistas estándar:

- Marcos
- Botones
- Clusters
- Menús
- Etiquetas
- Historias
- Campos de edición
- Visores de lista
- Vistas con scroll
- Barras de desplazamiento
- Dispositivos de texto
- Textos estáticos
- Líneas de estado

<sup>51</sup> Debería distinguirse entre *visible* y *visualizable*, dado que puede haber veces que una vista esté completa o parcialmente oculta por otras vistas.

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

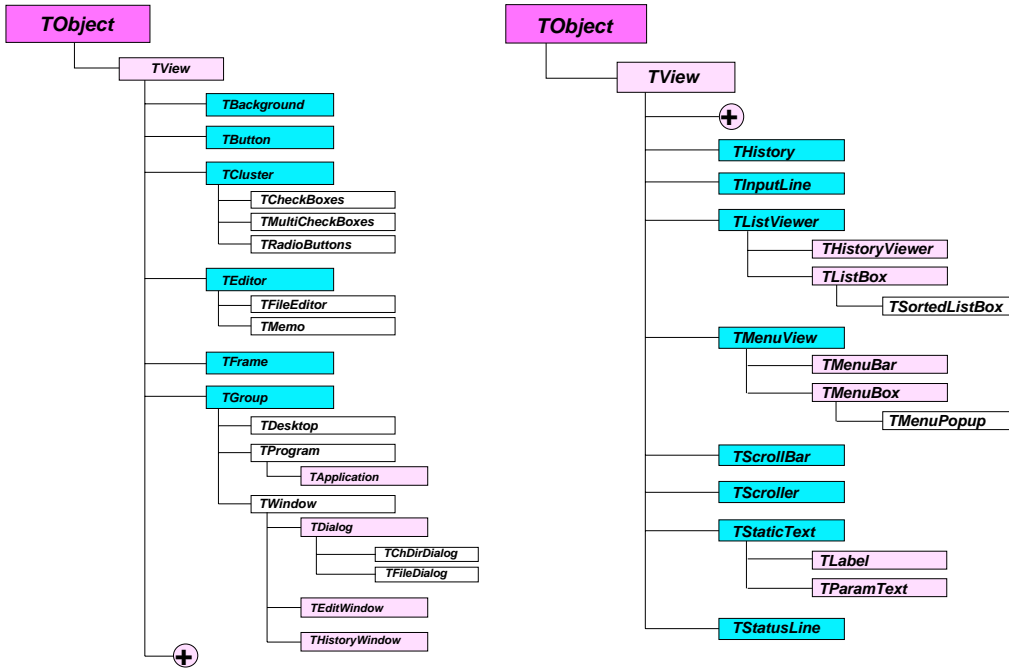


Fig. 14.23 Tipos objeto vista de Turbo Vision

### Marcos

*TFrame* proporciona el marco (borde) visualizable para un objeto *TWindow* junto con iconos para mover y cerrar la ventana. Los objetos *TFrame* no se usan nunca solos, sino conjuntamente con un objeto *TWindow*.

```
TFrame = object(TView)
 constructor Init(var Bounds: TRect);
 procedure Draw; virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure SetState(AState: Word; Enable: Boolean); virtual;
private
 FrameMode: Word;
 procedure FrameLine(var FrameBuf; Y, N: Integer; Color: Byte);
end;
```

### Botones

Un objeto *TButton* es un cuadro con título usado para generar un evento de comando específico cuando es pulsado. Generalmente se colocan dentro de los cuadros de diálogo,

ofreciendo opciones tales como **OK** o **Cancel**. Cuando aparece, el cuadro de diálogo generalmente es la **vista modal**<sup>52</sup>, por lo tanto atrapa y maneja todos los eventos, incluyendo los de sus botones. El manejador de eventos ofrece varias formas de pulsar un botón: pinchando con el ratón en el rectángulo del botón, tecleando el acelerador de teclado, o seleccionando el botón por defecto con la tecla **↵**.

```
TButton = object(TView)
 Title: PString;
 Command: Word;
 Flags: Byte;
 AmDefault: Boolean;
 constructor Init(var Bounds: TRect; ATitle: TTitleStr; ACommand: Word;
 AFlags: Word);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Draw; virtual;
 procedure DrawState(Down: Boolean);
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure MakeDefault(Enable: Boolean);
 procedure Press; virtual;
 procedure SetState(AState: Word; Enable: Boolean); virtual;
 procedure Store(var S: TStream);
end;
```

## Clusters

*TCluster* es un tipo abstracto usado para implementar casillas de verificación y botones de radio. Un cluster es un grupo de controles que responden todos de la misma manera. Los controles de cluster a menudo se asocian con objetos *TLabel*, permitiendo seleccionar el control al activar la etiqueta de texto adyacente.

Los *botones de radio* son clusters especiales en los que sólo un control puede ser seleccionado. Cada selección subsiguiente desactiva la selección anterior (igual que el selector de emisoras de la radio de un coche). Las casillas de verificación son clusters en los que se pueden marcar (seleccionar) cualquier número de controles.

```
TCluster = object(TView)
 Value: LongInt;
 Sel: Integer;
 EnableMask: LongInt;
 Strings: TStringCollection;
 constructor Init(var Bounds: TRect; AStrings: PSItem);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 function ButtonState(Item: Integer): Boolean;
 function DataSize: Word; virtual;
 procedure DrawBox(const Icon: String; Marker: Char);
 procedure DrawMultiBox(const Icon, Marker: String);
```

---

<sup>52</sup> *Modal* significa que una vista es la única parte activa de la aplicación (se puede aplicar en otras partes de la aplicación pero no hay reacción). Una vez que se hace modal una vista (ventana o cuadro de diálogo), no se puede interactuar con ninguna parte de la aplicación fuera de la vista hasta que se cierre o se ejecute otro cuadro de diálogo.



## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
procedure GetData(var Rec); virtual;
function GetHelpCtx: Word; virtual;
function GetPalette: PPalette; virtual;
procedure HandleEvent(var Event: TEvent); virtual;
function Mark(Item: Integer): Boolean; virtual;
function MultiMark(Item: Integer): Byte; virtual;
procedure Press(Item: Integer); virtual;
procedure MovedTo(Item: Integer); virtual;
procedure SetButtonState(AMask: Longint; Enable: Boolean);
procedure SetData(var Rec); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Store(var S: TStream);
private
function Column(Item: Integer): Integer;
function FindSel(P: TPoint): Integer;
function Row(Item: Integer): Integer;
end;

TRadioButtons = object(TCluster)
procedure Draw; virtual;
function Mark(Item: Integer): Boolean; virtual;
procedure MovedTo(Item: Integer); virtual;
procedure Press(Item: Integer); virtual;
procedure SetData(var Rec); virtual;
end;

TCheckBoxes = object(TCluster)
procedure Draw; virtual;
function Mark(Item: Integer): Boolean; virtual;
procedure Press(Item: Integer); virtual;
end;

TMultiCheckBoxes = object(TCluster)
SelRange: Byte;
Flags: Word;
States: PString;
constructor Init(var Bounds: TRect; AStrings: PSItem;
ASelRange: Byte; AFlags: Word; const AStates: String);
constructor Load(var S: TStream);
destructor Done; virtual;
function DataSize: Word; virtual;
procedure Draw; virtual;
procedure GetData(var Rec); virtual;
function MultiMark(Item: Integer): Byte; virtual;
procedure Press(Item: Integer); virtual;
procedure SetData(var Rec); virtual;
procedure Store(var S: TStream);
end;
```

### Menús

*TMenuView* y sus dos descendientes proporcionan los objetos básicos para la creación de menús desplegables (*pull-down*) y submenús anidados a cualquier nivel. Basta con suministrar las cadenas de texto para las selecciones del menú (con aceleradores de teclado resaltados opcionalmente) junto con los comandos asociados con cada selección.

Por defecto, las aplicaciones Turbo Vision reservan la línea superior de la pantalla para una barra de menús, desde la cual caen los cuadros de menú. Se pueden crear cuadros de menú que se desplieguen en respuesta a clicks del ratón.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
TMenuView = object(TView)
 ParentMenu: PMenuView;
 Menu: PMenu;
 Current: PMenuItem;
 constructor Init(var Bounds: TRect);
 constructor Load(var S: TStream);
 function Execute: Word; virtual;
 function FindItem(Ch: Char): PMenuItem;
 procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
 function GetHelpCtx: Word; virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 function HotKey(KeyCode: Word): PMenuItem;
 function NewSubView(var Bounds: TRect; AMenu: PMenu;
 AParentMenu: PMenuView): PMenuView; virtual;
 procedure Store(var S: TStream);
end;
```

### Historias

El tipo abstracto *THistory* implementa un mecanismo genérico *pick-list* (lista de selección). *THistory* trabaja en conjunción con *THistoryWindow* y *THistoryViewer*.

```
THistory = object(TView)
 Link: PInputLine;
 HistoryId: Word;
 constructor Init(var Bounds: TRect; ALink: PInputLine; AHistoryId: Word);
 constructor Load(var S: TStream);
 procedure Draw; virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 function InitHistoryWindow(var Bounds: TRect): PHistoryWindow; virtual;
 procedure RecordHistory(const S: String); virtual;
 procedure Store(var S: TStream);
end;
```

### Campos de edición

*TInputLine* proporciona un campo editor de cadenas básico. Maneja todas las entradas del teclado usuales y movimientos del cursor. Ofrece borrados e inserciones, modos de inserción y sobrescritura seleccionables, y control automático de la forma del cursor.

Los campos de edición soportan validación de datos con objetos de validación de alguno de los tipos objeto descendientes del tipo objeto de validación abstracto *TValidator* (*TPXPictureValidator*, *TFilterValidator*, *TLookupValidator*, *TRangeValidator* y *TStringLookupValidator*)

```
TInputLine = object(TView)
 Data: PString;
 MaxLen: Integer;
 CurPos: Integer;
 FirstPos: Integer;
 SelStart: Integer;
 SelEnd: Integer;
 Validator: PValidator;
 constructor Init(var Bounds: TRect; AMaxLen: Integer);
 constructor Load(var S: TStream);
```

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
destructor Done; virtual;
function DataSize: Word; virtual;
procedure Draw; virtual;
procedure GetData(var Rec); virtual;
function GetPalette: PPalette; virtual;
procedure HandleEvent(var Event: TEvent); virtual;
procedure SelectAll(Enable: Boolean);
procedure SetData(var Rec); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure SetValidator(AValid: PValidator);
procedure Store(var S: TStream);
function Valid(Command: Word): Boolean; virtual;
private
function CanScroll(Delta: Integer): Boolean;
end;

TValidator = object(TObject)
Status: Word;
Options: Word;
constructor Init;
constructor Load(var S: TStream);
procedure Error; virtual;
function IsValidInput(var S: string;
 SuppressFill: Boolean): Boolean; virtual;
function IsValid(const S: string): Boolean; virtual;
procedure Store(var S: TStream);
function Transfer(var S: String; Buffer: Pointer;
 Flag: TVTransfer): Word; virtual;
function Valid(const S: string): Boolean;
end;
```

### Visores de lista

El tipo objeto *TListViewer* es un tipo base abstracto desde el cual se derivan visores de lista tales como *TListBox*. Los campos y métodos de *TListViewer* permiten visualizar listas enlazadas de cadenas con control sobre una o dos barras de desplazamiento. *TListBox*, derivado de *TListViewer*, implementa los cuadros de lista más comúnmente utilizados, como las que muestran listas de cadenas con nombres de ficheros.

```
TListViewer = object(TView)
HScrollBar: PScrollBar;
VScrollBar: PScrollBar;
NumCols: Integer;
TopItem: Integer;
Focused: Integer;
Range: Integer;
constructor Init(var Bounds: TRect; ANumCols: Word;
 AHScrollBar, AVScrollBar: PScrollBar);
constructor Load(var S: TStream);
procedure ChangeBounds(var Bounds: TRect); virtual;
procedure Draw; virtual;
procedure FocusItem(Item: Integer); virtual;
function GetPalette: PPalette; virtual;
function GetText(Item: Integer; MaxLen: Integer): String; virtual;
function IsSelected(Item: Integer): Boolean; virtual;
procedure HandleEvent(var Event: TEvent); virtual;
procedure SelectItem(Item: Integer); virtual;
procedure SetRange(ARange: Integer);
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Store(var S: TStream);
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
private
 procedure FocusItemNum(Item: Integer); virtual;
end;
```

### Vistas con scroll

Un objeto *TScroller* es una vista con capacidad de desplazamiento de texto (*scroll*) que sirve como escenario a otra vista que maneja un texto cuyo tamaño desborda la capacidad de la pantalla. El *scroll* se efectúa en respuesta a entradas desde el teclado o acciones en los objetos *TScrollBar* asociados.

```
TScroller = object(TView)
 HScrollBar: PScrollBar;
 VScrollBar: PScrollBar;
 Delta: TPoint;
 Limit: TPoint;
 constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar:PScrollBar);
 constructor Load(var S: TStream);
 procedure ChangeBounds(var Bounds: TRect); virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure ScrollDraw; virtual;
 procedure ScrollTo(X, Y: Integer);
 procedure SetLimit(X, Y: Integer);
 procedure SetState(AState: Word; Enable: Boolean); virtual;
 procedure Store(var S: TStream);
private
 DrawLock: Byte;
 DrawFlag: Boolean;
 procedure CheckDraw;
end;
```

### Barras de desplazamiento

Los objetos *TScrollBar* proporcionan control vertical u horizontal. Las ventanas que contienen interiores con *scroll* utilizan barras de desplazamiento para controlar la posición del *scroll*. Los visores de lista también utilizan barras de desplazamiento.

```
TScrollBar = object(TView)
 Value: Integer;
 Min: Integer;
 Max: Integer;
 PgStep: Integer;
 ArStep: Integer;
 constructor Init(var Bounds: TRect);
 constructor Load(var S: TStream);
 procedure Draw; virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure ScrollDraw; virtual;
 function ScrollStep(Part: Integer): Integer; virtual;
 procedure SetParams(AValue, AMin, AMax, APgStep, AArStep: Integer);
 procedure SetRange(AMin, AMax: Integer);
 procedure SetStep(APgStep, AArStep: Integer);
 procedure SetValue(AValue: Integer);
 procedure Store(var S: TStream);
private
 Chars: TScrollChars;
```

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
procedure DrawPos(Pos: Integer);
function GetPos: Integer;
function GetSize: Integer;
end;
```

### Dispositivos de texto

Un *TTextDevice* es un controlador de dispositivo visor de texto de tipo TTY. Aparte de los campos y métodos heredados de *TScroller*, *TTextDevice* define métodos virtuales de lectura y escritura sobre el dispositivo. *TTextDevice* existe únicamente como tipo base para la derivación de controladores de terminales reales. *TTerminal* implementa un terminal *mudo* con lecturas y escrituras de cadenas mediante buffers. En esencia es un controlador de dispositivo fichero de texto que escribe en una ventana.

Estos tipos objeto está implementado en la Unit *TextView*.

```
TTextDevice = object(TScroller)
function StrRead(var S: TextBuf): Byte; virtual;
procedure StrWrite(var S: TextBuf; Count: Byte); virtual;
end;

TTerminal = object(TTextDevice)
BufSize: Word;
Buffer: PTerminalBuffer;
QueFront, QueBack: Word;
constructor Init(var Bounds:TRect; AHScrollBar, AVScrollBar: PScrollBar;
ABufSize: Word);
destructor Done; virtual;
procedure BufDec(var Val: Word);
procedure BufInc(var Val: Word);
function CalcWidth: Integer;
function CanInsert(Amount: Word): Boolean;
procedure Draw; virtual;
function NextLine(Pos:Word): Word;
function PrevLines(Pos:Word; Lines: Word): Word;
function StrRead(var S: TextBuf): Byte; virtual;
procedure StrWrite(var S: TextBuf; Count: Byte); virtual;
function QueEmpty: Boolean;
end;
```

### Textos estáticos

Los objetos *TStaticText* son vistas sencillas utilizadas para visualizar cadenas fijas proporcionadas por el campo *Text*. Ignoran todos los eventos que se les envía. El tipo *TLabel* añade la propiedad de que la vista que contiene el texto, conocida como etiqueta, puede ser seleccionada (resaltada) mediante un click del ratón, tecla de cursor, o acelerador de teclado **Alt**+**letra**. Las etiquetas se asocian con otra vista, generalmente una vista control. Seleccionando la etiqueta se selecciona el control enlazado y seleccionando el control enlazado se resalta la etiqueta también.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
TStaticText = object(TView)
 Text: PString;
 constructor Init(var Bounds: TRect; const AText: String);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Draw; virtual;
 function GetPalette: PPalette; virtual;
 procedure GetText(var S: String); virtual;
 procedure Store(var S: TStream);
end;

TLabel = object(TStaticText)
 Link: PView;
 Light: Boolean;
 constructor Init(var Bounds: TRect; const AText: String; ALink: PView);
 constructor Load(var S: TStream);
 procedure Draw; virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure Store(var S: TStream);
end;
```

### Líneas de estado

Un objeto *TStatusLine* se utiliza para presentar visualizaciones del estado de la aplicación y de indicaciones de ayuda (*hint*), generalmente en la línea inferior de la pantalla. Una línea de estado es una franja de un carácter de alto y longitud cualquiera hasta el ancho total de la pantalla. El objeto ofrece visualizaciones dinámicas (varían según el estado del programa) que reaccionan a eventos de la aplicación que se explica.

```
TStatusLine = object(TView)
 Items: PStatusItem;
 Defs: PStatusDef;
 constructor Init(var Bounds: TRect; ADefs: PStatusDef);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Draw; virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 function Hint(AHelpCtx: Word): String; virtual;
 procedure Store(var S: TStream);
 procedure Update; virtual;
private
 procedure DrawSelect(Selected: PStatusItem);
 procedure FindItems;
end;
```

### • Vistas de grupo

La importancia de *TView* se ve claramente al observar el gráfico de jerarquía mostrado en la figura 14.20. Todo lo que se puede ver en una aplicación Turbo Vision deriva de alguna manera de *TView*. Pero algunos de esos objetos visibles son también importantes por otra razón: hacen que los objetos no actúen de forma individual.

Turbo Vision incluye las siguientes vistas de grupo estándar:

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

- El grupo abstracto
- Aplicaciones
- Desktops
- Ventanas
- Cuadros de diálogo

### Grupo abstracto

*TGroup* permite manejar dinámicamente listas encadenadas de *subvistas* relacionadas que interactúan entre sí, mediante una vista denominada la *propietaria* del grupo. Dado que un grupo es una vista, puede haber subvistas que sean a su vez grupos que poseen sus propias subvistas, y así sucesivamente. El estado de la cadena está en constante cambio a medida que el usuario interactúa con una aplicación. Se pueden crear nuevos grupos y añadir (insertar) y borrar subvistas en un grupo.

```
TGroup = object(TView)
 Last: PView;
 Current: PView;
 Phase: (phFocused, phPreProcess, phPostProcess);
 Buffer: PVideoBuf;
 EndState: Word;
 constructor Init(var Bounds: TRect);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Awaken; virtual;
 procedure ChangeBounds(var Bounds: TRect); virtual;
 function DataSize: Word; virtual;
 procedure Delete(P: PView);
 procedure Draw; virtual;
 procedure EndModal(Command: Word); virtual;
 procedure EventError(var Event: TEvent); virtual;
 function ExecView(P: PView): Word;
 function Execute: Word; virtual;
 function First: PView;
 function FirstThat(P: Pointer): PView;
 function FocusNext(Forwards: Boolean): Boolean;
 procedure ForEach(P: Pointer);
 procedure GetData(var Rec); virtual;
 function GetHelpCtx: Word; virtual;
 procedure GetSubViewPtr(var S: TStream; var P);
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure Insert(P: PView);
 procedure InsertBefore(P, Target: PView);
 procedure Lock;
 procedure PutSubViewPtr(var S: TStream; P: PView);
 procedure Redraw;
 procedure SelectNext(Forwards: Boolean);
 procedure SetData(var Rec); virtual;
 procedure SetState(AState: Word; Enable: Boolean); virtual;
 procedure Store(var S: TStream);
 procedure Unlock;
 function Valid(Command: Word): Boolean; virtual;
private
 Clip: TRect;
 LockFlag: Byte;
 function At(Index: Integer): PView;
 procedure DrawSubViews(P, Bottom: PView);
 function FirstMatch(AState: Word; AOptions: Word): PView;
 function FindNext(Forwards: Boolean): PView;
 procedure FreeBuffer;
 procedure GetBuffer;
 function IndexOf(P: PView): Integer;
 procedure InsertView(P, Target: PView);
 procedure RemoveView(P: PView);
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
procedure ResetCurrent;
procedure ResetCursor; virtual;
procedure SetCurrent(P: PView; Mode: SelectMode);
end;
```

### Aplicaciones

*TProgram* es un tipo abstracto que proporciona una serie de métodos virtuales a su descendiente, *TApplication*. *TApplication* proporciona un objeto a modo de patrón de programa para una aplicación Turbo Vision. Es un descendiente de *TGroup* (vía *TProgram*). Típicamente, posee subvistas *TMenuBar*, *TDesktop* y *TStatusLine*. *TApplication* tiene métodos para la creación e inserción de estas tres subvistas. El método clave de *TApplication* es *TApplication.Run* el cual ejecuta el código de la aplicación.

```
TProgram = object(TGroup)
 constructor Init;
 destructor Done; virtual;
 function CanMoveFocus: Boolean;
 function ExecuteDialog(P: PDialog; Data: Pointer): Word;
 procedure GetEvent(var Event: TEvent); virtual;
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure Idle; virtual;
 procedure InitDesktop; virtual;
 procedure InitMenuBar; virtual;
 procedure InitScreen; virtual;
 procedure InitStatusLine; virtual;
 function InsertWindow(P: PWindow): PWindow;
 procedure OutOfMemory; virtual;
 procedure PutEvent(var Event: TEvent); virtual;
 procedure Run; virtual;
 procedure SetScreenMode(Mode: Word);
 function ValidView(P: PView): PView;
end;

TApplication = object(TProgram)
 constructor Init;
 destructor Done; virtual;
 procedure Cascade;
 procedure DosShell;
 procedure GetTileRect(var R: TRect); virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure Tile;
 procedure WriteShellMsg; virtual;
end;
```

### Desktops (escritorios)

*TDesktop* es la vista normal de arranque que aparece en el fondo, proporcionando el familiar desktop de usuario, generalmente rodeado por una barra de menús y una línea de estado. Otras vistas (por ejemplo ventanas y cuadros de diálogo) son creadas, visualizadas, y manipuladas en el desktop en respuesta a acciones del usuario (eventos de ratón y teclado). La mayoría del trabajo efectivo de una aplicación ocurre dentro del *desktop*.



## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

```
TDesktop = object(TGroup)
 Background: PBackground;
 TileColumnsFirst: Boolean;
 constructor Init(var Bounds: TRect);
 constructor Load(var S: TStream);
 procedure Cascade(var R: TRect);
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure InitBackground; virtual;
 procedure Store(var S: TStream);
 procedure Tile(var R: TRect);
 procedure TileError; virtual;
end;
```

### Ventanas

Los objetos *TWindow*, con la ayuda de los objetos *TFrame*, son los rectángulos con borde que se pueden arrastrar, redimensionar, y ocultar utilizando los métodos heredados de *TView*. Un objeto ventana también puede hacer un zoom y cerrarse a sí mismo utilizando sus propios métodos. Las ventanas numeradas se pueden seleccionar con las hot keys **Alt+n°**.

```
TWindow = object(TGroup)
 Flags: Byte;
 ZoomRect: TRect;
 Number: Integer;
 Palette: Integer;
 Frame: PFrame;
 Title: PString;
 constructor Init(var Bounds: TRect; ATitle: TTitleStr; ANumber: Integer);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 procedure Close; virtual;
 function GetPalette: PPalette; virtual;
 function GetTitle(MaxSize: Integer): TTitleStr; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 procedure InitFrame; virtual;
 procedure SetState(AState: Word; Enable: Boolean); virtual;
 procedure SizeLimits(var Min, Max: TPoint); virtual;
 function StandardScrollBar(AOptions: Word): PScrollBar;
 procedure Store(var S: TStream);
 procedure Zoom; virtual;
end;
```

### Cuadros de diálogo

*TDialog* es un descendiente de *TWindow* utilizado para crear cuadros de diálogo que manejan una variedad de interacciones de usuario. Generalmente los cuadros de diálogo contienen controles tales como botones y casillas de verificación. La diferencia principal entre cuadros de diálogo y ventanas es que los cuadros de diálogo están especializados en la operación modal.

```
TDialog = object(TWindow)
 constructor Init(var Bounds: TRect; ATitle: TTitleStr);
 constructor Load(var S: TStream);
 function GetPalette: PPalette; virtual;
 procedure HandleEvent(var Event: TEvent); virtual;
 function Valid(Command: Word): Boolean; virtual;
end;
```

• Motores

Turbo Vision incluye cinco grupos de tipos objeto no-vistas derivados de *TObject* cuya jerarquía se puede observar en la figura 14.24:

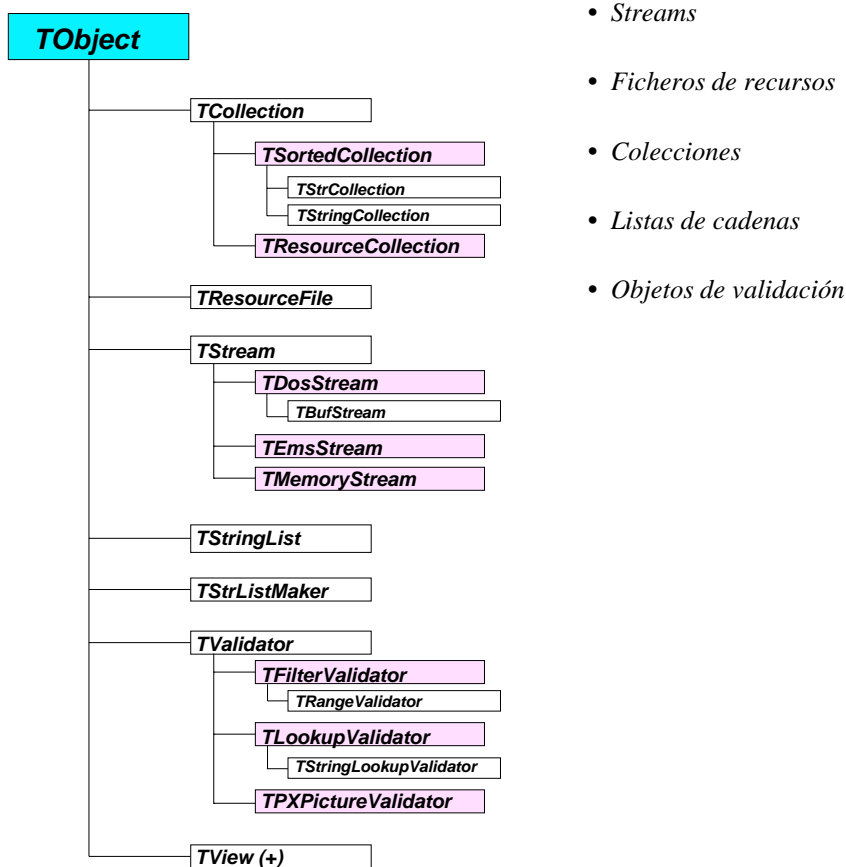


Fig. 14.24 Jerarquía de objetos motores

**Streams**

Un stream es un objeto generalizado para el manejo de entrada y salida. En la E/S tradicional de dispositivos y ficheros, se tenían que implementar diferentes conjuntos de funciones para la extracción y conversión de diferentes tipos de datos. Con los streams de Turbo Vision, se pueden crear métodos polimórficos de E/S como por ejemplo *Read* y *Write* que saben cómo procesar sus propios contenidos particulares del stream.

## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

*TStream* es el objeto abstracto base que proporciona E/S polimórfica sobre un dispositivo de almacenamiento. Turbo Vision además incluye streams especializados, incluyendo streams de ficheros DOS, streams DOS con buffers, streams en memoria, y streams EMS.

```
TStream = object(TObject)
 Status: Integer;
 ErrorInfo: Integer;
 constructor Init;
 procedure CopyFrom(var S: TStream; Count: Longint);
 procedure Error(Code, Info: Integer); virtual;
 procedure Flush; virtual;
 function Get: PObject;
 function GetPos: Longint; virtual;
 function GetSize: Longint; virtual;
 procedure Put(P: PObject);
 procedure Read(var Buf; Count: Word); virtual;
 function ReadStr: PString;
 procedure Reset;
 procedure Seek(Pos: Longint); virtual;
 function StrRead: PChar;
 procedure StrWrite(P: PChar);
 procedure Truncate; virtual;
 procedure Write(var Buf; Count: Word); virtual;
 procedure WriteStr(P: PString);
end;
```

### Recursos

Un fichero de recursos es una clase especial de stream en el que se pueden indexar objetos genéricos mediante cadenas clave. En vez de derivar los ficheros de recursos de *TStream*, utiliza el tipo objeto *TResourceFile* que tiene un campo *Stream* que asocia un stream con el fichero de recursos.

```
TResourceFile = object(TObject)
 Stream: PStream;
 Modified: Boolean;
 constructor Init(AStream: PStream);
 destructor Done; virtual;
 function Count: Integer;
 procedure Delete(Key: String);
 procedure Flush;
 function Get(Key: String): PObject;
 function KeyAt(I: Integer): String;
 procedure Put(Item: PObject; Key: String);
 function SwitchTo(AStream: PStream; Pack: Boolean): PStream;
private
 BasePos: Longint;
 IndexPos: Longint;
 Index: TResourceCollection;
end;
```

### Colecciones

*TCollection* implementa un conjunto general de elementos, incluyendo objetos arbitrarios de tipos diferentes. Al contrario que los arrays, conjuntos, y listas enlazadas, las colecciones de Turbo Vision permiten dimensionamiento dinámico. *TCollection* es una base abstracta para

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

colecciones más especializadas. Turbo Vision incluye varios tipos de colecciones especializadas: una colección ordenada abstracta *TSortedCollection*, colecciones de cadenas *TStringCollection* y colecciones de recursos *TResourceCollection*.

```

TCollection = object(TObject)
 Items: PItemList;
 Count: Integer;
 Limit: Integer;
 Delta: Integer;
 constructor Init(ALimit, ADelta: Integer);
 constructor Load(var S: TStream);
 destructor Done; virtual;
 function At(Index: Integer): Pointer;
 procedure AtDelete(Index: Integer);
 procedure AtFree(Index: Integer);
 procedure AtInsert(Index: Integer; Item: Pointer);
 procedure AtPut(Index: Integer; Item: Pointer);
 procedure Delete(Item: Pointer);
 procedure DeleteAll;
 procedure Error(Code, Info: Integer); virtual;
 function FirstThat(Test: Pointer): Pointer;
 procedure ForEach(Action: Pointer);
 procedure Free(Item: Pointer);
 procedure FreeAll;
 procedure FreeItem(Item: Pointer); virtual;
 function GetItem(var S: TStream): Pointer; virtual;
 function IndexOf(Item: Pointer): Integer; virtual;
 procedure Insert(Item: Pointer); virtual;
 function LastThat(Test: Pointer): Pointer;
 procedure Pack;
 procedure PutItem(var S: TStream; Item: Pointer); virtual;
 procedure SetLimit(ALimit: Integer); virtual;
 procedure Store(var S: TStream);
end;

TSortedCollection = object(TCollection)
 Duplicates: Boolean;
 constructor Init(ALimit, ADelta: Integer);
 constructor Load(var S: TStream);
 function Compare(Key1, Key2: Pointer): Integer; virtual;
 function IndexOf(Item: Pointer): Integer; virtual;
 procedure Insert(Item: Pointer); virtual;
 function KeyOf(Item: Pointer): Pointer; virtual;
 function Search(Key: Pointer; var Index: Integer): Boolean; virtual;
 procedure Store(var S: TStream);
end;

TStringCollection = object(TSortedCollection)
 function Compare(Key1, Key2: Pointer): Integer; virtual;
 procedure FreeItem(Item: Pointer); virtual;
 function GetItem(var S: TStream): Pointer; virtual;
 procedure PutItem(var S: TStream; Item: Pointer); virtual;
end;

TResourceCollection = object(TStringCollection)
 procedure FreeItem(Item: Pointer); virtual;
 function GetItem(var S: TStream): Pointer; virtual;
 function KeyOf(Item: Pointer): Pointer; virtual;
 procedure PutItem(var S: TStream; Item: Pointer); virtual;
end;

```

## Listas de cadenas

*TStringList* implementa una clase especial de recurso de cadenas en el cual se puede acceder a las cadenas mediante un índice numérico. *TStringList* simplifica la internacionalización de las aplicaciones con texto multilingüe. *TStringList* ofrece acceso sólo a listas de cadenas existentes numéricamente indexadas. *TStrListMaker* aporta un método *Put* para la introducción de una cadena en una lista de cadenas, y un método *Store* para guardar listas de cadenas en un stream.

```
TStringList = object(TObject)
 constructor Load(var S: TStream);
 destructor Done; virtual;
 function Get(Key: Word): String;
private
 Stream: PStream;
 BasePos: Longint;
 IndexSize: Integer;
 Index: PStrIndex;
 procedure ReadStr(var S: String; Offset, Skip: Word);
end;

TStrListMaker = object(TObject)
 constructor Init(AStrSize, AIndexSize: Word);
 destructor Done; virtual;
 procedure Put(Key: Word; S: String);
 procedure Store(var S: TStream);
private
 StrPos: Word;
 StrSize: Word;
 Strings: PByteArray;
 IndexPos: Word;
 IndexSize: Word;
 Index: PStrIndex;
 Cur: TStrIndexRec;
 procedure CloseCurrent;
end;
```

## Objetos de validación

*TValidator* es un objeto de validación abstracto que sirve de base para una familia de tipos objeto utilizados para validar los contenidos de campos de edición. Los objetos de validación utilizables *TFilterValidator*, *TRangeValidator*, *TLookupValidator*, *TStringLookupValidator*, y *TPXPictureValidator* derivan su comportamiento básico de *TValidator*, que proporcionan formas diferentes de validación.

```
TValidator = object(TObject)
 Status: Word;
 Options: Word;
 constructor Init;
 constructor Load(var S: TStream);
 procedure Error; virtual;
 function IsValidInput(var S: String;
 SuppressFill: Boolean): Boolean; virtual;
 function IsValid(const S: String): Boolean; virtual;
 procedure Store(var S: TStream);
 function Transfer(var S: String; Buffer: Pointer);
```

```

 Flag: TVTransfer): Word; virtual;
function Valid(const S: String): Boolean;
end;

```

### SISTEMA DE COORDENADAS EN TURBO VISION

Al contrario que los sistemas de coordenadas que designan los espacios de carácter en la pantalla, las coordenadas Turbo Vision especifican la rejilla entre los caracteres.

Un *punto del sistema de coordenadas* se designa mediante sus coordenadas X e Y. El tipo objeto *TPoint* encapsula las coordenadas en sus campos, *X* e *Y*. *TPoint* no tiene métodos como ya se vio en el apartado **Tipos objeto primitivos**, pero hace sencillo el manejo de ambas coordenadas en un elemento único.

Todo elemento sobre una pantalla Turbo Vision es rectangular, definido por un objeto rectángulo de tipo *TRect*. *TRect* tiene dos campos, *A* y *B*, cada uno de los cuales es un *TPoint*, donde *A* representa la esquina superior izquierda y *B* contiene la esquina inferior derecha. Cuando se especifica los límites de un objeto vista, se pasan esos límites al constructor de la vista en un objeto *TRect* constructor `Init(var Limites: TRect);`

Por ejemplo, si *R* es un objeto *TRect*, `R.Assign(0,0,0,0)` designa un rectángulo sin tamaño —es sólo un punto. El rectángulo más pequeño que puede realmente contener algo se crearía con `R.Assign(0,0,1,1)`.

La figura 14.25 muestra un *TRect* creado con `R.Assign(4,3,7,5)`, rectángulo que contiene seis espacios de carácter. Este tipo de sistema de coordenadas simplifica el cálculo de cosas tales como tamaños de rectángulos y coordenadas de rectángulos adyacentes.

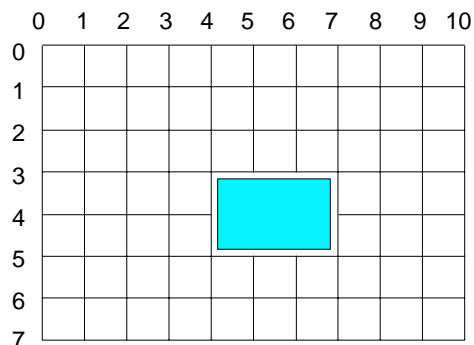


Fig. 14.25 Sistema de coordenadas

La mayor parte del tiempo, una vista sólo trata con su propio sistema de **coordenadas local**, el cual tiene su origen en la esquina superior izquierda de la vista. Cuando se coloca un control en un cuadro de diálogo, por ejemplo, se especifica su posición relativa al origen del cuadro de diálogo. De esa manera, cuando se mueve el cuadro de diálogo, el control se mueve con él. El sistema de **coordenadas global** de una aplicación tiene su origen en la esquina superior izquierda de la pantalla.

El único momento en que hay que preocuparse de otro sistema de coordenadas que no sea el local es cuando se manejan *eventos posicionales* tales como clicks del ratón. Los clicks de ratón son manejados por la aplicación, la cual registra la posición del click en su sistema de coordenadas global. Determinando en qué lugar de la pantalla hizo el usuario click con el ratón, la aplicación puede decidir qué vista de la pantalla debería responder al evento.

Cuando una vista necesita responder a un evento así, tiene que convertir las coordenadas globales en coordenadas locales. Toda vista hereda un método denominado *MakeLocal* que convierte un punto en coordenadas globales de pantalla a coordenadas locales de vista. Si es necesario, se puede pasar de coordenadas locales a globales, utilizando otro método, *MakeGlobal*.

### LOS CAMPOS DE PROPIEDADES DE VISTAS

Las vistas de Turbo Vision utilizan varios campos con formato *bitmap* o campos binarios. Esto campos utilizan los bits individuales de un byte o palabra para indicar diferentes propiedades. Los bits individuales se denominan corrientemente *flags*, ya que estando activados (iguales a 1) o desactivados (iguales a 0), indican si la propiedad designada está o no activada.

Por ejemplo, cada vista tiene un campo bitmap de tipo *Word* denominado *Options*. Cada uno de los bits individuales de la palabra tiene un significado diferente para Turbo Vision. La figura 14.26 muestra las definiciones de los bits de la palabra *Options*. *msb* indica el *bit más significativo* (most significant bit), también denominado *bit de orden alto* porque en la construcción de un número binario, ese bit tiene el mayor peso ( $2^{15}$ ). El bit del final del número binario está marcado como *lsb*, por ser el *bit menos significativo* (least significant bit), también llamado el *bit de orden bajo*.

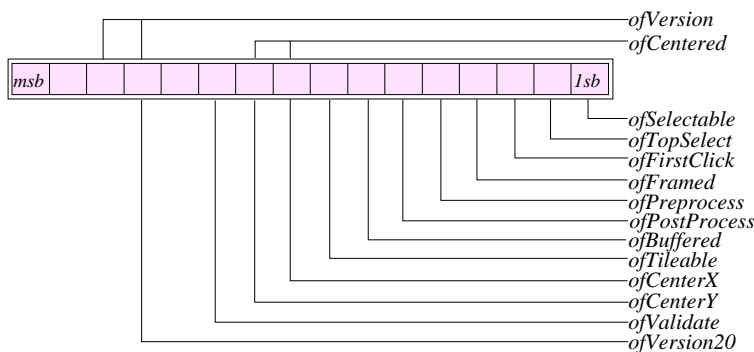


Fig. 14.26 Flags del campo con formato *bitmap Options*

Así, por ejemplo, el cuarto bit se denomina *ofFramed*. Si el bit *ofFramed* está puesto a 1, significa que la vista tiene un marco visible a su alrededor. Si el bit está a 0, la vista no tiene marco.

Generalmente no hay que preocuparse de los valores que tienen los bits de flag a no ser que se planea definir un campo bitmap propio, y aún en ese caso, sólo hay que asegurarse de que dichas definiciones son únicas. Los bits de mayor orden de la palabra *Options* están actualmente sin definir por Turbo Vision.

Para manipular los campos *bitmap* se utilizan operadores de manejo de bits<sup>53</sup> y máscaras de bits.

• **Máscaras de bits**

Una *máscara* es una forma de simplificar el tratamiento de un grupo de flags de bits al mismo tiempo. Por ejemplo, Turbo Vision define máscaras para diferentes clases de eventos. La máscara *evMouse* simplemente contiene los cuatro bits que designan las diferentes clases de eventos de ratón, así si una vista necesita comprobar si hay eventos de ratón, puede comparar el tipo de evento para ver si está en la máscara, en vez de tener que chequear cada una de las clases individuales de eventos de ratón.

```

{ Códigos de eventos54 } { Máscaras de eventos }
evMouseDown = $0001; evNothing = $0000;
evMouseUp = $0002; evMouse = $000F;
evMouseMove = $0004; evKeyboard = $0010;
evMouseAuto = $0008; evMessage = $FF00;
evKeyDown = $0010;
evCommand = $0100;
evBroadcast = $0200;

```

• **Operaciones de bits para la manipulación de campos *bitmap***

Turbo Pascal proporciona operaciones para la manipulación de bits individuales. La tabla 14.3 muestra como se pueden utilizar para manejar (activar, desactivar, cambiar y comprobar su valor) los flags de los campos *bitmap*<sup>55</sup>.

| Operación                     | Implementación                                          |
|-------------------------------|---------------------------------------------------------|
| Activar un flag               | campo := campo <b>or</b> flag;                          |
| Desactivar un flag            | campo := campo <b>and not</b> flag;                     |
| Cambiar un flag               | campo := campo <b>xor</b> flag;                         |
| Comprobar un flag             | <b>if</b> campo <b>and</b> campo = flag <b>then</b> ... |
| Comprobar un flag con máscara | <b>if</b> campo <b>and</b> mascara <> 0 <b>then</b> ... |

Tabla 14.3 Operaciones sobre campos *bitmap*.

Veamos algunos ejemplos de utilización de estas operaciones.

<sup>53</sup> Los operadores de manejo de bits se estudiaron en el capítulo 12.

<sup>54</sup> Se utilizan para activar un flag de una determinada propiedad

<sup>55</sup> Recuérdese que los operadores de manejo de bits tienen menos prioridad que los relacionales.



## TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO

Para **activar** el bit *ofPostProcess* en el campo *Options* de un botón denominado *Boton*, se utiliza el código:

```
Boton.Options := Boton.Options or ofPostProcess;
```

No se debería utilizar la suma para activar bits a no ser que se esté absolutamente seguro de lo que se está haciendo. Por ejemplo, si en vez del código anterior, se utiliza

```
Boton.Options := Boton.Options + ofPostProcess;
```

la operación funcionaría si y sólo si el bit *ofPostProcess* no estaba ya activado. Si el bit estaba activado, la suma binaria llevaría el acarreo al siguiente bit (*ofBuffered*), activándolo o desactivándolo, dependiendo de si estaba o no activado al comienzo.

Se podrían activar varios bits en una única operación haciendo un **or** sobre el campo con varios bits al mismo tiempo. El código siguiente activa a la vez dos flags del modo de crecimiento de una vista *scroll* denominada *Desplazador*:

```
Desplazador.GrowMode := Desplazador.GrowMode or gfGrowHiX or gfGrowHiY;
```

Para **desactivar** el bit *dmLimitLoX* del campo *DragMode* de una etiqueta *TLabel* denominada *Etiqueta*, se hace:

```
Etiqueta.DragMode := Etiqueta.DragMode and not dmLimitLoX;
```

Al igual que en la activación de bits, se pueden desactivar múltiples bits con una única operación.

Para **cambiar** el centrado horizontal de un cuadro de diálogo *TDialog* situado en el *desktop* de nombre *Dialogo*, se cambia el bit *ofCenterX* de la siguiente manera:

```
Dialogo.Options := Dialogo.Options xor ofCenterX;
```

Para **comprobar** si la ventana *Ventana* puede aparecer en cascada o en mosaico en el *desktop*, hay que chequear el flag de opción *ofTileable*:

```
Ventana.Options and ofTileable = ofTileable then ...
```

Las **máscaras** se pueden utilizar de manera similar a los bits individuales. Por ejemplo, para ver si un registro de evento llamada *Evento* contiene algún evento de ratón se escribiría:

```
if Evento.What and evMouse <> 0 then ...
```

## 14.5 LAS VISTAS EN TURBO VISION

Las vistas son objetos que representan regiones rectangulares de pantalla, y constituyen el único medio de visualizar información a los usuarios en las aplicaciones Turbo Vision.

Una vista es un objeto que gestiona un área de pantalla rectangular. Por ejemplo, la barra de menús de la parte superior de la pantalla es una vista. Cualquier acción del programa en ese área de pantalla (por ejemplo, pinchando la barra de menús con el ratón) será tratado por la vista que controla ese área.

Hay tres tareas fundamentales que toda vista debe realizar:

- *Gestionar una región rectangular.*
- *Dibujarse a sí misma bajo petición.*
- *Manejar los eventos que ocurran dentro de sus límites.*

A veces para una vista la forma más fácil de gestionar su área es delegando ciertas partes del trabajo en otras vistas, conocidas como *subvistas*. Una vista que tiene subvistas se denomina *grupo*. Cualquier vista puede ser una subvista, pero los grupos deben ser descendientes de *TGroup*, el cual es a su vez descendiente de *TView*. Un grupo con subvistas se dice que *posee* a las subvistas, dado que gestiona esas subvistas. Cada subvista se dice que tiene una *vista propietaria*, que es el grupo que la posee.

## CONSTRUCCION DE OBJETOS VISTA

Por convenio, todos los constructores de objetos de Turbo Vision se denominan *Init*. El constructor *Init* de *TView* tiene un único parámetro, el rectángulo límite de la vista:

```
constructor TView.Init(var Bounds: TRect);
```

Antes de hacer ninguna otra cosa, *TView.Init* invoca al constructor *Init* heredado de *TObject*, el cual rellena todos los campos de la vista con ceros. Dado que todos los constructores de vistas terminan por invocar a *TObject.Init*, es preciso asegurarse de no inicializar ningún campo antes de invocar al constructor heredado.

*Init* toma el parámetro *Bounds* que se le pasa y establece dos importantes campos basándose en él: *Origin* y *Size*. *Origin* es la esquina superior izquierda del rectángulo límite. *Size* contiene la anchura y altura del rectángulo. Cada uno de estos puntos está representado en el objeto mediante un campo de tipo *TPoint*. *Origin* es un punto del sistema de coordenadas de la vista propietaria.

## GESTION DE LIMITES DE UNA VISTA

Una vez se ha construido la vista, hay numerosos métodos para manipular los límites de la misma.

## LAS VISTAS EN TURBO VISION

Para obtener las coordenadas de la vista se utiliza el método *GetExtent* que lleva un sólo parámetro **var** de tipo *TRect* y devuelve en dicho rectángulo los límites de la vista. El rectángulo dado por *GetExtent* siempre tiene en su campo *A* el punto (0,0), y en *B* el tamaño de la vista. En otras palabras, *GetExtent* devuelve las coordenadas de la vista en su propio sistema de coordenadas local.

Para obtener las coordenadas de la vista relativas a su vista propietaria, se usa el método *GetBounds* en vez de *GetExtent*. *GetBounds* devuelve las coordenadas de la vista en el sistema de coordenadas de la vista propietaria, dando al campo *A* de su parámetro el valor del campo *Origin* de la vista, y a *B* el del tamaño de la vista desplazado desde el origen.

Para cambiar la posición de una vista sin afectar a su tamaño, se invoca al método *MoveTo* de la vista. *MoveTo* lleva dos parámetros, las coordenadas X e Y del nuevo origen de la vista. En el ejemplo 14.4 se define el método *InitStatusLine* que inicializa la línea de estado de la aplicación *TGestionFichas*. Una vez construido el objeto *StatusLine* se obtienen las dimensiones de la vista aplicación (que en este caso son las de la pantalla) y se mueve la línea de estado a la parte inferior de la vista aplicación, es decir, se pone el origen de *StatusLine* igual al punto (0, R.B.Y - 1).

### Ejemplo 14.4.

```
procedure TGestionFichas.InitStatusLine;
var
 R: TRect;
begin
 . . .
 GetExtent(R);
 StatusLine^.MoveTo(0, R.B.Y - 1);
end;
```

Para cambiar el tamaño de una vista sin moverla (es decir, sin cambiar el origen), se invoca al método *GrowTo* de la vista. *GrowTo* lleva dos parámetros, que determinan las coordenadas X e Y de la esquina inferior derecha de la vista, relativas al origen.

Por ejemplo, el código siguiente hace que una vista doble tanto su anchura como su altura:

```
GrowTo(Size.X, Size.Y);
```

Para establecer el tamaño y posición de una vista en un único paso, se invoca al método *Locate* de la vista. *Locate* lleva un rectángulo como único parámetro, el cual pasa a ser los límites de la vista. Normalmente se utiliza el método *Assign* de *TRect* para definir el rectángulo que dará dimensiones a la ventana.

*Grow* es un método de *TRect* que incrementa (o con parámetros negativos, decrementa) los tamaños horizontal y vertical de un rectángulo. Usado conjuntamente con el método *GetExtent* de una vista, *Grow* hace fácil ajustar una vista dentro de otra:

```
GetExtent(R); { obtiene límites de la vista en R }
R.Grow(-1, -1); { encoge el rectángulo en 1 por cada lado }
```

## MANEJO DE LOS CAMPOS DE OPCIONES DE UNA VISTA

Toda vista hereda cuatro campos de *TView* que contienen información bitmap. Es decir, cada bit de cada campo tiene un significado especial, estableciendo alguna opción en la vista.

*Options* es una palabra bitmap que existe en toda vista. Varios descendientes de *TView* tienen un valor diferente por defecto para *Options*. Los flags *GrowMode* y *DragMode*, aunque están presentes en todas las vistas, no tienen efecto hasta que se inserta la vista en un grupo y se convierte en una subvista del grupo, así que se explican en la parte de este capítulo dedicada a las subvistas. El cuarto campo, *EventMask*, se describe en el apartado de **Eventos en Turbo Vision**.

*Options* tiene tres bits que gobiernan la selección de la vista por el usuario: *ofSelectable*, *ofTopSelect*, y *ofFirstClick*.

La mayoría de las vistas tienen activado por defecto *ofSelectable*, lo que significa que el usuario puede seleccionar la vista con el ratón. Si la vista está en un grupo, el usuario también puede seleccionarla con la tecla **Tab**. Se podría querer que el usuario no seleccionase vistas puramente informativas, luego se desactivarían sus bits *ofSelectable*. Los objetos texto estático y marco de ventana, por ejemplo, no son seleccionables por defecto.

El bit *ofTopSelect*, si está activado, hace que la ventana se mueva al primer plano de las subvistas del propietario cuando es seleccionada. Esta opción está principalmente diseñada para las ventanas del *desktop*, así que no debe usarse para las vistas de un grupo.

El bit *ofFirstClick* controla si el click de ratón que selecciona la vista es también pasado a la vista para ser procesado. Por ejemplo, si el usuario hace click en un botón, se querrá seleccionar el botón y pulsarlo con un solo click, luego los botones tienen activado *ofFirstClick* por defecto. Pero si el usuario pincha una ventana inactiva, probablemente sólo se quiera seleccionar la ventana y no procesar el click como una acción sobre la ventana una vez está activada. Esto hace menos probable que un usuario cierre una ventana o haga un zoom cuando lo único que intentaba era activarla.

Si se activa el bit *ofFramed*, la vista tiene un marco visible a su alrededor. No afecta al marco de los objetos ventana y cuadro de diálogo. Esas son vistas aparte controladas por un campo del objeto ventana. El bit *ofFramed* afecta sólo a vistas insertadas en ventanas o cuadros de diálogo.

Los bits *ofPreProcess* y *ofPostProcess* permiten a una vista procesar eventos enfocados antes o después de que la vista que tiene el foco los vea. Se verán en el apartado de **Eventos en Turbo Vision**.

Las vistas tienen dos bits que controlan el centrado de la vista dentro de su propietaria. El bit *ofCenterX* centra la vista horizontalmente, y *ofCenterY* la centra verticalmente. Si se quiere centrar tanto horizontal como verticalmente, se puede utilizar la máscara *ofCentered*, que contiene ambos bits de centrado.

## ESTADO DE UNA VISTA

El campo bitmap de tipo *Word* denominado *State* contiene información sobre el estado de la vista. Al contrario que los flags de opciones y bits de modo, que se activan cuando se construye una vista (si una ventana es redimensionable, *siempre* es redimensionable), los flags de estado cambian durante la vida de una vista a medida que cambia el estado de la vista. La información de estado incluye si la vista es visible, tiene cursor o sombra, está siendo arrastrada, o tiene el foco de entrada.

En la mayoría de los casos, no se necesita cambiar manualmente los bits de estado, ya que los cambios de estado más comunes son manejados por algún método. Por ejemplo, el bit *sfCursorVis* controla si la vista tiene un cursor de texto visible. En lugar de manipular ese bit directamente, se puede invocar a *ShowCursor* o *HideCursor*, que se ocupan de cambiar el bit *sfCursorVis*. La Tabla 14.4 muestra los flags de estado y los métodos que los manipulan.

| Flag de estado                                         | Método                    |
|--------------------------------------------------------|---------------------------|
| <i>sfVisible</i>                                       | Show, Hide                |
| <i>sfCursorVis</i>                                     | ShowCursor, HideCursor    |
| <i>sfCursorIns</i>                                     | BlockCursor, NormalCursor |
| <i>sfShadow</i>                                        | Ninguno                   |
| <i>sfActive</i> , <i>sfSelected</i> , <i>sfFocused</i> | Select                    |
| <i>sfDragging</i>                                      | DragView                  |
| <i>sfModal</i>                                         | Execute                   |
| <i>sfExposed</i>                                       | TGroup.Insert             |

Tabla 14.4 Métodos para el cambio de flags de estado

Para cambiar un flag de estado que no tenga un método específico dedicado, hay que invocar al método *SetState* de la vista, pasándole dos parámetros: el bit a cambiar, y un flag Boolean indicando si se desea activar el bit. Por ejemplo, para activar el flag *sfShadow*, se haría lo siguiente:

```
SetState(sfShadow, True);
```

*SetState* es llamado también siempre que una vista obtiene el foco, cede el foco, o se selecciona, para cambiar los flags de estado pertinentes. Pero el cambio de flags de estado a menudo requiere que la vista haga algunos otros cambios en respuesta al nuevo estado, como por ejemplo redibujar la vista. Si se quiere que una vista responda de alguna forma especial a un cambio de estado, hay que redefinir *SetState*, invocando al método *SetState* heredado para asegurarse de que el cambio sucede, y respondiendo luego al nuevo estado.

## MANEJO DEL CURSOR

El cursor proporciona al usuario una indicación visual de a dónde irá la entrada del teclado, pero es cuestión del programador asegurarse de que el programa relaciona efectivamente la posición del cursor con la de la entrada.

*TView* tiene un campo denominado *Cursor*, de tipo *TPoint*, que indica la posición del cursor dentro de la vista, relativa al origen de la vista. Las vistas tienen varios métodos dedicados al manejo del cursor, los cuales permiten hacer lo siguiente:

**Mostrar u ocultar el cursor.** Las vistas tienen dos métodos, *ShowCursor* y *HideCursor*, que muestran y ocultan el cursor de texto, respectivamente. Por defecto, el cursor está oculto, aunque algunos descendientes de *TView* (sobre todo los campos de edición y editores) redefinen este comportamiento y muestran sus cursores por defecto.

Uno de los bits del campo *State* de una vista (*SfCursorVis*) controla si la vista tiene un cursor visible. *ShowCursor* y *HideCursor* activan y desactivan el bit *SfCursorVis*. Cuando la vista obtiene el foco de entrada, Turbo Vision muestra el cursor en la posición indicada por *Cursor* si *SfCursorVis* está activado.

**Cambiar el estilo del cursor.** Existen dos estilos de cursor de texto: un carácter de subrayado y un bloque macizo. Los métodos *NormalCursor* y *BlockCursor* de *TView* establecen el estilo del cursor en subrayado o bloque, respectivamente. Generalmente un estilo indica un modo de inserción, y el otro un modo de sobrescritura. Por defecto, el estilo del cursor es normal, o subrayado. El bit *SfCursorIns* de la campo *State* de la vista controla qué estilo de cursor utiliza la vista. *BlockCursor* y *NormalCursor* activan y desactivan el bit *SfCursorIns*.

**Mover el cursor.** Para cambiar la posición del cursor de texto en una vista, se invoca al método *SetCursor*. Este método lleva dos parámetros, que representan las coordenadas  $x$  e  $y$  de la nueva posición del cursor, relativas al origen de la vista. Se debe evitar la modificación del campo *Cursor* directamente. En su lugar, debe usarse *SetCursor*, el cual cambia la posición del cursor y además actualiza la pantalla.

## VISUALIZACION DE UNA VISTA

La apariencia de un objeto vista está determinada por su método *Draw*. Casi todos los nuevos tipos de vista necesitarán tener su propio *Draw*, puesto que es, generalmente, la apariencia de una vista lo que la distingue de otras vistas.

Hay un par de reglas que atañen a todas las vistas con respecto a su apariencia:

- 1ª *Cubrir completamente el área de la que es responsable.*
- 2ª *Ser capaz de dibujarse a sí misma en cualquier momento.*

La escritura de métodos *Draw* implica tres tareas:

**Selección de colores.** Cuando se escriben datos en la pantalla, no se especifica directamente el color de un elemento, sino que se utiliza un índice a la paleta de colores de la vista. Así, por ejemplo, si una vista tiene dos clases de texto, normal y resaltado, su paleta probablemente tendrá dos entradas, una para texto normal y otra para texto resaltado. En el método *Draw*, se le pasará a *GetColor* el índice apropiado según el atributo que se quiera.

**Escritura directa en la vista.** Las vistas tienen dos métodos similares para escribir en ellas caracteres (*WriteChar*) y cadenas (*WriteStr*). En cada caso, se especifican las coordenadas de la vista en donde comenzaría el texto, el texto a visualizar, y el índice de paleta del color del texto. Además para *WriteChar* se añade un quinto parámetro que indica el número de caracteres consecutivos a escribir.

**Escritura a través de buffers.** La manera más eficiente de gestionar la tarea de dibujar vistas grandes o complejas es escribir el texto sobre un buffer<sup>56</sup>, y después visualizar el buffer de una sola vez. El uso de buffers mejora la velocidad de dibujo, y reduce el parpadeo causado por el gran número de escrituras individuales en la pantalla. Generalmente se utilizará el buffer para escribir líneas o vistas enteras de una sola vez.

El tipo *TDrawBuffer*, definido en la unit *Views*, proporciona un array de palabras conveniente que se puede utilizar para dibujar buffers.

El dibujo de un buffer conlleva tres pasos para cada uno de los cuales se utilizan unos procedimientos determinados:

1. *Establecimiento del color del texto.*

(*GetColor*)

2. *Movimiento del texto al buffer.*

(*MoveBuf, MoveChar, MoveCStr* O *MoveStr*)

3. *Escritura del buffer en la pantalla.*

(*WriteBuf* O *WriteLine*)

El ejemplo 14.5 implementa el método *Draw* del tipo objeto *TVistaContador* que se utiliza para presentar el n° de fichas en la agenda del programa AGENDA. Este tipo está definido en la Unit *CONTADOR.TPU*.

#### Ejemplo 14.5.

```

procedure TVistaContador.Draw;
var
 B: TDrawBuffer;
 C: Word;
 Params: array[0..1] of Longint;
 Display: String[20];
begin
 C := GetColor(2); { Usa el color del marco de ventana }

```

---

<sup>56</sup> Un *buffer de dibujo* es un array de palabras, donde cada palabra representa un carácter y su atributo de color, que es la misma manera que tiene la pantalla de vídeo de representar cada carácter.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
MoveChar(B, '=', C, Size.X);
Params[0] := Activo;
Params[1] := Contador;
FormatStr(Display, '~%d~ de %d ', Params);
{ Si Activo mayor que Contador, resaltar Activo }
if Activo > Contador then C := GetColor($0504)
else C := GetColor($0202);
MoveCStr(B, Display, C);
WriteLine(0, 0, Size.X, Length(Display), B);
end;
```

Este procedimiento obtiene el color del marco (`GetColor(2)`) de la ventana *TVentanaFicha*, que es la vista propietaria de *TVistaContador* e inicializa el buffer de dibujo `B` con tantos caracteres '=' como el ancho de la vista *TVistaContador*, de forma que al insertar en el texto se mantenga el marco de *TVentanaFicha* donde no se escriban caracteres. En el string `Display` se introduce una cadena que indica el n° del elemento activo dentro del total de fichas de la agenda utilizando el procedimiento `FormatStr`. Este string se lleva al buffer `B` en el color del marco a no ser que el elemento activo sea mayor que el n° de elementos indicado en el contador. Si esto ocurriese se lleva el string al buffer en el color del marco excepto los caracteres que indican el n° del elemento activo, que irían resaltados. Para copiar el string sobre el buffer se utiliza el procedimiento `MoveCStr`. Por último el buffer es escrito en la pantalla mediante la llamada a `WriteLine`.

## VALIDACION DE UNA VISTA

Cada vista tiene un método virtual denominado *Valid* que lleva como único parámetro una constante de comando y devuelve un valor Boolean. En general, invocar a *Valid* es una manera de interrogar a la vista para saber si está lista para recibir un determinado comando. Si *Valid* devuelve *True*, indica que ese comando es válido.

*Valid* se utiliza para tres clases diferentes de validación, aunque se puede redefinir para desarrollar otros tipos de operaciones:

*Comprobación de construcción correcta.* El comando *cmValid*, se usa para asegurarse de que las vistas se construyen correctamente. El método *ValidView* del objeto aplicación invoca al método *Valid* de una vista, pasándole como parámetro *cmValid*. Las vistas deberían responder a tales llamadas asegurando que cualquier cosa hecha durante la construcción, como por ejemplo la reserva de memoria, tuvo éxito.

*Comprobación de cierre seguro.* El momento más común para comprobar *Valid*, aparte de la comprobación de una construcción correcta de la vista, es cuando se cierra una vista. Por ejemplo, cuando se invoca al método *Close* de un objeto ventana, este invoca a *Valid*, pasándole *cmClose*, para asegurarse de que es seguro cerrar la ventana. Si *Valid* devuelve *False*, la vista no se cerraría.



## LOS GRUPOS EN TURBO VISION

Cuando se escribe métodos *Valid*, si se detecta una razón para que la vista no sea correcta hay dos opciones: que *Valid* devuelva *False*, o llevar a cabo alguna acción que haga válida a la vista y devolver después *True*. Por ejemplo, cuando un editor de ficheros con cambios sin guardar es validado para cerrarse, despliega un cuadro de diálogo que pregunta al usuario si desea guardar los cambios. El usuario entonces tiene tres opciones: guardar los cambios y cerrar, abandonar los cambios y cerrar, o no cerrar. En los dos primeros casos, *Valid* devuelve *True*, en el tercero, *False*.

*Validación de datos.* Las vistas campo de edición pueden usar *Valid* para determinar si los contenidos de la cadena de texto contienen valores legales mediante su comprobación con objetos de validación. La validación de datos puede tener lugar cuando el usuario cierra una ventana, pero se puede utilizar exactamente el mismo mecanismo para validar en cualquier otro momento.

Por ejemplo, los objetos campo de edición comprueban la validez de sus contenidos cuando se invoca a *Valid* con el comando *cmClose*. Bastaría con invocar a *Valid(cmClose)* después de cada pulsación de tecla para comprobar la entrada según el usuario la va tecleando.

## 14.6 LOS GRUPOS EN TURBO VISION

Un *grupo* es un cuadro vacío que contiene y gestiona otras vistas. Técnicamente es una vista, y por la tanto responsable de todo aquello que una vista debe ser capaz de hacer: gestionar un área de pantalla rectangular, representarse visualmente a sí misma en cualquier momento, y manejar eventos en su región de pantalla. La diferencia está en cómo lleva a cabo estas tareas mediante la utilización de subvistas.



Fig. 14.27 Desktop con dos subvistas visores de texto

Un grupo sirve para contener otras vistas. Se puede decir que un grupo es una vista compuesta. En lugar de manejar todas sus responsabilidades por sí misma, divide sus deberes entre varias subvistas. Una **subvista** es una vista que pertenece a otra vista, y el grupo que la posee se denomina *vista propietaria*.

TApplication es un ejemplo de vista propietaria que en este caso en concreto controla la totalidad de la pantalla. TApplication es un grupo que posee tres subvistas: la barra de menús, el *desktop*, y la línea de estado. La aplicación delega una región de la pantalla en cada una de estas subvistas. La barra de menús obtiene la línea superior, la línea de estado obtiene la línea inferior, y el *desktop* obtiene todas las líneas de en medio. La figura 14.27 muestra una pantalla típica de TApplication, que en este caso tiene además de las tres subvistas anteriores dos subvistas visores de ficheros de texto dentro del *desktop*.

### ARBOL DE VISTAS

Cuando se insertan subvistas en un grupo, las vistas forman una especie de *árbol de vistas*, con la propietaria como *tronco* y las subvistas como *ramas*. Los enlaces de pertenencia de todas las vistas de una aplicación compleja pueden llegar a ser bastante complicados, pero si se plantean como una jerarquía, se puede comprender la estructura global.

Así por ejemplo, el objeto aplicación posee tres subvistas, como se muestra en la figura 14.27. El *desktop* a su vez es un propietario con tres subvistas: el fondo y dos visores de texto. Cada ventana visor posee a su vez subvistas: un marco, un desplazador (la vista interior que contiene un array de texto con capacidad de *scroll*), y un par de barras de desplazamiento. El árbol de vistas correspondiente sería el que se presenta en la figura 14.28.

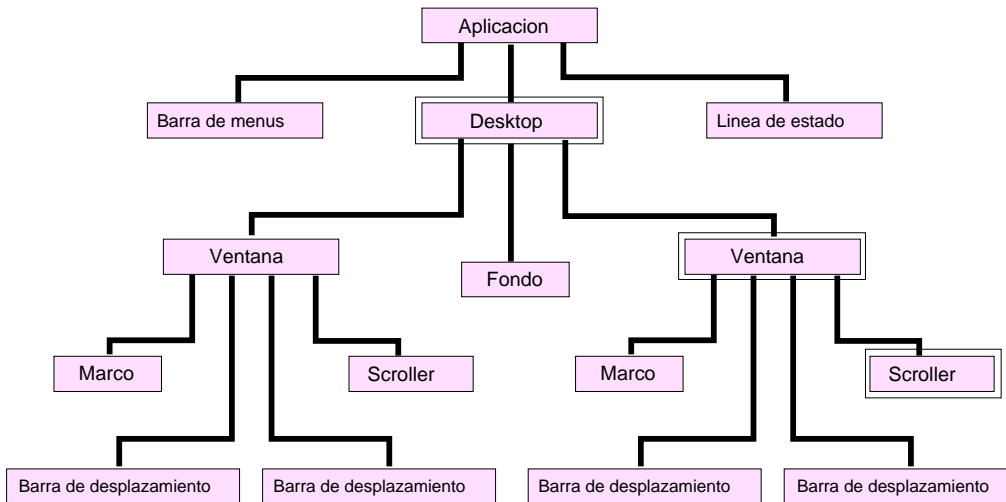


Fig. 14.28 Arbol de vistas

## LOS GRUPOS EN TURBO VISION

Si el usuario pincha en el icono de cierre del visor de fichero activo o en un elemento de menú *Close Window*, éste se cerrará. Turbo Vision entonces lo saca del árbol de vistas y lo libera. La ventana liberará todas sus subvistas, y después se liberará a sí misma.

### EL ORDEN-Z

Los grupos recuerdan el orden en que son insertadas las subvistas. Ese orden se denomina *orden-Z*. El término *orden-Z* se refiere al hecho de que las subvistas tienen una relación espacial tridimensional. Toda vista tiene una posición y tamaño dentro del plano de la vista tal como se ve (las dimensiones X e Y), determinados por sus campos *Origin* y *Size*. Pero las vistas y las subvistas se pueden solapar, y para que Turbo Vision sepa si una vista es ocultada (parcial o totalmente) por otras hay que añadir una tercera dimensión, la *dimensión-Z*.

El orden-Z es el inverso al orden de inserción, es decir, el orden en que se encuentran las vistas empezando por la más cercana y moviéndose hacia el *interior* de la pantalla. La última vista insertada es la vista "delantera". Hay que pensar en el orden-X como si se fuese de izquierda a derecha, el orden-Y de arriba a abajo, y el orden-Z de delante a atrás.

Todo grupo puede ser considerado como una superposición de vistas, como se ilustra en la figura 14.29.

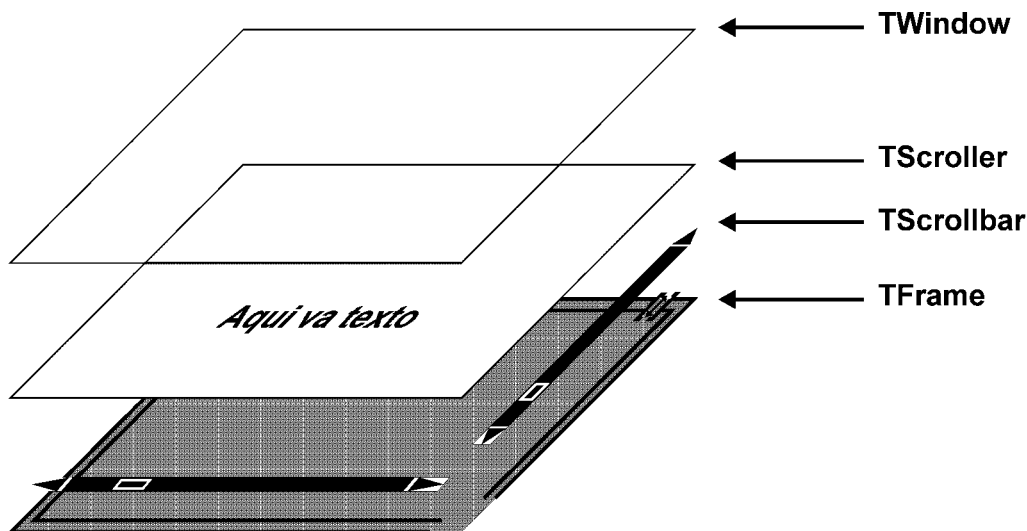


Fig. 14.29 Vista lateral de un visor de texto

La ventana (*TWindow*) en sí es simplemente un cristal que cubre un grupo de vistas. Dado que todo lo que se ve en la pantalla es una proyección plana de las vistas que hay detrás del cristal, no se puede apreciar qué vistas están por delante de otras a no ser que se solapen.

Por defecto, una ventana tiene un marco (*TFrame*), el cual es insertado antes que ninguna otra subvista. Es por lo tanto la vista fondo. Al crear un interior con *scroll*, dos barras de desplazamiento (*TScrollbar*) se superponen al marco. Viendo de frente la escena completa, parecen parte del marco, pero de lado, se puede comprobar que realmente *flotan* sobre el marco, ocultando parte del marco de la vista.

Finalmente, es insertado el propio desplazador, cubriendo todo el área dentro del borde del marco. El texto se escribe en el desplazador, no en la ventana, pero se puede ver cuando se mira a través de la ventana.

A mayor escala, se puede ver en la figura 14.30 el *desktop* como un cristal mayor, que cubre un volumen mayor donde se superponen más ventanas, muchas de las cuales a su vez son el resultado de la superposición de otras ventanas.

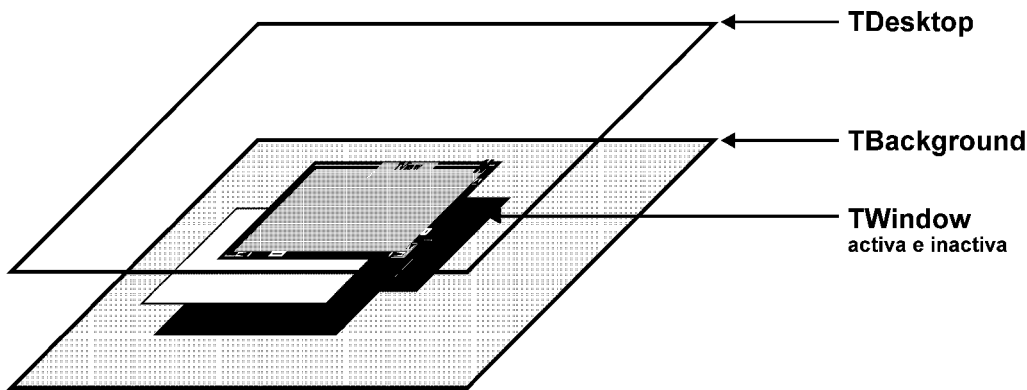


Fig. 14.30 Vista lateral del *desktop*

Al igual que antes, el grupo (*desktop*) es un cristal. Su primera subvista es el fondo (un objeto *TBackground*), luego esa vista está detrás del resto. Sobre él hay dos ventanas con *scroll* interior.

## GESTION DE SUBVISTAS

En este apartado vamos a estudiar brevemente las operaciones que se pueden realizar para insertar, manipular y destruir una subvista.

Para enlazar una subvista a una propietaria, se *inserta* la subvista en la propietaria utilizando un método denominado *Insert*. La propietaria trata a sus subvistas como una lista enlazada de vistas, guardando un puntero a una sola subvista, y utilizando el campo *Next* de cada subvista para apuntar a la siguiente.

Como mínimo, las subvistas de un grupo deben cubrir el área entera dentro de los límites del grupo. Hay dos maneras de gestionar esto:

**Dividiendo el grupo.** Se utiliza en casos en que las subvistas no se solapan, como en la aplicación o en una ventana dividida en paneles separados.


**Proporcionando un fondo.** Se usa en casos en que las subvistas necesitan solaparse y moverse, como en el *desktop*, o donde las subvistas importantes están separadas, como los controles de un cuadro de diálogo. No hay ninguna razón para que las vistas no puedan solaparse. Una de las grandes ventajas de un entorno de ventanas es la habilidad de tener ventanas múltiples solapándose en el *desktop*. Los grupos (incluso el *desktop*) saben cómo manejar subvistas solapadas.

La función de un fondo es asegurar que *algo* es dibujado sobre el área entera del grupo, dejando que otras subvistas cubran tan solo el área particular que necesiten. Un ejemplo claro es el *desktop*, el cual proporciona un fondo de color gris (por defecto) por detrás de las ventanas. Si una ventana o grupo cubre por completo el *desktop*, el fondo está oculto, pero si al mover o cerrar ventanas se descubre alguna parte del *desktop*, el fondo asegura que hay algo ahí dibujado.

Dentro de cada grupo de vistas, una y sola una subvista esta *seleccionada*. Por ejemplo, cuando la aplicación construye su barra de menús, *desktop*, y línea de estado, el *desktop* es la vista seleccionada, porque es ahí donde tendrá lugar más adelante el trabajo.

Cuando se tienen abiertas varias ventanas en el *desktop*, la ventana seleccionada es aquella donde se está trabajando actualmente. También se la denomina ventana *activa* (normalmente la ventana superior).

Dentro de la ventana activa, la subvista seleccionada se denomina **vista enfocada**, que es donde la acción tendrá lugar. En la aplicación esquematizada en la figura 14.27, *Aplicacion* es la vista modal, y *DeskTop* su vista seleccionada. Dentro del *desktop*, la segunda ventana (la más recientemente insertada) está seleccionada, y por lo tanto activa. Dentro de esa ventana, el *scroll* interior está seleccionado, y puesto que es una vista terminal (es decir, no es un grupo), es el final de la cadena, la vista enfocada. La figura 14.28 representa la *cadena de vistas enfocadas* o *cadena de foco* resaltada mediante cuadros con línea doble.

La vista actualmente enfocada generalmente está resaltada de alguna manera en la pantalla. Por ejemplo, si hay abiertas varias ventanas en el *desktop*, la ventana activa es la que tiene un marco con línea doble. Los marcos del resto son de línea única<sup>57</sup>. Dentro de un cuadro de diálogo, el control enfocado es más brillante que los otros, indicando que es sobre el que se actúa si se pulsa . El control enfocado es por tanto el control por defecto también.

Una vista puede **obtener el foco** de dos formas:

*Foco por defecto.* Cuando se crea un grupo de vistas, la vista propietaria especifica cuál de sus subvistas será enfocada invocando el método *Select* de esa subvista.

---

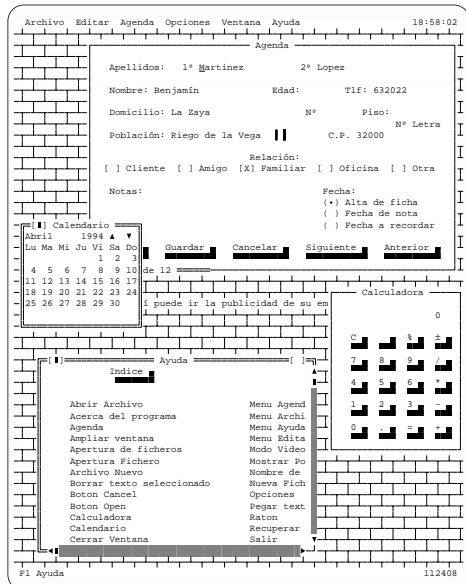
<sup>57</sup> En monitores monocromo, Turbo Vision incluye caracteres de flecha para indicar el foco.

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

*Una acción del usuario.* El usuario generalmente determina qué vista tiene actualmente el foco seleccionando una concreta. Por ejemplo, si la aplicación tiene varias ventanas abiertas en el *desktop*, el usuario puede seleccionar una simplemente pinchando en ella. En un cuadro de diálogo, el usuario puede mover el foco entre vistas pulsando **Tab**, con lo cual se recorren cíclicamente todas las vistas seleccionables, pinchando en una vista particular, o pulsando una hot key.

• Redimensionar una subvista

El campo *GrowMode* de una vista determina cómo cambia la vista cuando su grupo propietario es redimensionado. Los bits individuales de *GrowMode* permiten anclar un lado de la vista a su propietaria, de forma que al redimensionar la propietaria se mueve y/o redimensiona la subvista, basándose en su modo de crecimiento.



```
cmOpcionesVideo:
begin
 SetScreenMode(ScreenMode xor smFont8x8);
 . . .
```

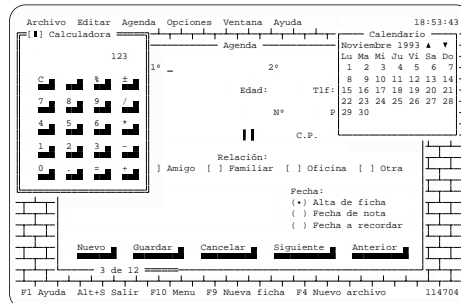


Fig. 14.31 Modo de vídeo de 50 y 25 líneas con una tarjeta gráfica VGA

El bit *gfGrowLoX* ancla el lado izquierdo de la vista al lado izquierdo de su propietaria, lo que significa que la vista mantiene una distancia constante al lado izquierdo de su propietaria. Los bits *gfGrowLoY*, *gfGrowHiX*, y *gfGrowHiY* anclan el lado superior, derecho, e inferior de la vista a los lados correspondientes de la propietaria. La máscara *gfGrowAll* ancla los cuatro lados, redimensionando la vista a medida que la esquina inferior derecha de la propietaria se mueve. Los interiores de ventana a menudo utilizan *gfGrowAll* para mantenerse adecuadamente dimensionados dentro de sus marcos.

## LOS GRUPOS EN TURBO VISION

El flag *gfGrowRel* está destinado para usarse sólo con las ventanas del *desktop*. Activando *gfGrowRel* las ventanas retienen sus tamaños relativos cuando el usuario cambia la aplicación entre dos modos de vídeo diferentes. Para cambiar el modo de vídeo se utiliza la función *setscreenMode*. En el fragmento de programa siguiente se utiliza para conmutar entre el modo de vídeo de 25 líneas y el de 40 o 50 líneas (según el tipo de tarjeta gráfica que disponga el ordenador). El modo por defecto es de 25 líneas. La figura 14.31 presenta una pantalla de la ejecución del programa AGENDA en un ordenador con tarjeta VGA en modo de 50 y 25 líneas en pantalla.

### • Dibujar una subvista

Los grupos son la excepción a la regla de que las vistas deben saber cómo dibujarse a sí mismas. Un *TGroup* les dice a sus subvistas que se dibujen. El efecto acumulado del dibujo de subvistas deberá cubrir el área total asignada al grupo.

El grupo llama a dibujarse a sus subvistas en orden-Z, lo que significa que la última subvista insertada en el grupo es la primera en dibujarse. Si las subvistas se solapan, la más recientemente insertada estará delante del resto.

Todas las vistas tienen un bit en su palabra *Options* denominado *ofBuffered*, pero sólo los grupos hacen uso de él. Cuando este bit está activado, los grupos pueden acelerar su salida a pantalla escribiendo en un buffer caché. Por defecto, todos los grupos tienen activado *ofBuffered* y lo utilizan para dibujarse. Copiar la imagen existente en el buffer es mucho más rápido que regenerar la imagen.

El gestor de memoria de Turbo Vision libera estos buffers de grupo siempre que otras reservas de memoria necesiten ese espacio. No se pierde ninguna información cuando el buffer es liberado, pero el grupo tendrá que redibujarse llamando a sus subvistas la próxima vez que lo necesite.

También se puede forzar a un grupo a dibujarse completamente sin copiar su buffer invocando a su método *Redraw*.

Invocando al método *Lock* de un grupo se detienen todas las escrituras a la pantalla del grupo hasta la correspondiente llamada al método *Unlock*. Cuando es invocado *Unlock*, el buffer del grupo se escribe en la pantalla. Los bloqueos pueden disminuir considerablemente el parpadeo durante actualizaciones de pantalla complicadas. Por ejemplo, el *desktop* se bloquea mientras realiza distribuciones mosaicos o cascadas de subvistas.

Cuando se dibujan las subvistas de un grupo, se recortan automáticamente al alcanzar los bordes del mismo. Para hallar el área que requiere redibujarse (es decir, la parte de la vista que no está recortada), se invoca al método *GetClipRect*.

### • Borrar, iterar y buscar una subvista

Una vez insertada una subvista en un grupo, el grupo la gestiona prácticamente por completo, asegurándose de que es dibujada, movida, etcétera. Cuando se libera un objeto grupo, éste libera

automáticamente todas sus subvistas, para así no tener que liberarlas individualmente. Así, por ejemplo, aunque el constructor de un cuadro de diálogo es a menudo bastante largo y complicado, construyendo e inicializando numerosos controles como subvistas, el destructor a menudo ni siquiera es redefinido, como hace el objeto cuadro de diálogo por defecto que utiliza el destructor *Done* que hereda de *TGroup*, el cual libera cada subvista antes de liberarse a sí mismo.

Aparte del manejo automático de subvistas, a veces se necesitará desarrollar las siguientes tareas sobre las subvistas de un grupo:

*Borrar subvistas.* Cuando se desea eliminar una subvista mientras se sigue usando el grupo se utiliza el método *Delete* del propietario. *Delete* es el inverso de *Insert*: elimina la subvista de la lista de subvistas del propietario, pero no libera la vista borrada.

*Iteración de subvistas.* Los grupos manejan varios de sus deberes, como su dibujo, invocando a todas sus subvistas en orden-Z. El proceso de llamada a cada subvista por orden se denomina *iteración*. Además de las iteraciones estándar, *TGroup* proporciona métodos iteradores que permiten definir acciones propias a desarrollar por o sobre cada subvista.

*Búsqueda de una subvista particular.* Para localizar una subvista dentro de un grupo se utiliza el método *FirstThat* de *TGroup*. Este método lleva como parámetro un puntero a una función Boolean y aplica esa función a cada de las subvistas del grupo en orden-Z hasta que la función devuelve *True*, momento en el que *FirstThat* devuelve un puntero a dicha subvista.

#### • Mover una subvista

Una manera de mover una vista es dejar que el usuario la posicione o redimensione con un ratón. El movimiento de una ventana con el ratón se denomina *arrastre*. Cada vista tiene un campo bitmap denominado *DragMode* que proporciona los límites por defecto en los que el usuario puede arrastrar la vista.

Los bits de *DragMode* determinan si partes de la vista se pueden mover fuera de su propietaria. Cuando se arrastra algunas vistas, como las ventanas en el *desktop*, cuando se mueve la vista más allá de los límites de la propietaria se recorta la subvista a partir de esos límites. Los bits cuyos nombres comienzan con *dmLimit* restringen el arrastre de una vista fuera de su propietaria.

La máscara *dmLimitAll* contiene todos los bits de modo de arrastre. Activando *dmLimitAll* en una vista se consigue que el usuario no sea capaz de arrastrar ninguna parte de la vista fuera de su propietaria. Los bits individuales *dmLimitLoX*, *dmLimitLoY*, *dmLimitHiX*, y *dmLimitHiY* restringen el arrastre más allá de los límites izquierdo, superior, derecho, e inferior de la propietaria, respectivamente. Por defecto, las vistas tienen activado *dmLimitLoY*, lo que significa que el usuario no puede arrastrar la parte superior de una vista más allá de la parte superior de su propietaria.

El arrastre efectivo de la vista es manejado por un método denominado *DragView*.



## GRUPOS MODALES

Los programas más complejos tienen distintos *modos* de operación, entendiéndose por modos distintas formas de funcionamiento.

Casi cualquier vista de Turbo Vision puede definir un modo de operación, en cuyo caso es denominada una *vista modal*, pero las vistas modales casi siempre son grupos. El ejemplo clásico de vista modal es un cuadro de diálogo. Generalmente, cuando un cuadro de diálogo está activo, no funciona nada fuera de él. No se pueden usar los menús u otros controles que no pertenezcan al cuadro de diálogo. Además, pinchar con el ratón fuera del cuadro de diálogo no tiene efecto. El cuadro de diálogo tiene el control del programa hasta que el usuario lo cierre.

Cuando se hace modal una vista, sólo esa vista y sus subvistas pueden interactuar con el usuario. Cualquier parte del árbol de vistas que no sea la vista modal o no pertenezca a ella está inactiva. Los eventos sólo son gestionados por la vista modal y sus subvistas.

Hay una excepción a esta regla: *La línea de estado está disponible en todo momento*. De esa manera se pueden tener elementos de la línea de estado activos, incluso cuando el programa esté ejecutando un cuadro de diálogo modal que no posea la línea de estado. Los eventos y comandos generados por la línea de estado, sin embargo, serán manejados como si hubiesen sido generados dentro de la vista modal.

La forma de ejecutar cuadros de diálogo modales en el *desktop*, es por medio del método *ExecuteDialog*. En un caso más general, se puede convertir a un grupo en la vista modal actual *ejecutándolo*; es decir, invocando a su método *Execute*. *TGroup.Execute* implementa un bucle de eventos, interactuando con el usuario y despachando eventos a las subvistas apropiadas. En la mayoría de los casos, no se invocará directamente a *Execute*, sino que en su lugar se utilizará *ExecView*<sup>58</sup>.

Cualquier vista puede poner fin al estado modal en curso invocando al método *EndModal*.

### 14.7 EVENTOS EN TURBO VISION

El propósito de Turbo Vision® es proporcionar un marco de trabajo para las aplicaciones de forma que el programador se pueda centrar en crear la funcionalidad (modelo) de esas aplicaciones. Las dos herramientas principales de Turbo Vision son el soporte de ventanas incorporado (vista) y el manejo de eventos (control).

Los *eventos* se pueden definir brevemente como acontecimientos a los cuales la aplicación debe responder.

---

**58 ExecView** inserta una vista en el grupo, ejecuta la nueva subvista, y después borra la subvista cuando el usuario termina el estado modal.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

En un programa tradicional de Pascal, típicamente se escribe un bucle de código que lee las entradas de usuario de teclado, ratón, y otras, y después toma decisiones basadas en esa entrada. Se invoca a procedimientos o funciones, o se bifurca a un bucle de código en alguna otra parte que de nuevo lee la entrada del usuario:

```
repeat
 B := ReadKey;
 case B of
 'i': InvertirArray;
 'e': EditarArray;
 'v': VisualizarArray;
 'q': Salir := true;
 end;
until Salir;
```

Un programa dirigido por eventos no se estructura de una forma muy distinta. De hecho, es difícil imaginar un programa interactivo que no funcione así. Sin embargo, un programa dirigido por eventos se ve diferente desde el punto de vista del programador.

En una aplicación Turbo Vision, no se tiene que leer la entrada del usuario porque Turbo Vision se encarga de esta tarea. Empaqueta la entrada en registros Pascal denominados **eventos**, y los envía a las vistas apropiadas del programa. Eso significa que el código sólo necesita saber cómo tratar la entrada pertinente, en vez de entresacar del flujo de entrada la información a manejar.

Por ejemplo, si el usuario pincha una ventana inactiva, Turbo Vision lee la acción del ratón, la empaqueta en un registro de evento, y envía el registro de evento a la ventana inactiva.

Las vistas pueden manejar gran parte de la entrada de usuario por sí mismas. Una ventana sabe cómo abrirse, cerrarse, moverse, ser seleccionada y redimensionarse. Un menú sabe cómo abrirse, interactuar con el usuario, y cerrarse. Los botones saben como ser pulsados, cómo interactuar con el resto y cómo cambiar de color. Las barras de desplazamiento saben cual debe ser su comportamiento. La ventana inactiva puede activarse a sí misma sin ninguna atención por parte del programador.

El trabajo del programador consistirá en definir nuevas vistas con nuevas acciones, que necesitarán conocer ciertas clases de eventos que se definan. Enseñar a esas vistas a responder a comandos estándar, e incluso a generar sus propios comandos (*mensajes*) para otras vistas. El mecanismo ya está puesto, no hace falta programarlo. Todo lo que hay que hacer es generar comandos y enseñar a las vistas a responder a ellos.

### CLASIFICACION DE EVENTOS

Se pueden ver los eventos como pequeños paquetes de información que describen sucesos a los cuales necesita responder la aplicación. Cada pulsación de tecla, cada acción del ratón, y ciertas condiciones producidas por otros componentes del programa, hace que se genere un evento. Los eventos no se pueden romper en elementos más pequeños. Por lo tanto, cuando el usuario teclea una palabra no se genera un único evento, sino una serie de eventos individuales de pulsación de tecla.

## EVENTOS EN TURBO VISION

En el mundo orientado a objetos de Turbo Vision, probablemente se espera que los eventos sean también objetos. Pero no lo son. Los eventos por sí mismos no desarrollan ninguna acción. Ellos sólo transportan información a los objetos, por lo que son estructuras registro.

```
TEvent = record
 What: Word;
 case Word of
 evNothing: (...);
 evMouse: (...);
 evKeyDown: (...);
 evMessage: (...);
 end;
```

El corazón de todo registro de evento es un campo de tipo *Word* denominado **What**. El valor numérico del campo *What* describe la clase de evento ocurrido, y el resto del registro de evento contiene información específica sobre ese evento: el código scan de teclado para un evento de pulsación de tecla, información sobre la posición del ratón y el estado de sus botones para un evento de ratón, etcétera.

Los eventos se pueden clasificar en una primera instancia en cuatro categorías: *eventos de ratón*, *eventos de teclado*, *eventos de mensaje*, y *eventos nulos*. Cada categoría tiene definida una máscara, así los objetos pueden determinar rápidamente qué tipo de evento ha ocurrido sin preocuparse de su tipo específico. Por ejemplo, en vez de comprobar los cuatro tipos de eventos de ratón diferentes, se puede verificar simplemente si el flag de evento está en la máscara. En lugar de

```
if Event.What and
 (evMouseDown or evMouseUp or evMouseMove or evMouseAuto) <> 0 then
```

se puede hacer

```
if Event.What and evMouse <> 0 then
```

Las máscaras disponibles para distinguir eventos son *evNothing* (para eventos nulos), *evMouse* para eventos de ratón, *evKeyboard* para eventos de teclado, y *evMessage* para mensajes.

```
evNothing = $0000;
evMouse = $000F;
evKeyboard = $0010;
evMessage = $FF00;
```

Los bits de máscara de eventos se definen en la figura 14.32.

• **Eventos de ratón.** Básicamente hay cuatro clases de eventos de ratón: un click arriba o abajo con cada botón, un cambio de posición, o un evento "auto". Presionando un botón del ratón tiene lugar un evento *evMouseDown*. Soltando el botón se genera un evento *evMouseUp*. Moviendo el ratón se produce un evento *evMouseMove*. Y si se deja pulsado el botón, Turbo Vision generará periódicamente un evento *evMouseAuto*, permitiendo a la aplicación llevar a cabo acciones tales como un *scroll* repetido. Todos los registros de evento de ratón incluyen la posición del ratón, así un objeto que procesa el evento sabe dónde estaba el ratón cuando éste ocurrió.

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
evMouseDown = $0001;
evMouseUp = $0002;
evMouseMove = $0004;
evMouseAuto = $0008;
```

• **Eventos de teclado.** Los eventos de teclado son incluso más sencillos. Cuando se pulsa una tecla, Turbo Vision genera un evento *evKeyDown*, el cual recuerda qué tecla fue pulsada.

```
evKeyDown = $0010;
```

• **Eventos de mensaje.** Los eventos de mensaje provienen de comandos, emisiones o mensajes de usuario. La diferencia está en cómo son manejados, lo cual se explica más adelante. Básicamente, los comandos se marcan en el campo *What* como *evCommand*, las emisiones como *evBroadcast*, y los mensajes definidos por el usuario utilizan alguna constante también definida por el usuario.

```
evCommand = $0100;
evBroadcast = $0200;
```

• **Eventos nulos.** Un evento nulo es realmente un evento muerto. Ha dejado de ser un evento, porque ya ha sido manejado. Si el campo *What* de un registro de evento contiene el valor *evNothing*, ese registro de evento contiene información no útil que no necesita ser tratada.

Cuando un objeto de Turbo Vision finaliza el manejo de un evento, invoca a un método denominado *ClearEvent*, el cual da al campo *What* el valor *evNothing*, indicando que el evento ha sido manejado. Los objetos deberían simplemente ignorar los eventos *evNothing*.

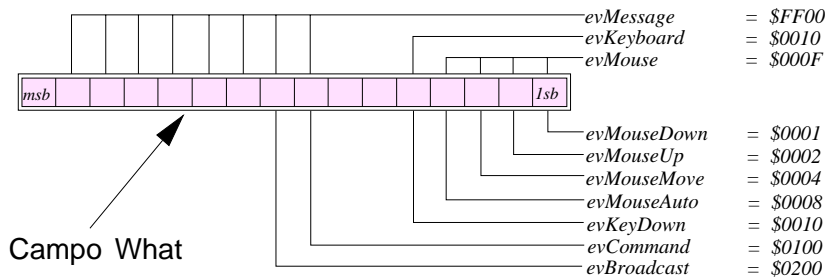


Fig. 14.32 Mapeo de bits del campo *TEvent.What*

La mayoría de los eventos terminan siendo traducidos a comandos. Por ejemplo, pinchando un elemento de la línea de estado se genera un evento de ratón. Cuando llega al objeto línea de estado, ese objeto responde al evento de ratón generando un evento de comando, con un valor del campo *Command* determinado por el comando vinculado al elemento de la línea de estado. Pinchando sobre **Alt-X Exit** se genera el comando *cmQuit*, el cual interpreta la aplicación como una instrucción para cerrar y terminar.

## ENCAMINAMIENTO DE EVENTOS

Las vistas de Turbo Vision operan bajo el principio "*Habla sólo cuando te hablen*". Es decir, en vez de buscar activamente entradas, esperan pasivamente a que el gestor de eventos les diga que ha sucedido un evento al que necesitan responder.

Para que los programas Turbo Vision actúen de la manera que se espera de ellos, no sólo debe decirse a las vistas qué hacer cuando ocurran ciertos eventos, también es necesario comprender cómo llegan los eventos a las vistas. La clave para que los eventos lleguen al lugar adecuado es el correcto encaminamiento de los eventos. Algunos eventos se emiten por toda la aplicación, mientras que otros se dirigen de forma rigurosa a partes particulares del programa.

El bucle principal de procesamiento de un objeto *TApplication* se activa cuando el método *Run* invoca a *TGroup.Execute*, que es básicamente un bucle repeat parecido a este:

```

var
 E: TEvent;
begin
 E.What := evNothing; { indica que no ha ocurrido ningún evento }
 repeat
 if E.What <> evNothing then EventError(E);
 GetEvent(E); { empaqueta un registro de evento }
 HandleEvent(E); { encamina el evento al lugar adecuado }
 until EstadoFinal <> Continuar; { hasta que se active el flag de fin }
end;

```

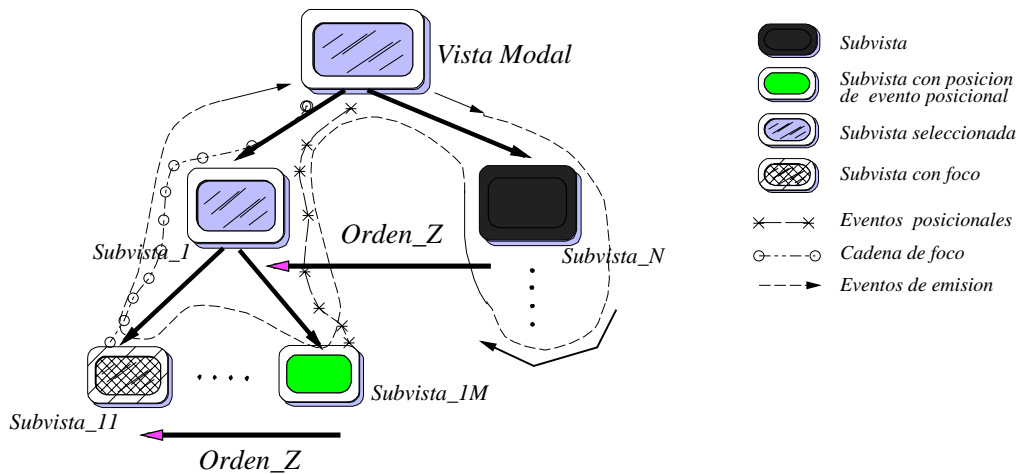


Fig. 14.33 Encaminamiento de eventos.

*GetEvent* mira a su alrededor y comprueba si ha ocurrido algo que debería ser un evento. Si lo hay, *GetEvent* crea el registro de evento apropiado. Después *HandleEvent* encamina el evento

a las vistas adecuadas. Si el evento no es manejado (y desactivado) para cuando vuelve a este bucle, se invoca a *EventError* para indicar una evento abandonado. Por defecto, *EventError* no hace nada.

Los eventos siempre comienzan su ruta por la vista modal en curso. Generalmente ésta será el objeto aplicación. Cuando se ejecuta un cuadro de diálogo modal, ese objeto cuadro de diálogo es la vista modal. En cualquier caso, la vista modal es la que inicia el manejo del evento. Dónde va el evento a partir de ahí depende de la naturaleza del mismo.

Los eventos se encaminan de una de las tres maneras posibles, dependiendo del tipo de eventos que sean: *posicional*, *enfocada*, y *emisión*. Es importante comprender cómo es encaminado cada tipo de evento. La figura 14.33 muestra los tres tipos de rutas que puede seguir un evento dependiendo de su naturaleza.

- ✧ **Eventos posicionales.** Los eventos posicionales son casi siempre eventos de ratón (*evMouse*).

La vista modal obtiene primero el evento posicional, y empieza a mirar a sus subvistas en orden-Z hasta que encuentra una que contiene la posición donde ocurrió el evento. Entonces la vista modal pasa el evento a esa vista. Dado que las vistas pueden solaparse, es posible que más de una contenga ese punto. Siguiendo el orden-Z se garantiza que la vista más al frente de todas las que lo contienen será la que reciba el evento. Después de todo, es en ésta en la que el usuario hizo el click.

Este proceso continúa hasta que un objeto no pueda encontrar una vista a la que pasar el evento, bien porque sea una vista terminal (que no tiene subvistas) o porque no hay ninguna subvista en la posición donde ocurrió el evento (como cuando se pincha en un espacio despejado en un cuadro de diálogo). En ese punto, el evento ha alcanzado el objeto donde tuvo lugar el evento posicional, y ese objeto maneja el evento.

- ✧ **Eventos enfocados.** Los eventos enfocados generalmente son pulsaciones de teclas (*evKeyDown*) o comandos (*evCommand*), que van bajando por la cadena de foco hasta llegar a la vista que maneja estos eventos.

La vista modal en curso obtiene primero el evento enfocado, y lo pasa a su subvista seleccionada. Si esa subvista tiene una subvista seleccionada, le pasa el evento. Este proceso continúa hasta que se alcanza una vista terminal: Esta es la vista enfocada. La vista enfocada recibe y maneja el evento enfocado.

## EVENTOS EN TURBO VISION

Si la vista enfocada no sabe cómo manejar el evento particular que recibe, devuelve el evento por la cadena de enfoque a su propietaria. Este proceso se repite hasta que el evento es manejado o alcanza de nuevo la vista modal. Si la vista modal no sabe cómo manejar el evento cuando este regresa, invoca a *EventError*. Esta situación constituye un *evento abandonado*.

Los eventos de teclado ilustran el fundamento de los eventos enfocados muy claramente. Por ejemplo, en el entorno integrado del Turbo Pascal, podría haber varios ficheros abiertos en ventanas editor en el *desktop*. Cuando se pulsa una tecla, se sabe qué fichero se quiere que reciba el carácter.

La pulsación de tecla produce un evento *evKeyDown*, el cual se dirige a la vista modal en curso, el objeto *TApplication*. *TApplication* envía el evento a su vista seleccionada, el *desktop* (el *desktop* siempre es la vista seleccionada de *TApplication*). El *desktop* envía el evento a su vista seleccionada, que es la ventana activa (la que tiene el marco con doble línea). Esa ventana editor también tiene subvistas —un marco, una vista interior con *scroll*, y dos barras de desplazamiento. De todas ellas, sólo el interior es seleccionable (y por lo tanto seleccionado por defecto), luego el evento de teclado va ahí. La vista interior, un editor, no tiene subvistas, así que decide cómo manejar el carácter del evento *evKeyDown*.

✕ **Eventos de emisión.** Los eventos de emisión generalmente son emisiones (*ev-Broadcast*) o mensajes definidos por el usuario.

Los eventos de emisión no son tan directos como los eventos posicionales o los enfocados. Por definición, una emisión no conoce su destino, luego se envía a *todas* las subvistas de la vista modal en curso.

La vista modal actual obtiene el evento, y comienza a pasárselo a sus subvistas en orden-Z. Si alguna de esas subvistas es un grupo, éste también envía el evento a sus subvistas, de nuevo en orden-Z. El proceso continúa hasta que todas las vistas que pertenecen (directa o indirectamente) a la vista modal hayan recibido el evento, o hasta que una vista lo anule.

Los eventos de emisión son comúnmente usados para la *comunicación entre vistas*. Por ejemplo, cuando se pincha una barra de desplazamiento de un visor de fichero, la barra de desplazamiento necesita hacer saber a la vista texto que debería mostrar alguna otra parte de sí misma. Hace eso enviando una emisión que indica que ha cambiado y que otras vistas, incluyendo la vista texto, recibirán y reaccionarán a ella.

### • Eventos definidos por el usuario

A medida que se va conociendo mejor la forma de trabajar en Turbo Vision con los eventos, se puede desear definir nuevas categorías completas de eventos, utilizando los bits de mayor orden del campo *What* del registro de evento. Por defecto, Turbo Vision encaminará tales eventos como eventos de emisión. Pero se puede desear que los nuevos eventos sean enfocados o posicionales, y Turbo Vision proporciona un mecanismo para permitir esto.

Turbo Vision define dos máscaras, *PositionalEvents* y *FocusedEvents*, que contienen los bits del campo *What* del registro de evento que corresponden a eventos que deberían ser encaminados como posicionales y como enfocados, respectivamente. Por defecto, *PositionalEvents* contiene todos los bits de *evMouse*, y *FocusedEvents* contiene *evKeyboard* y *evCommand*. Si se define algún otro bit como una nueva clase de evento que se quiera encaminar por posición o por foco, simplemente se añade ese bit a la máscara apropiada.

```
PositionalEvents: Word = evMouse;
FocusedEvents: Word = evKeyboard + evCommand;
```

### • Enmascarar eventos

Todo objeto vista tiene un campo bitmap denominado *EventMask* (`EventMask: Word;`) que se usa para determinar qué eventos manejará la vista. Los bits de *EventMask* corresponden a los bits del campo *TEvent.What*. Si el bit de una clase de evento está activado, la vista aceptará manejar ese tipo de evento. Si el bit de una clase de evento está desactivado, la vista ignorará ese tipo de evento.

Por ejemplo, por defecto el campo *EventMask* de una vista excluye *evBroadcast*, pero el *EventMask* de un grupo lo incluye. Por lo tanto, los grupos reciben por defecto eventos de emisión, pero las vistas no.

Hay ciertas ocasiones en que se quiere que una vista que no es la vista enfocada maneje eventos enfocados (especialmente pulsaciones de teclas). Por ejemplo, cuando se mira una ventana de texto con *scroll*, se querría usar pulsaciones de tecla para realizar el *scroll* del texto, pero dado que la ventana de texto es la vista enfocada, los eventos de tecla van a ella, y no a las barras de desplazamiento que pueden hacer el *scroll*.

Turbo Vision proporciona un mecanismo para permitir que otras vistas aparte de la enfocada puedan ver y manejar eventos enfocados denominado *fase de encaminamiento de eventos*. Aunque el encaminamiento descrito anteriormente es esencialmente correcto, hay *dos excepciones al estricto encaminamiento de la cadena de foco*.

Cuando la vista modal obtiene un evento enfocado que manejar, realmente existen tres fases de encaminamiento:

- El evento es enviado a las subvistas (en orden-Z) que tienen activado su flag de opciones *ofPreProcess*.



- Si el evento no es anulado por ninguna de ellas, se envía a la vista enfocada.
- Si el evento todavía no ha sido anulado, se envía (de nuevo en orden-Z) a las subvistas que tengan activado su flag de opciones *ofPostProcess*.

Así en el ejemplo anterior, si una barra de desplazamiento necesita ver pulsaciones de tecla que están dirigidas a la vista de texto enfocada, la barra de desplazamiento debería ser inicializada con su flag de opciones *ofPreProcess* activado (figura 14.34).

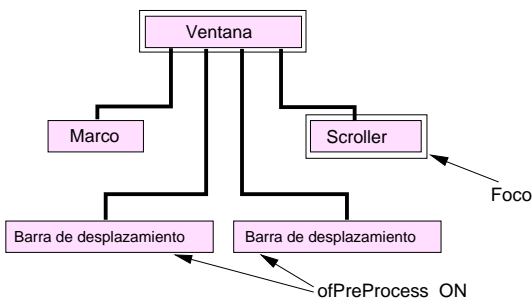


Fig. 14.34 Encaminamiento *ofPreProcess*

En este ejemplo concreto no existe mucha diferencia entre activar *ofPreProcess* o *ofPostProcess*: ambos funcionarán. Dado que en este caso la vista enfocada no maneja el evento (el propio *TScroller* no hace nada con las pulsaciones de tecla), las barras de desplazamiento pueden mirar los eventos tanto antes como después de que el evento sea encaminado al desplazador.

En general, sin embargo, en un caso como éste se utiliza *ofPostProcess*; eso proporciona mayor flexibilidad. Más adelante podría quererse añadir funcionalidad al interior chequeando las pulsaciones de tecla. Sin embargo, si las pulsaciones han sido capturadas por la barra de desplazamiento antes de llegar a la vista enfocada (*ofPreProcess*), el interior nunca actuará sobre ellas.

Aunque hay veces que se necesita atrapar eventos enfocados antes de que la vista enfocada pueda acceder a ellos, es una buena idea dejar tantas opciones abiertas como sea posible. De esa forma, se podrá derivar algún nuevo objeto de éste en el futuro.

Todo grupo tiene un campo denominado **Phase**, el cual tiene uno de tres posibles valores: *phFocused*, *phPreProcess*, y *phPostProcess*. Comprobando el flag *Phase* de su propietario, una vista puede decir si el evento que está manejando le llega antes, durante, o después del encaminamiento enfocado. Esto es a veces necesario, porque algunas vistas buscan eventos diferentes, o reaccionan a los mismos eventos diferentemente, dependiendo de la fase.

```
Phase: (phFocused, phPreProcess, phPostProcess);
```

Considérese el caso de un cuadro de diálogo sencillo que contiene un campo de edición y un botón etiquetado *Aceptar*, donde **A** es el acelerador de teclado del botón. Con controles de cuadro de diálogo normales, realmente no hay que preocuparse por la fase. La mayoría de los controles tienen *ofPostProcess* activado por defecto, luego las pulsaciones de tecla (eventos enfocados) llegarán a ellos y les permitirán atrapar el foco si es su acelerador de teclado el que se ha teclado. Pulsando **A** se mueve el foco al botón *Aceptar*.

Pero supóngase que el campo de edición tiene el foco, con lo que las pulsaciones de tecla son manejadas e insertadas en el campo de edición. Pulsando la tecla **A** se pone una **A** en el campo de edición, y el botón nunca llega a ver el evento, puesto que la vista enfocada lo manejó. La primera reacción instintiva podría ser que el botón buscara la tecla **A** en el preproceso, y poder así tratar la tecla de atajo antes de que lo haga la vista enfocada. Desafortunadamente, esto excluiría la posibilidad de teclear la letra **A** en el campo de edición.

La solución es que el botón compruebe diferentes aceleradores de teclado antes y después de que la vista enfocada maneje el evento. Por defecto, un botón buscará su acelerador de teclado en la forma **Alt+letra** en el preproceso, y como **letra** en el postproceso. Por eso es por lo que siempre se pueden usar aceleradores **Alt+letra** en un cuadro de diálogo, pero también se pueden usar las letras normales cuando el control enfocado (*e.j.* campo de edición) no las recoge.

Por defecto, los botones tienen activados tanto *ofPreProcess* como *ofPostProcess*, así que pueden ver eventos enfocados antes y después de que lo haga la vista enfocada. Pero dentro de su *HandleEvent*, el botón comprueba sólo ciertas pulsaciones de tecla si el control enfocado ha visto ya el evento:

```

evKeyDown: { esto es parte de una sentencia case }
 begin
 C := HotKey(Title^); { obtiene letra de atajo del botón }

 { 1. Alt+letra en preproceso }
 { 2. letra en postproceso }
 { 3. espacio si el botón tiene el foco }

 if (Event.KeyCode = GetAltCode59(C)) or { 1 }
 (Owner^.Phase = phPostProcess) and (C <> #0) { 2 }
 and (Ucase(Event.CharCode) = C) or
 (State and sfFocused <> 0) and (Event.CharCode = ' ') then { 3 }
 begin
 Press; { genera el efecto asociado con pulsar el botón }
 ClearEvent(Event); { Eliminar evento }
 end;
 end;

```

## COMANDOS

La mayoría de los eventos posicionales y enfocados son traducidos a comandos por los objetos que los manejan. Es decir, un objeto a menudo responde a un click del ratón o a una pulsación de tecla generando un evento de comando.

Por ejemplo, pinchando la línea de estado en una aplicación Turbo Vision, se genera un evento posicional (de ratón). La aplicación determina que el click estaba posicionado en el área controlada por la línea de estado, así que le pasa el evento al objeto línea de estado, *StatusLine*.

---

<sup>59</sup> *GetAltCode(C)* devuelve el código de scan de 2 bytes que se generaría al pulsar la combinación *Alt+C*.

## EVENTOS EN TURBO VISION

*StatusLine* determina cuál de sus elementos de estado controla el área donde se hizo click con el ratón, y lee el registro de ese elemento de estado. Dicho elemento generalmente tendrá vinculado un comando, luego *StatusLine* creará un registro de evento pendiente donde el campo *What* tendrá el valor *evCommand* y el campo *Command* contendrá el comando vinculado a ese elemento de estado. Después anulará el evento de ratón, lo que significa que el siguiente evento encontrado por *GetEvent* será el evento de comando recién generado.

```
TEvent = record
 What: Word; { vale evCommand }
 case Word of
 evNothing: (...);
 evMouse: (...);
 evKeyDown: (...);
 evMessage: (
 Command: Word; { contiene comando vinculado }
 case Word of
 ...
 end;
 end;
```

Turbo Vision tiene muchos comandos predefinidos, y en una aplicación se definirán muchos más. Cuando se crea una nueva vista, también se crea un comando para invocarla. Los comandos se pueden llamar de cualquier manera, pero el convenio de Turbo Vision es que un identificador de comando debería empezar por **cm**. La creación de un comando es sencilla —basta crear una constante:

```
const
 cmObtenerFicha = 114;
```

Turbo Vision reserva los comandos de 0 a 99 y de 256 a 999 para uso propio. Las aplicaciones pueden utilizar los números de 100 a 255 y de 1.000 a 65.535 para comandos.

| Rango         | Reservado | Puede ser desactivado |
|---------------|-----------|-----------------------|
| 0 .. 99       | Sí        | Sí                    |
| 100 .. 255    | No        | Sí                    |
| 256 .. 999    | Sí        | No                    |
| 1000 .. 65535 | No        | No                    |

Tabla 14.5 Rangos de comandos de *Turbo Vision*.

La razón para tener dos rangos de comandos es que sólo los comandos del 0 al 255 pueden ser desactivados. Turbo Vision reserva algunos de los comandos que pueden ser desactivados y algunos de los que no lo pueden ser para sus comandos estándar y trabajos internos. Sobre el resto de los comandos se tiene control completo. Los rangos de comandos disponibles se resumen en la Tabla 14.5. El ejemplo 14.6 muestra la unit *Comandos* que se utiliza en el programa AGENDA, donde se definen los comandos que maneja el programa aparte de los estándar de Turbo Vision.

### Ejemplo 14.6.

```

unit Comandos;

interface
const
 cmNuevaFicha = 110;
 cmVentanaFicha = 111;
 cmSalvarFicha = 112;
 cmCancelarFicha = 113;
 cmSigFicha = 114;
 cmAntFicha = 115;
 cmAyuda = 130;
 cmIndiceAyuda = 131;
 cmMostrarClip = 260;
 cmAcerca = 270;
 cmOpcionesVideo = 1100;
 cmOpcionesSalvar = 1101;
 cmOpcionesCargar = 1102;
 cmRaton = 1103;
 cmColor = 1104;
 cmBuscarVentanaFicha = 1200;
 cmCalculadora = 1201;
 cmCalendario = 1202;
implementation
end.

```

Cuando se crea un elemento de menú o un elemento de línea de estado, se le vincula un comando. Cuando el usuario escoge ese comando, se genera un registro de evento, con el campo *What* puesto a *evCommand*, y el campo *Command* puesto al valor del comando vinculado. El comando puede ser un comando estándar de Turbo Vision o bien uno propio. Al mismo tiempo que se vincula el comando a un elemento de menú o de línea de estado, se puede también vincular una hot key. De esa forma, el usuario puede invocar al comando pulsando un acelerador de teclado en vez de utilizar los menús o el ratón.

```

{ elemento de menú con comando asociado propio de la aplicación }
NewItem('~N-ueva Ficha', 'F9', kbF9, cmNuevaFicha, hcNuevaFicha, ...

{ elemento de línea de estado con comando asociado estándar }
NewStatusKey('~Alt+S~ Salir', kbAltS, cmQuit, ...

```

La definición del comando no especifica la acción a tomar cuando éste aparezca en un registro de evento. Para esto es preciso decirles a los objetos apropiados cómo responder a ese comando.

Hay momentos en los que se quiere que ciertos comandos no estén disponibles para el usuario durante un período de tiempo. Por ejemplo, si no se tienen ventanas abiertas, no tiene sentido que el usuario pueda generar *cmClose*, el comando estándar de cierre de ventana. Turbo Vision proporciona una manera de **activar** y **desactivar** conjuntos de comandos.

```
TCommandSet = set of Byte;
```

Para activar y desactivar un grupo de comandos, se utiliza el tipo global *TCommandSet*, que es un conjunto de números del 0 al 255. (por ello sólo pueden ser desactivados los comandos dentro del rango 0..255) El código siguiente desactiva un grupo de cinco comandos relacionados con ventanas:

## EVENTOS EN TURBO VISION

```
var
 ComandosVentana: TCommandSet;
begin
 ComandosVentana := [cmNext, cmPrev, cmZoom, cmResize, cmClose];
 DisableCommands(ComandosVentana);
end;
```

Para activar y desactivar comandos de forma que una vista sea insensible a un conjunto de comandos *TView* define dos métodos que permiten realizar esta tarea:

```
procedure DisableCommands(Commandos: TCommandSet); { desactiva Comandos }
procedure EnableCommands(Commandos: TCommandSet); { activa Comandos }
```

## UTILIZACION DE LOS EVENTOS

Una vez se ha definido un comando y establecido alguna clase de control que lo genere —por ejemplo, un elemento de menú o un botón de cuadro de diálogo —es necesario enseñar a la vista cómo responder cuando ocurra ese comando.

Toda vista hereda un método *HandleEvent* que ya sabe cómo responder a gran parte de la entrada de usuario. Si se quiere que una vista haga algo específico para la aplicación, es necesario redefinir su método *HandleEvent* y enseñarle al nuevo *HandleEvent* dos cosas: cómo responder a los nuevos comandos que se hayan definido, y cómo responder a los eventos de ratón y teclado de la forma que se pretende.

El método *HandleEvent* de una vista determina cómo se comporta ésta. Dos vistas con métodos *HandleEvent* idénticos responderán a eventos de la misma manera. Cuando se deriva un nuevo tipo de vista, generalmente se quiere que se comporte más o menos como su vista antecesora (padre), con algunos cambios. La manera más fácil de conseguir ésto es invocar al método *HandleEvent* del padre como parte del método *HandleEvent* del nuevo objeto.

El esquema general del método *HandleEvent* de un descendiente sería:

```
procedure TNuevoDescendiente.HandleEvent(var Evento: TEvent);
begin
 { código para cambiar o eliminar el comportamiento paternal }
 inherited HandleEvent(Evento);
 { código para desarrollar funciones adicionales }
end;
```

En otras palabras, si se quiere que el nuevo objeto maneje ciertos eventos de forma diferente que su ascendiente (o no los maneje en absoluto), se atraparían esos eventos particulares antes de pasar el evento al método *HandleEvent* del ascendiente. Si se quiere que el nuevo objeto se comporte igual que su ascendiente, pero con ciertas funciones adicionales, se añadiría el código para ellas después de la llamada al procedimiento *HandleEvent* del ascendiente.

En el ejemplo 14.7 se define el método *HandleEvent* del tipo objeto *TVentanaFicha* descendiente de *TDialog* que se utiliza para presentar un diálogo donde se introduce y se presenta la información de una ficha de la agenda. Antes de llamar al *HandleEvent* del ascendiente se define un cambio en el comportamiento para cuando se quiera cerrar la ventana y se genere así el comando

estándar *cmClose*. Cuando se está insertando una ficha no se permite definir otra nueva hasta que se guarde la ficha en inserción o se cancele la operación (comando *cmNuevaFicha* inactivo). Por ello si se cierra la ventana se deberá activar el comando *cmNuevaFicha* para que se pueda abrir posteriormente la ventana de ficha. Sino esta opción quedaría inhabilitada durante el resto de la ejecución del programa.

Para realizar esta acción no podemos definirla como un comportamiento adicional al del ascendiente, ya que el comando *cmClose* es estándar y será consumido por el *HandleEvent* de *TDialog*, que al tratar el evento de comando pondrá el campo *What* a *evNothing* después de encargarse de cerrar la ventana y no dejará rastro del evento que se estaba tratando. Por ello deberá definirse un comportamiento previo al cierre de la ventana que activará el comando *cmNuevaFicha* cuando se genere un comando *cmClose*. Por otro lado en este caso, una vez activado el comando no se puede destruir el evento pues a continuación debe ser tratado por el método del ascendiente que se encargará de cerrar la ventana.

Cuando el método *HandleEvent* de una vista ha manejado un evento, finaliza el proceso invocando a su método *ClearEvent*. *ClearEvent* **anula un evento** dando al campo *Evento.What* el valor *evNothing* y a *Evento.InfoPtr*<sup>60</sup> el valor *@Self*, que son las señales universales de que el evento ha sido manejado. Si el evento fuese pasado a otro objeto, ese objeto ignoraría este *evento nulo*. *ClearEvent* también ayuda a las vistas a comunicarse entre sí. Baste recordar que no se ha terminado de manejar un evento hasta que se invoque a *ClearEvent*.

#### Ejemplo 14.7.

```

procedure TVentanaFicha.HandleEvent(var Evento: TEvent);
begin
 if Evento.What = evCommand then
 case Evento.Command of { Se debe activar el comando de nueva ficha }
 cmClose: { antes de cerrar la ventana de fichas. }
 EnableCommands([cmNuevaFicha]);
 end;
 inherited HandleEvent(Evento);
 if (Evento.What = evBroadcast) and
 (Evento.Command = cmBuscarVentanaFicha) then
 ClearEvent(Evento);
end;

```

En *TVentanaFicha* se añade un comportamiento adicional para el caso de que se reciba un evento de emisión (*evBroadcast*) con el comando *cmBuscarVentanaFicha*, comando que sólo es manejado por objetos *TVentanaFicha* y se utiliza para realizar comunicación entre vistas<sup>61</sup>.

El ejemplo 14.8 muestra la definición del manejador de eventos del tipo objeto *TGestionFichas*, descendiente de *TApplication*, que es el tipo de aplicación que se utiliza en el programa AGENDA. Este manejador añade el código necesario para manejar todos los comandos que se pueden producir en la aplicación y no son manejados por el ascendiente. Para tratar algunos eventos

---

<sup>60</sup> Ver el siguiente apartado *El registro de evento*.

<sup>61</sup> Ver el apartado *Comunicación entre vistas* en este capítulo

## EVENTOS EN TURBO VISION

se implementan procedimientos adicionales como *colores* que se utiliza para presentar el diálogo de modificación de colores de la aplicación. PColorDialog es un puntero al tipo objeto TColorDialog que está definido en la *Unit ColorSel*.

### Ejemplo 14.8.

```
procedure TGestionFichas.HandleEvent(var Evento: TEvent);
{ Utilidades Generales }
procedure Colores;
var
 D: PColorDialog;
begin
 D := New(PColorDialog, Init('',
 ColorGroup('Desktop', DesktopColorItems(nil),
 ColorGroup('Menus', MenuColorItems(nil),
 ColorGroup('Dialogos|Calc', DialogColorItems(dpGrayDialog, nil),
 ColorGroup('Editor', WindowColorItems(wpBlueWindow, nil),
 ColorGroup('Calendario',
 WindowColorItems(wpCyanWindow,
 ColorItem('Día actual', 22, nil)),
 nil))));
 D^.HelpCtx := hcDialogoColores;
 if ExecuteDialog(D, Application^.GetPalette) <> cmCancel then
 begin
 DoneMemory; { Elimina todos los buffers de grupo }
 ReDraw; { Redibuja la aplicación con la nueva paleta }
 end;
end;

procedure Raton;
var
 D: PDialog;
begin
 D := New(PDialogoRaton, Init);
 D^.HelpCtx := hcDialogoRaton;
 ExecuteDialog(D, @MouseReverse);
end;

procedure Calculadora;
var
 P: PCalculator;
begin
 P := New(PCalculator, Init);
 P^.HelpCtx := hcCalculadora;
 InsertWindow(P);
end;

procedure Calendario;
var
 P: PVentanaCalendario;
begin
 P := New(PVentanaCalendario, Init);
 P^.HelpCtx := hcCalendario;
 InsertWindow(P);
end;

var
 R: TRect;
begin
 { TGestionFichas.HandleEvent }
 inherited HandleEvent(Evento);
 if Evento.What = evCommand then
 begin
 case Evento.Command of
 cmNuevaFicha:
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
begin
 MeterNuevaFicha;
 ClearEvent(Evento);
end;
cmCancelarFicha:
begin
 CancelarFicha;
 ClearEvent(Evento);
end;
cmSigFicha:
begin
 MostrarFicha(FichaActiva + 1);
 ClearEvent(Evento);
end;
cmAntFicha:
begin
 MostrarFicha(FichaActiva - 1);
 ClearEvent(Evento);
end;
cmSalvarFicha:
begin
 SalvarDatosFicha;
 ClearEvent(Evento);
end;
cmVentanaFicha:
begin
 AbrirVentanaFicha;
 ClearEvent(Evento);
end;
cmOpcionesCargar:
begin
 CargarDesktop;
 ClearEvent(Evento);
end;
cmOpcionesSalvar:
begin
 SalvarDesktop;
 ClearEvent(Evento);
end;
cmMostrarClip:
with VentanaClipboard^ do
begin
 Select;
 Show;
 ClearEvent(Evento);
end;
cmNew:
begin
 NuevaVentana;
 ClearEvent(Evento);
end;
cmOpen:
begin
 AbrirVentana;
 ClearEvent(Evento);
end;
cmOpcionesVideo:
begin
 SetScreenMode(ScreenMode xor smFont8x8);
 { Actualizar la posición de la vista de memoria heap libre }
 GetExtent(R);
 Dec(R.B.X);
 R.A.X := R.B.X - 9; R.A.Y := R.B.Y - 1;
 Heap^.Locate(R);
 ClearEvent(Evento);
end;
cmRaton:
```



## EVENTOS EN TURBO VISION

```
begin
 Raton;
 ClearEvent(Evento);
end;
cmColor:
begin
 Colores;
 ClearEvent(Evento);
end;
cmCalculadora:
begin
 Calculadora;
 ClearEvent(Evento);
end;
cmCalendario:
begin
 Calendario;
 ClearEvent(Evento);
end;
cmAcerca:
begin
 CajaAbout;
 ClearEvent(Evento);
end;
end; { Fin CASE }
end; { Fin IF }
end; { Fin de Handle }
```

### • El registro de evento

La unit DRIVERS.TPU de Turbo Vision define el tipo *TEvent* como un registro:

```
TEvent = record
 What: Word;
 case Word of
 evNothing: ();
 evMouse: (
 Buttons: Byte;
 Double: Boolean;
 Where: TPoint);
 evKeyDown: (
 case Integer of
 0: (KeyCode: Word);
 1: (CharCode: Char;
 ScanCode: Byte));
 evMessage: (
 Command: Word;
 case Word of
 0: (InfoPtr: Pointer);
 1: (InfoLong: Longint);
 2: (InfoWord: Word);
 3: (InfoInt: Integer);
 4: (InfoByte: Byte);
 5: (InfoChar: Char));
 end;
```

*TEvent* es un registro variable. Se puede saber qué hay en el registro mirando el campo *What*.

De este modo, si *TEvent.What* es *evMouseDown*, *TEvent* contendrá:

```
Buttons: Byte;
Double: Boolean;
Where: TPoint;
```

Si *TEvent.What* es *evKeyDown*, el compilador permitirá acceder a los datos tanto a través del campo

```
KeyCode: Word;
```

como de los campos

```
CharCode: Char;
ScanCode: Byte;
```

El campo variable final del registro de evento almacena un valor *Pointer*, *Longint*, *Word*, *Integer*, *Byte* o *Char*. Este campo se utiliza de distintas maneras en Turbo Vision. Realmente las vistas pueden generar eventos por sí mismas y enviárselos a otras vistas. Cuando lo hacen, a menudo utilizan el campo *InfoPtr*.

Normalmente, todo evento será manejado por alguna vista en una aplicación. Si no se puede encontrar ninguna vista que maneje el evento, la vista modal invoca a *EventError*. *EventError* invoca al método *EventError* de la vista propietaria y así sucesivamente por el árbol arriba hasta que *TApplication.EventError* es invocado. Los eventos que no son manejados por ninguna vista se denominan **eventos abandonados**.

*TApplication.EventError* por defecto no hace nada. Puede ser útil redefinir *EventError* durante el desarrollo del programa para desplegar un cuadro de diálogo de error o emitir un sonido de aviso. Dado que el usuario final del software no es responsable del fallo del mismo a la hora de manejar un evento, tal cuadro de diálogo de error en una versión de distribución probablemente sería algo irritante.

#### • Modificación del mecanismo de eventos

Como ya hemos visto la vista modal en curso tiene un bucle principal de procesamiento similar al siguiente:

```
var
 E: TEvent;
begin
 E.What := evNothing;
 repeat
 if E.What <> evNothing then EventError(E);
 GetEvent(E);
 HandleEvent(E);
 until EstadoFinal <> Continuar;
end;
```

Una de las mayores ventajas de la programación dirigida por eventos es que el código no tiene que saber de dónde vienen los eventos. Un objeto ventana, por ejemplo, sólo necesita saber que cuando vea un comando *cmClose* en un evento, debería cerrarse. No le importa si el comando vino de un click sobre su icono de cierre, de una selección de menú, de una hot key, o de un mensaje de algún otro objeto del programa. Tampoco tiene que preocuparse de si ese comando está destinado para él. Todo lo que tiene que saber es que se le ha dado un evento que manejar, y puesto que sabe cómo manejar ese evento, lo hace.

## EVENTOS EN TURBO VISION

La clave de esta *caja negra* de los eventos es el método *GetEvent* de la aplicación. *GetEvent* es la única parte del programa que tiene que interesarse por la fuente de eventos. Los objetos de la aplicación simplemente invocan a *GetEvent* y dejan que él se encargue de leer el ratón, el teclado, y los eventos pendientes generados por otros objetos.

Si se quieren crear nuevas clases de eventos (por ejemplo, lectura de un controlador del dispositivo puerto serie), simplemente se redefiniría *TApplication.GetEvent* en el objeto aplicación. Como se puede comprobar en el código de *TProgram.GetEvent* en APP.PAS (Unit App.tpu), el bucle *GetEvent* explora el ratón y el teclado y después invoca a *Idle*. Para insertar una nueva fuente de eventos, se puede redefinir *Idle* para buscar caracteres en el puerto serie y generar eventos basados en ellos, o bien redefinir *GetEvent* para añadir una llamada a *GetCommandEvent(Event)* en el bucle, donde *GetCommandEvent* devuelve un registro de evento si hay disponible un carácter en el puerto serie.

El método *GetEvent* de la vista modal en curso invoca al método *GetEvent* de su propietario, y así sucesivamente, subiendo por el árbol de vistas hasta *TApplication.GetEvent*, que es donde realmente se va a buscar el siguiente registro.

Como Turbo Vision utiliza siempre *TApplication.GetEvent* para buscar realmente los eventos, se pueden modificar eventos para la aplicación entera redefiniendo este único método. Por ejemplo, para implementar macros de teclado, se podría vigilar los eventos devueltos por *GetEvent*, atrapar ciertas pulsaciones de teclas, y convertirlas en macros. Para el resto de la aplicación, el flujo de eventos vendría directamente del usuario.

```
procedure TMiAplicacion.GetEvent (var Evento: TEvent);
begin
 inherited GetEvent(Event); { invoca al método TApplication }
 .
 . { procesamiento especial aquí }
 .
end;
```

El ejemplo 14.9 muestra la redefinición de *GetEvent* en el tipo *TGestionFichas* del programa AGENDA. En este caso el procesamiento añadido no busca nuevos eventos ni redefine eventos ya existentes. Lo que hace es mirar a ver si se ha generado un comando *cmAyuda* o *cmIndiceAyuda* de manera que la gestión de la ayuda sensible al contexto se realice en este punto y estos eventos sean totalmente ocultos a los manejadores de eventos de las vistas, ya que el evento es anulado dentro de *GetEvent*.

### Ejemplo 14.9.

```
procedure TGestionFichas.GetEvent(var Evento: TEvent);
const
 AyudaEnUso: Boolean = False;
var
 V: PWindow;
 ContextoAyuda: Word;
 FichAyuda: PHelpFile;
 StrmAyuda: PDosStream;
begin
 inherited GetEvent(Evento);
 if Evento.What = evCommand then
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

case Evento.Command of
 cmAyuda, cmIndiceAyuda:
 if not AyudaEnUso then
 begin
 AyudaEnUso := True;
 StrmAyuda := New(PDosStream, Init(FicheroAyuda, stOpenRead));
 FichAyuda := New(PHelpFile, Init(StrmAyuda));
 if StrmAyuda^.Status <> stOk then
 begin
 MessageBox('No se puede abrir el fichero de ayuda.',
 nil, mfError + mfOkButton);
 Dispose(FichAyuda, Done);
 end
 else
 begin
 if (evento.command = cmAyuda) then
 ContextoAyuda := GetHelpCtx
 else
 ContextoAyuda := hcIndiceAyuda;
 V := New(PHelpWindow, Init(FichAyuda, ContextoAyuda));
 if ValidView(V) <> nil then
 begin
 ExecView(V);
 Dispose(V, Done);
 end;
 end;
 ClearEvent(Evento);
 end;
 AyudaEnUso := False;
 end; { Fin de CmAyuda, cmIndiceAyuda }
 end; { Fin CASE }
 end;

```

Otro beneficio del papel central de *TApplication.GetEvent* es que éste invoca a un método denominado *TApplication.Idle* si no hay listo ningún evento. *TApplication.Idle* es un método que se puede redefinir para llevar a cabo procesamiento concurrente con el de la vista en curso y aprovechar así el *tiempo muerto* entre eventos.

En el programa *Agenda* se utiliza para presentar al usuario la hora en la parte superior derecha de la pantalla y el espacio de memoria heap disponible en la parte inferior derecha. Además inhibe o activa los comandos para poner las ventanas del desktop en cascada o en mosaico si existe alguna ventana del desktop que lo permita. Para ello se redefine *Idle* en el tipo *TGestionFichas* descendiente de *TApplication*:

```

procedure TGestionFichas.Idle;

function Estileable(P: PView): Boolean; far;
begin { Permite saber si una ventana se puede poner en mosaico }
 Estileable := (P^.Options and ofTileable <> 0) and
 (P^.State and sfVisible <> 0);
end;

begin
 inherited Idle;
 Reloj^.Actualizar;
 Heap^.Actualizar;
 if Desktop^.FirstThat(@Estileable) <> nil then
 EnableCommands([cmTile, cmCascade])
 else
 DisableCommands([cmTile, cmCascade]);
end;

```

## EVENTOS EN TURBO VISION

Reloj y Heap son dos campos del tipo objeto *TGestionFichas*. Reloj es un puntero a ventanas del tipo *TVistaReloj* y Heap un puntero a ventanas del tipo *PVistaDeHeap*. Ambos tipos de vistas utilizan un método denominado *Actualizar* para visualizar la hora actual en un caso y la memoria del heap actualmente disponible en el otro. Estos tipos objeto se definen en la Unit *Utiles.tpu* que se puede ver el fichero *Utiles.pas* de los ejemplos del capítulo 14.

```
TGestionFichas = object(TApplication)
 VentanaClipboard: PEditWindow;
 VentanaFicha: PVentanaFicha;
 Reloj: PVistaReloj;
 Heap: PVistaDeHeap;
 constructor Init;
 ...
end;
```

Si se redefine el método *Idle* se debe tener precaución de llamar al método *Idle* heredado y también asegurarse de que ninguna de las tareas realizadas por este nuevo método suspenda la ejecución de la aplicación durante un periodo de tiempo excesivo, ya que si esto sucediese podría bloquear las entradas de usuario y dar la sensación de que el programa no responde a las ordenes que se le indiquen.

El motivo de que se llame al método *Idle* heredado se debe a que este método se encarga de tareas tales como la actualización de la línea de estado y la notificación a las vistas de los comandos que han sido habilitados o inhabilitados.

### • Comunicación entre vistas

Un programa Turbo Vision está encapsulado en objetos, y el código se escribe sólo dentro de objetos. Supóngase que un objeto necesita intercambiar información con otro objeto dentro del programa. En un programa tradicional, probablemente significaría copiar información de una estructura de datos a otra. En un programa orientado a objetos, eso puede no ser tan sencillo, dado que los objetos puede que no sepan dónde encontrar a otro.

La comunicación entre objetos vista no es algo tan sencillo como el envío de datos entre partes equivalentes de un programa Pascal tradicional.

Si se necesita realizar comunicación entre vista, la primera cuestión a responder es si se han dividido las tareas apropiadamente entre las dos vistas. Puede que el problema sea el de un diseño pobre del programa. Quizás lo realmente necesario sea combinar las dos vistas en una sola vista, o mover parte de una vista a la otra.

Si de hecho el diseño del programa es sólido, y las vistas necesitan aún comunicarse entre ellas, puede que el camino apropiado sea crear una vista intermediaria.

Por ejemplo, supóngase que hay un objeto hoja de cálculo y un objeto procesador de textos, y se quiere poder pegar algo de la hoja de cálculo en el procesador de textos, y viceversa. En una aplicación Turbo Vision, se puede conseguir esto mediante una comunicación vista-a-vista

directa. Pero supóngase que en el futuro se quiere añadir, por ejemplo, una base de datos a este grupo de objetos, y *pegar a y desde* la base de datos. Ahora sería necesario duplicar la comunicación establecida entre los dos primeros objetos para realizarla entre los tres.

Una mejor solución es establecer una *vista intermedia* —en este caso, por ejemplo, un portapapeles. Un objeto entonces necesitaría saber sólo cómo copiar algo al portapapeles, y cómo pegar algo del portapapeles. No importa cuántos nuevos objetos se añadan al grupo, el trabajo nunca se será más complicado.

Si se ha analizado la situación cuidadosamente y se está seguro de que el diseño del programa es sólido y que no se necesita crear un intermediario, se puede implementar una simple *comunicación entre dos vistas*.

Antes de que una vista pueda comunicarse con otra, primero debe saber dónde está la otra vista, y quizás hasta asegurarse de que la otra vista existe en ese momento.

Veamos un ejemplo. La unit *StdDlg* contiene un cuadro de diálogo denominado *TFileDialog* (la vista que aparece en el entorno integrado cuando se quiere cargar un nuevo fichero figura 14.35). *TFileDialog* tiene un *TFileList* que muestra un directorio del disco, y sobre él, un *TFileInputLine* que visualiza el fichero actualmente seleccionado para cargar. Cada vez que el usuario selecciona otro fichero en el *TFileList*, el *TFileList* necesita decirle al *TFileInputLine* que visualice el nuevo nombre de fichero.



Fig. 14.35 Cuadro de diálogo con comunicación entre vistas.

En este caso, *TFileList* puede estar seguro de que *TFileInputLine* existe, porque ambos son inicializados dentro del mismo objeto, *TFileDialog*. Para decir *TFileList* a *TFileInputLine* que el usuario ha seleccionado un nuevo nombre *TFileList* crea y envía un mensaje. *TFileList.FocusItem* envía el evento, y el *HandleEvent* de *TFileInputLine* lo recibe:

## EVENTOS EN TURBO VISION

```
procedure TFileList.FocusItem(Item: Integer);
var Evento: TEvent;
begin
 inherited FocusItem(Item); { invoca al método heredado primero }
 { TopView apunta a la vista modal en curso }
 Message(TopView, evBroadcast, cmFileFocused, List^.At(Item));
end;
procedure TFileInputLine.HandleEvent(var Evento: TEvent);
var Nombre: NameStr;
begin
 inherited HandleEvent(Evento);
 if (Evento.What = evBroadcast) and (Evento.Command = cmFileFocused)
 and (State and sfSelected = 0) then
 begin
 { Si es un directorio }
 if PSearchRec(Evento.InfoPtr)^.Attr and Directory <> 0 then
 Data^ := PSearchRec62(Evento.InfoPtr)^.Name + '\'+
 PFileDialog(Owner)^.Wildcard
 else { Sino es un fichero }
 Data^ := PSearchRec(Evento.InfoPtr)^.Name;
 DrawView;
 end;
end;
```

*Message* es una función que genera un evento de mensaje y devuelve un puntero al objeto (si lo hay) que manejó el evento. *message* crea un registro de evento con los argumentos *evBroadcast*, *cmFileFocused* y *List^.At(Item)* asignándolos a los campos *What*, *Command* y *InfoPtr*, respectivamente, del registro *TEvent*. *InfoPtr* es un puntero a una información pasada con el mensaje. En este caso se pasa el elemento seleccionado de la lista de ficheros *TFileList*, que es un registro *SearchRec*. El evento así construido se pasa al manejador de eventos de la vista pasada como primer argumento si es posible. En este caso la vista (*TopView*) es el diálogo *TFileDialog*. En el ejemplo, *TFileDialog.Handle* tomará el evento y hará un encaminamiento por emisión a sus subvistas, de forma que llegará al manejador *TFileInputLine.HandleEvent* que será el único que reconocerá el mensaje, redibujándose con el nuevo fichero seleccionado que se le pasa en *Evento.InfoPtr*.

```
function Message(Receptor: PView; What, Command: Word;
 InfoPtr: Pointer): Pointer;
```

*Message* devuelve *Nil* si *Receptor* es *Nil* o si el evento generado no fue manejado por ninguna vista. Si *Receptor* manejó el evento (*HandleEvent* devuelve *TEvent.What* como *evNothing*), *Message* devuelve *TEvent.InfoPtr*. En este ejemplo devolverá *Nil* pues *TFileInputLine.HandleEvent* no anula el evento una vez tratado. No obstante esto es indiferente, ya que el valor devuelto por la función *Message* no es tenido en cuenta ya que no se necesita para la tarea que se implementa.

*TFileList.FocusItem* usa la sintaxis extendida del Turbo Pascal (la directiva del compilador **\$X+**) para utilizar la función *Message* como un procedimiento, puesto que no importa qué resultado devuelva *Message*.

---

**62** *PSearchRec* apunta a variables de tipo *SearchRec*, que es el tipo de las vbles. utilizadas por los procedimientos *FindFirst* y *FindNext*.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

El valor devuelto por la función *Message* se puede utilizar para averiguar cual fue la vista que manejó el evento enviado, ya que se el evento tratado se anula con *ClearEvent* este hace que *InfoPtr* apunte al objeto que anuló el evento.

Veamos un ejemplo concreto. En el programa AGENDA si el usuario pide abrir la ventana mostrar fichas de la agenda , el código de apertura de la ventana necesita comprobar si ya hay alguna ventana de agenda abierta. Si no la hay, abre una; si la hay, la trae al primer plano.

El envío del mensaje de emisión es fácil:

```
Message(Desktop, evBroadcast, cmBuscarVentanaFicha, nil);
```

En el código del método *HandleEvent* de una ventana que es buscada hay un test que responde al comando de búsqueda anulando el evento:

```
case Evento.Command of
 cmBusquedaVentana: ClearEvent(Evento);
end;
```

que en el caso de una ventana de agenda que responde a *cmBuscarVentanaFicha* es el siguiente:

```
procedure TVentanaFicha.HandleEvent(var Evento: TEvent);
begin
 if Evento.What = evCommand then
 case Evento.Command of
 { Se debe activar el comando de nueva ficha }
 cmClose: { antes de cerrar la ventana de fichas. }
 EnableCommands([cmNuevaFicha]);
 end;

 inherited HandleEvent(Evento);

 if (Evento.What = evBroadcast) and
 (Evento.Command = cmBuscarVentanaFicha) then
 ClearEvent(Evento);
end;
```

*ClearEvent* no sólo da al campo *What* del registro de evento el valor *evNothing*; también da al campo *InfoPtr* el valor *@Self*. *Message* lee estos campos, y si el evento ha sido manejado, devuelve un puntero al objeto que manejó el evento de mensaje. En este caso, sería la ventana de agenda. Así que siguiendo la línea que envía la emisión, se incluirá

```
procedure TGestionFichas.AbrirVentanaFicha;
begin
 if Message(Desktop, evBroadcast, cmBuscarVentanaFicha, nil) = nil then
 begin
 VentanaFicha := New(PVentanaFicha, Init); { si no hay, crea una }
 InsertWindow(VentanaFicha); { y la inserta en el Desktop }
 end
 else
 { sino la trae al primer plano }
 if PView(VentanaFicha) <> Desktop^.TopView then { si no lo está }
 VentanaFicha^.Select;
 ...
 end;
```



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

Mientras el único objeto que sepa responder a la emisión *cmBuscarVentanaFicha* sea una ventana de agenda, el código puede asegurar que cuando finalice, habrá una y sólo una ventana de agenda al frente de las vistas en el *desktop*.

Para concluir el apartado de eventos en Turbo Vision, cuando se desee que un evento sea manejado sólo por una vista o un conjunto de vistas determinado se puede crear o modificar un evento, e invocar después directamente a *HandleEvent*. La llamada se realizará de una de las siguientes formas:

1. Se puede hacer que una vista invoque al *HandleEvent* de una vista compañera<sup>63</sup> directamente. El evento no se propagará a otras vistas. Va directamente al otro *HandleEvent*, y después el control vuelve inmediatamente.
2. Se puede invocar al *HandleEvent* del propietario. El evento se propagaría hacia abajo por la cadena de vistas. (Si se está invocando al *HandleEvent* desde dentro del propio *HandleEvent*, éste último *HandleEvent* será invocado recursivamente). Después de que el evento sea manejado, el control volverá.
3. Se puede invocar al *HandleEvent* de una vista de una cadena de vistas diferente. El evento irá bajando por esa cadena de vistas. Después de ser manejado, el control volverá.

## 14.8 UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

En este apartado vamos a ver como utilizar de una forma práctica los tipos objeto más comunes de la jerarquía de Turbo Vision

### APLICACIONES

En el corazón de todo programa está un objeto aplicación como ya hemos visto en el ejemplo 14.1 del apartado **Bucle principal de procesamiento de eventos** y el apartado **Turbo Vision: un marco de aplicación en modo texto** de este mismo capítulo.

Un objeto aplicación (*TApplication* o descendiente) tiene dos papeles principales en una aplicación Turbo Vision. Por una parte es una vista que gestiona toda la pantalla, y por otra es un motor para la manipulación de eventos, que interactúa con el ratón, el teclado, y otras partes del ordenador.

---

<sup>63</sup> Las vistas "compañeras" o hermanas son subvistas con el mismo propietario o padre.

Los límites de una vista aplicación abarcan toda la pantalla, pero el objeto aplicación no es visible por sí mismo. Divide la pantalla en tres zonas diferentes y asigna una subvista a cada una para manejarlas. Por defecto, el objeto aplicación asigna un objeto barra de menús en la primera línea de la pantalla, un objeto línea de estado en la última línea, y un objeto desktop en las líneas intermedias.

El bloque principal de una aplicación Turbo Vision siempre consta de tres sentencias, que llaman a los tres métodos principales del objeto aplicación: *Init*, *Run* y *Done*, como se muestra en el listado del ejemplo 14. que muestra el objeto aplicación del programa AGENDA.

#### Ejemplo 14.10

```
TGestionFichas = object(TApplication)
 VentanaClipboard: PEditWindow;
 VentanaFicha: PVentanaFicha;
 Reloj: PVistaReloj;
 Heap: PVistaDeHeap;
 constructor Init;
 . . .
end;
. . .

var
 Agenda: TGestionFichas;
begin
 Agenda.Init;
 Agenda.Run;
 Agenda.Done;
end.
```

La mayor parte del tiempo el objeto aplicación es la vista modal en la ejecución de una aplicación Turbo Vision. La única excepción se produce cuando se ejecuta otra vista (generalmente un cuadro de diálogo), la cual pasa a ser la vista modal en curso hasta que se llama a su método *EndModal*, y el objeto aplicación vuelve a ser modal.

La principal diferencia entre *TApplication* y su tipo ascendiente *TProgram* es que *TApplication* redefine el constructor y el destructor del objeto para inicializar y luego cerrar los cinco **subsistemas principales** que hacen que funcionen las aplicaciones Turbo Vision. Estos cinco subsistemas son:

- El gestor de memoria
- El gestor de vídeo
- El gestor de eventos
- El sistema manejador de errores
- El gestor de listas históricas

Turbo Vision inicializa cada subsistema llamando a un procedimiento de la unit *App*. El constructor *TApplication* llama a cada uno de ellos antes de hacer la llamada al constructor *Init* que hereda de *TProgram*:

```
constructor TApplication.Init;
begin
 InitMemory; { inicializa el gestor de memoria }
 InitVideo; { " el gestor de vídeo }
 InitEvents; { " el gestor de eventos }
 InitSysError; { " el sistema manejador de errores }
```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
InitHistory; { " el gestor de listas históricas }
inherited Init; { llama a TProgram.Init }
end;
```

Turbo Vision almacena el modo actual de pantalla en una variable bitmap denominada *ScreenMode*. *ScreenMode* contiene una combinación de las constantes de modo de pantalla *smMono*, *smBW80*, *smCO80*, y *smFont8x8*. Por defecto, una aplicación Turbo Vision asume el modo de pantalla que se estaba utilizando en el entorno DOS cuando se arrancó la aplicación. Si era un modo color de 25 líneas, ése es el que utiliza la aplicación Turbo Vision. Si era un modo texto VGA de 50 líneas, también comenzará en ese modo.

En la mayoría de los casos, no es necesario cambiar entre los modos monocromo, blanco y negro, y color, debido a que generalmente dependen del hardware del usuario. Habitualmente, se cambiará entre un modo normal de 25 líneas y un modo de alta resolución de 43 o 50 líneas. Para hacer ésto, se debe cambiar el bit *smFont8x8* en *ScreenMode* llamando a *SetScreenMode*. En el apartado **Gestión de subvistas. Redimensionar un subvista** muestra parte del método **HandleEvent** del objeto aplicación *TGestionFichas* que responde a un comando *cmVideo* para el cambio a un modo de fuente de 8x8 pixels. En este apartado, en la figura 14.31 se pudo observar el resultado del cambio de modo de vídeo.

*TApplication* proporciona un mecanismo sencillo para que los usuarios lancen un shell al DOS por medio del método **DosShell**. Este método cierra cualquier subsistema del objeto aplicación antes de lanzar el shell, más tarde los reinicializa cuando el usuario finaliza el mismo. El intérprete de comandos que usa el shell es el especificado en la variable de entorno *COMSPEC*.

Antes de ejecutar el intérprete de comandos, *DosShell* llama un método virtual denominado *WriteShellMsg* para visualizar el siguiente mensaje:

```
Type EXIT to return...
```

Se puede personalizar el mensaje redefiniendo *WriteShellMsg* para visualizar otro texto, como se muestra en el ejemplo 14.11. Se utiliza el procedimiento *PrintStr* en vez de *Writeln* para evitar linkar código innecesario.

### Ejemplo 14.11

```
TGestionFichas = object(TApplication)
 . . .
 procedure WriteShellMsg; virtual;
end;
 . . .

procedure TGestionFichas.WriteShellMsg;
begin
 PrintStr(' Salida temporal de AGENDA a DOS.' + #13 + #10 +
 ' Teclee EXIT para volver a AGENDA. ');
end;
```

El bucle de eventos del objeto aplicación llama a un método virtual denominado *Idle* siempre que no encuentre eventos pendientes en la cola de eventos. Esto significa que se puede usar Turbo Vision para lanzar procesos background cuando no está atendiendo a entradas de usuario. En el ejemplo 14.9 del apartado **Modificación del mecanismo de eventos** de este capítulo se presenta un ejemplo de utilización del tiempo muerto para visualizar la hora y el espacio disponible de memoria heap.

## DESKTOP

El desktop por defecto abarca la pantalla completa, menos la primera y la última línea, y sabe como gestionar las ventanas y los cuadros de diálogo que se encuentren en él. En algunos casos puede hacer falta cambiar su tamaño o posición, o la trama por defecto del fondo.

Los objetos aplicación llaman a un método virtual denominado *InitDesktop* que construye un objeto desktop y lo asigna a la variable global *Desktop*. Por defecto, *InitDesktop* obtiene el rectángulo delimitador del objeto aplicación y construye una vista desktop de tipo *TDesktop* que abarca toda la vista aplicación, menos la primera y la última línea. Para construir un desktop que abarque un área distinta, se necesita redefinir *InitDesktop*.

En casi todos los casos, los objetos ventana y cuadro de diálogo de una aplicación pertenecen al desktop. Debido a que el desktop es un grupo, se pueden usar los métodos *Insert* y *Execute* habituales para insertar, vistas no modales y modales, respectivamente. Sin embargo el objeto aplicación ofrece una manera mejor, y más segura de gestionar la inserción y la ejecución utilizando los métodos *InsertWindow* para ventanas no modales y *ExecuteDialog* para ventanas modales (típicamente cuadros de diálogo).

Los objetos desktop saben como distribuir sus ventanas de dos maneras diferentes: distribución en *mosaico* y en *casca*. La primera distribuye y redimensiona las ventanas como baldosas, de esta manera ninguna se solapa. La distribución en cascada distribuye las ventanas en orden descendente desde la esquina superior izquierda del desktop. La primera ventana abarca el desktop completo, la siguiente se mueve un espacio a la derecha y abajo, y así sucesivamente. El resultado es una pila de ventanas que bajan en cascada por el desktop, quedando visibles la barra de título y el lado izquierdo de cada una de ellas. La distribución en mosaico y en cascada es realizada por los métodos de *TApplication Tile* y *Cascade*.

El objeto desktop posee una vista por defecto, incluso antes de que se inserte cualquier ventana, la vista **fondo**. Es una vista muy simple que no hace nada, pero se visualiza a si misma en cualquier parte no cubierta del desktop. En el orden-Z, el fondo está por detrás de todas las demás vistas, y debido a que no es seleccionable, no se puede desplazar. El desktop almacena un puntero a su vista fondo en un campo denominado *Background*.

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

El objeto desktop por defecto visualiza un único carácter repetidamente sobre toda su área. Cambiar ese único carácter es sencillo. Cambiar el fondo para que dibuje más de carácter es algo más complicado. En la figura 14.36 se presentan dos desktops con fondos distintos. El de la derecha corresponde al generado con el programa del ejemplo 14.12, que se encuentra en el fichero FONDO1.PAS del directorio OTROS de los ejemplos del capítulo 14. El de la izquierda, está generado por el programa FONDO3.PAS del mismo directorio. Este fondo es muy similar al de la aplicación AGENDA que se presenta en el ejemplo 14.13.

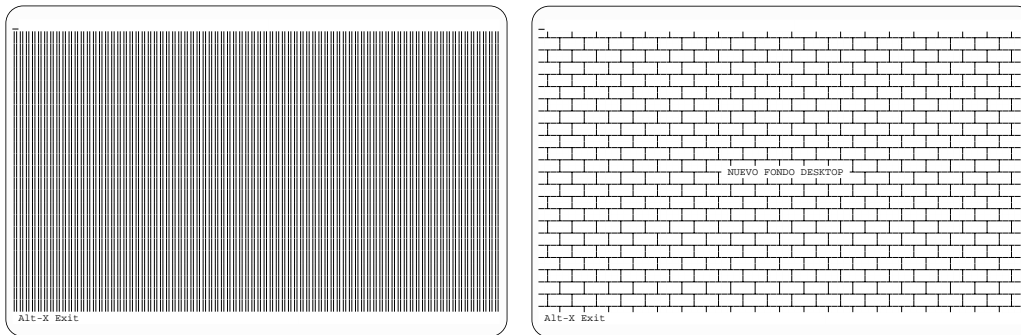


Fig. 14.36 Modificación del fondo del desktop.

La manera más sencilla de cambiar el carácter de trama del fondo es esperar hasta que el desktop cree su fondo por defecto. Entonces se puede cambiar el campo `Pattern` del objeto fondo, que almacena el carácter que se repite. El ejemplo 14.12 cambia el carácter del fondo por defecto por el carácter `ascii #186`.

### Ejemplo 14.12

```
program Fondo;
uses Objects, Drivers, Views, App;

type
 TAplicacionFondo = object(TApplication)
 procedure InitDesktop; virtual;
 end;

procedure TAplicacionFondo.InitDesktop;
begin
 inherited InitDesktop;
 Desktop^.Background^.Pattern := #186;
end;

var
 FondoApl: TAplicacionFondo;
begin
 FondoApl.Init;
 FondoApl.Run;
 FondoApl.Done;
end.
```

El valor inicial del carácter de trama del fondo se pasa como parámetro al constructor de la vista fondo, el cual es llamado por el método virtual *InitBackground* del objeto desktop. Si se deriva el objeto desktop, se puede redefinir *InitBackground*<sup>64</sup> para que pase el carácter deseado cuando se construye el fondo, en vez de hacerlo más tarde. Sin embargo, debido a que la única razón por la que se definiría un nuevo objeto desktop es para crear un fondo más complejo, se puede modificar el carácter de fondo sin redefinir el desktop. Esto se consigue poniendo un nuevo valor en el campo *Pattern* del objeto *Background* (de tipo *TBackground*) que posee todas las instancias de *TDesktop* desde dentro del método virtual *InitDesktop* de *TApplication*. Esta es la opción adoptada en el ejemplo 14.12.

Visualizar un fondo con un diseño que contenga más de un carácter requiere derivar dos nuevos objetos: un objeto fondo que se visualice a sí mismo de la forma que se quiera, y un objeto desktop que utilice el fondo creado en vez del estándar de *TBackground*.

La *unit* FondoMuro<sup>65</sup> del ejemplo 14.13 implementa un tipo objeto fondo *TFondoMuro* que repite dos cadenas de manera alternativa sobre las filas pares e impares de la superficie que ocupa. Estas dos cadenas *Linea1* y *Linea2* se construyen a partir de la repetición de dos cadenas más pequeñas de 4 caracteres (*muro\_impar* para *Linea1* y *muro\_par* para *Linea2*) de forma que se cubre el ancho del fondo (*size.Y*).

La clave del tipo objeto fondo *TFondoMuro* es su método *Draw*. Este método, llamado cada vez que alguna vista de la aplicación es cerrada o desplazada, lleva alternativamente sobre el buffer de la vista fondo los caracteres de sus campos cadena de caracteres *Linea1* y *Linea2*. Para ello se utiliza el procedimiento *WriteLine*. Además presenta una cadena almacenada en el campo *Texto* en el centro de la superficie de la vista fondo. Esta cadena es la que se pasa al constructor junto con los límites que ocupará la vista.

### Ejemplo 14.13

```
unit FondoMuro;

interface

type
PFondoMuro = ^TFondoMuro;
TFondoMuro = object(TBackground)
 Texto: TTitleStr;
 Linea1, Linea2: TTitleStr;
 constructor Init(var Limites: TRect; TextoFondo: TTitleStr);
 constructor Load(var S: TStream);
 procedure Draw; virtual;
```

---

<sup>64</sup> *InitBackground* es un método virtual que inicializa el campo **Background** del desktop con un objeto del tipo objeto *TBackground* o un descendiente, llamando al constructor *TBackground.Init* (si no es un descendiente) al cual se le pasan como argumentos los límites del fondo y el carácter de relleno:

```
constructor Init(Limites: TRect; C: Char); virtual;
```

<sup>65</sup> La *Unit FondoMuro* se puede encontrar en el fichero *FONDOMUR.PAS* del directorio *TPU* del capítulo 14.

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
 procedure Store(var S: TStream);
 end;
. . .

implementation

Const
 { Constantes de definición de fondo }
 muro_par = #196#194#196#193;
 muro_impar = #196#193#196#194;

constructor TFondoMuro.Init(var Limites: TRect; TextoFondo: TTitleStr) ;
begin
 inherited Init(Limites, ' ');
 Texto := TextoFondo;
 Linea1:= muro_impar;
 while Length(Linea1) < SizeOf(TTitleStr)-4 do
 Linea1 :=Linea1 + muro_impar;
 Linea2:= muro_par;
 while Length(Linea2) < SizeOf(TTitleStr)-4 do
 Linea2 :=Linea2 + muro_par;
end;
. . .

procedure TFondoMuro.Draw;
var
 BufferFondo: TDrawBuffer;
 i: integer;
 centroX, centroY, AnchoTexto :integer;
begin
 for i:=1 to size.Y do
 begin
 if odd(i) then
 MoveStr(BufferFondo, Linea1, GetColor(1))
 else
 MoveStr(BufferFondo, Linea2, GetColor(1));
 WriteLine(0, i-1, size.X, 1, BufferFondo);
 end;
 centroX := size.X Div 2;
 centroY := size.Y Div 2;
 AnchoTexto := Length(Texto) Div 2;
 WriteStr(CentroX - AnchoTexto, centroY, Texto, GetColor(2));
 . . .
 end.
```

Este tipo objeto *TFondoMuro* es utilizado en el programa AGENDA (ejemplo 14.14) para derivar un nuevo tipo desktop *TDesktopConFondo* que inicializa su fondo por medio del método *InitBackground* haciendo uso de una instancia de *TFondoMuro*. Para disponer del nuevo tipo desktop en la aplicación sólo resta por asignar a la variable global de la aplicación *Desktop* una instancia del nuevo tipo *TDesktopConFondo* redefiniendo el método *InitDesktop*.

### Ejemplo 14.14

```
const
 TituloFondoAplicacion = ' Aquí puede ir la publicidad de su empresa ' ;
. . .

{ TDesktopConFondo }
PDesktopConFondo = ^TDesktopConFondo;
TDesktopConFondo = object(TDesktop)
 procedure InitBackground; virtual;
end;
. . .
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
procedure TDesktopConFondo.InitBackground;
var
 R: TRect;
begin
 GetExtent(R);
 Background := New(PFondoMuro, Init(R, TituloFondoAplicacion));
end;
. . .

procedure TGestionFichas.InitDesktop;
var
 R: TRect;
begin
 GetExtent(R);
 R.Grow(0, -1);
 Desktop := New(PDesktopConFondo, Init(R));
end;
```

## MENUS

Un menú tiene dos partes: una *lista menú* que almacena las descripciones de los elementos del menú y los comandos que generan, y una *vista menú* que muestra esos elementos en la pantalla.

Las vistas menú pueden ser de dos clases: *barras de menús* y *cuadros de menú*. Ambas vistas utilizan las mismas listas de elementos de menú. De hecho, los mismos elementos se pueden visualizar o bien en una barra o en un cuadro. La diferencia principal es que una barra sólo puede ser un menú con un único nivel, generalmente localizado permanentemente en la primera línea de la pantalla de la aplicación. Un cuadro de menú puede ser o bien el menú principal (generalmente un menú desplegable, o uno local) o más a menudo un submenú desplegado por un elemento de una barra de menús o de otro cuadro de menú.

El constructor *Init* de un objeto aplicación llama a un método virtual denominado *InitMenuBar* para construir una barra de menús y asignarla a la variable *MenuBar*. Para definir una barra de menús personalizada, se necesita redefinir *InitMenuBar* que creará una barra de menús especial y la asignará a *MenuBar*.

Las barras de menús casi siempre ocupan la primera línea de la pantalla de la aplicación. Para asegurarse de que la barra de menús abarca esta primera línea lo más adecuado es establecer sus límites basándose en los de la aplicación. A diferencia de las barras, los cuadros de menú ajustan sus límites para acomodar sus contenidos, de esta manera no hay que preocuparse de establecer los tamaños de cada submenú. Sólo se deben establecer los límites de la barra de menús, los objetos menú se ocupan del resto.

El sistema de menú utiliza dos clases distintas de registros para definir una estructura de menú. Cada tipo de registro se diseña para su uso en una lista enlazada, con un campo puntero al siguiente registro.



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
TMenuItem = record
 Next: PMenuItem;
 Name: PString;
 Command: Word;
 Disabled: Boolean;
 KeyCode: Word;
 HelpCtx: Word;
 case Integer of
 0: (Param: PString);
 1: (SubMenu: PMenu);
end;

TMenu = record
 Items: PMenuItem;
 Default: PMenuItem;
end;
```

- **TMenu** define una lista de elementos de menú y almacena el elemento por defecto, o el seleccionado. Cada menú principal y cada submenú almacenan un registro *TMenu*. La lista de elementos es una lista enlazada de registros *TMenuItem*.
- **TMenuItem** define el texto, *hot key*, el comando, y el contexto de ayuda de un elemento de menú. Cada elemento, sea un comando o un submenú, posee su propio registro *TMenuItem*.

La función *NewItem* se utiliza para asignar e inicializar un registro elemento de menú. Se pueden crear listas de elementos por medio de llamadas anidadas a *NewItem*.

La función *NewSubMenu* permite crear submenús. Un *submenú* es un elemento de menú que despliega otro menú en vez de generar un comando. Las dos diferencias entre *NewSubMenu* y *NewItem* son:

El submenú no tiene comando asociado. De esta manera *NewSubMenu* pone el campo *Command* del elemento a cero, y no hay *hot key* asignada o definida.

Además de apuntar al siguiente elemento de su menú, el submenú apunta un registro *TMenu*, el cual contiene la lista de elementos del mismo.

El campo *Param* de *TMenuItem* se utiliza para almacenar la cadena que describe la *hot key* de un comando. Si el elemento de menú es un submenú se utiliza el campo *SubMenu* que apunta a un registro *Tmenu* con su lista de elementos.

En el ejemplo 14.15 se construye una sencilla barra de menús con un sólo menú **Menú 1** y dentro de éste una sólo opción o elemento de menú **Opción 1** que ejecuta el comando 100, no tiene *hot key* (cadena vacía '' y no se asigna ningún código de tecla 0) y se le asigna la constante de elemento sin contexto de ayuda *hcNoContext*.

### Ejemplo 14.15.

```
Uses App, Objects, Menus, Views;

Type
 TMiAplicacion = object(TApplication)
 procedure InitMenuBar; virtual;
 end;

procedure TMiAplicacion.InitMenuBar;
var
 R: TRect;
begin
 GetExtent(R);
 R.B.Y := R.A.Y + 1;
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

MenuBar := New(PMenuBar, Init(R, NewMenu(
 NewSubMenu('Menú 1', hcNoContext, NewMenu(
 NewItem('Elección 1', '', 0, 100, hcNoContext,
 nil)),
 nil)
));
end;

var
Prueba: TMIAplicacion;
begin
Prueba.Init;
Prueba.Run;
Prueba.Done;
end.

```

La construcción de menús se complica cuando se definen niveles de submenús. La dificultad se encuentra en el anidamiento de llamadas a *NewMenu*, *NewSubMenu* y *NewItem* para construir las listas de los elementos de menú. En el ejemplo 14.16 se modifica el método *InitMenuBar* del ejemplo 14.15 para construir una barra de menú con dos menús, en el que el segundo a su vez tiene dos elementos que son submenús. La figura 14.37 presenta los menús que se generan.

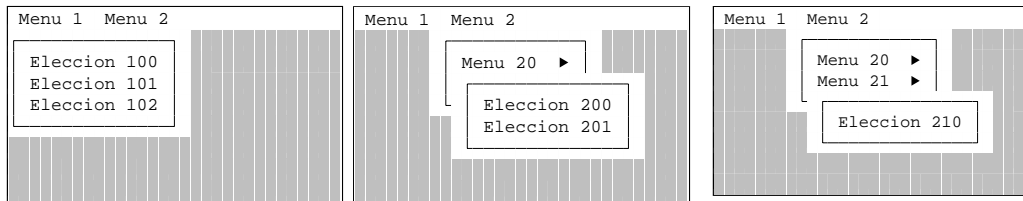


Fig. 14.37. Submenús y opciones del ejemplo 14.16.

### Ejemplo 14.16.

```

procedure TMIAplicacion.InitMenuBar;
var
R: TRect;
begin
GetExtent(R);
R.B.Y := R.A.Y + 1;
MenuBar := New(PMenuBar, Init(R,
 NewMenu(
 NewSubMenu('Menu 1', hcNoContext,
 NewMenu(
 NewItem('Eleccion 100', '', 0, 100, hcNoContext,
 NewItem('Eleccion 101', '', 0, 101, hcNoContext,
 NewItem('Eleccion 102', '', 0, 102, hcNoContext,
 nil)))
 { 1 "(" por cada NewItem }
),
 { fin de NewMenu }
),
 NewSubMenu('Menu 2', hcNoContext,
 NewMenu(
 NewSubMenu('Menu 20', hcNoContext,
 NewMenu(
 NewItem('Eleccion 200', '', 0, 200, hcNoContext,
 NewItem('Eleccion 201', '', 0, 201, hcNoContext,
 nil))
 { 1 "(" por cada NewItem }
),
 { fin de NewMenu }
),
 NewSubMenu('Menu 21', hcNoContext,
 NewMenu(
 NewItem('Eleccion 210', '', 0, 200, hcNoContext,
 nil)
 { 1 "(" por cada NewItem }
),
 { fin de NewMenu }
),
),
),
),
),
);
end;

```

UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```

 nil))
 },
 nil))
); { Finaliza Init y New }
end;
 { Finaliza NewMenu }
 { Finaliza NewMenu }
 { 1 "(" por cada NewSubMenu }
 { Finaliza NewMenu }
 { 1 "(" por cada NewSubMenu }
 { 1 "(" por cada NewSubMenu }

```

La ejecución del programa que contiene la barra de menús del ejemplo 14.16 crea cinco listas encadenadas como se puede apreciar en la figura 14.38, tantas como menús y submenús se han definido con las cinco llamadas realizadas a la función `NewMenu`.

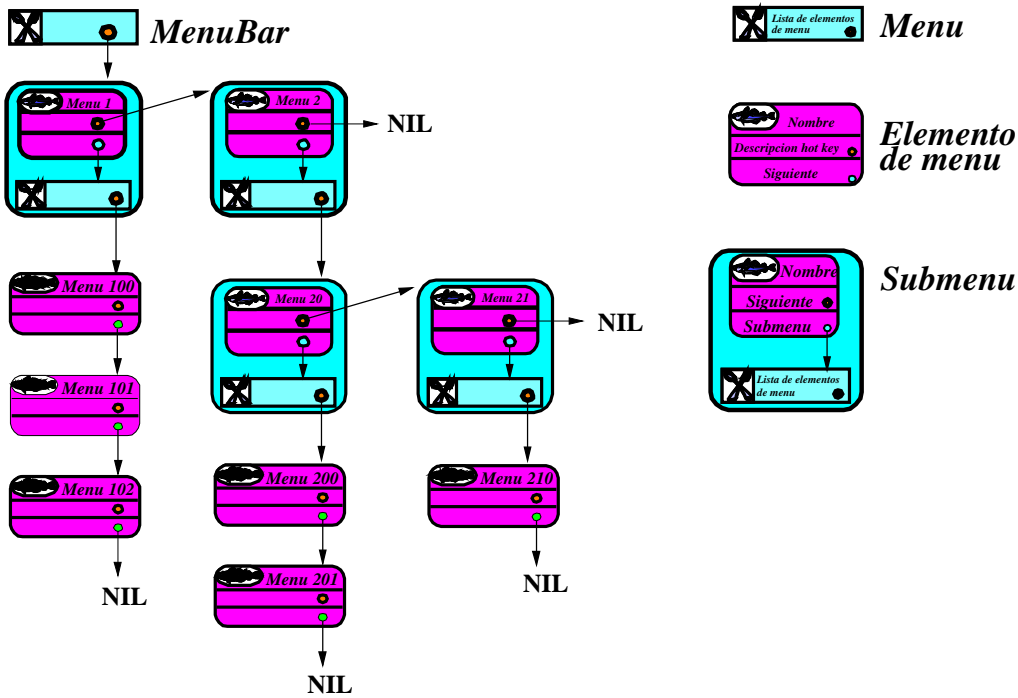


Fig. 14.38 Menús y listas de elementos de menús

**Ejemplo 14.17.**

```

{ Definición de los menús estándar }
function MenuArchivo(Mas: PMenuItem): PMenuItem;
begin
MenuArchivo :=
 NewItem('Archivo ~N~uevo', 'F4', kbF4, cmNew, hcNuevo,
 NewItem('~A~brir...', 'F3', kbF3, cmOpen, hcAbrir,
 NewItem('~G~uardar', 'F2', kbF2, cmSave, hcGuardar,
 NewItem('G~uardar como...', '', kbNoKey, cmSaveAs, hcGuardarComo,
 NewLine(
 NewItem('shell del ~D~OS', '', kbNoKey, cmDosShell, hcShell-
 Dos,
 NewItem('~S~alir', 'Alt+S', kbAltS, cmQuit, hcSalir,
 Mas))))));
end;

```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

function MenuEdicion(Mas: PMenuItem): PMenuItem;
begin
 MenuEdicion :=
 NewItem('~D~eshacer', 'Alt+Back', kbAltBack, cmUndo, hcDeshacer,
 NewLine(
 NewItem('Cor~t~ar', 'Shift+Supr', kbShiftDel, cmCut, hcCortar,
 NewItem('~C~opiar', 'Ctrl+Ins', kbCtrlIns, cmCopy, hcCopiar,
 NewItem('~P~egar', 'Shift+Ins', kbShiftIns, cmPaste, hcPegar,
 NewItem('~L~impiar', 'Ctrl+Supr', kbCtrlDel, cmClear, hcLimpiar,
 Mas))))));
end;

function MenuVentana(Mas: PMenuItem): PMenuItem;
begin
 MenuVentana :=
 NewItem('~M~osaico', '', kbNoKey, cmTile, hcMosaico,
 NewItem('Casca~d~a', '', kbNoKey, cmCascade, hcCascaada,
 NewLine(
 NewItem('Mo~v~er/Tamaño', 'Ctrl+F5', kbCtrlF5, cmResize, hcRedimensio-
nar,
 NewItem('~A~mpliar', 'F5', kbF5, cmZoom, hcAmpliar,
 NewItem('~S~iguiente', 'F6', kbF6, cmNext, hcSiguiete,
 NewItem('~P~revia', 'Shift+F6', kbShiftF6, cmPrev, hcPrevia,
 NewItem('~C~errar', 'Alt+F3', kbAltF3, cmClose, hcCerrar,
 Mas))))));
end;

var
 . . .
 MenuPrincipal: PMenuBar;
 . . .

begin
 { Definición del Menú Principal }
 R.Assign(0, 0, 80, 1);
 New(MenuPrincipal, Init(R, NewMenu(
 NewSubMenu('~A~rchivo', hcArchivo, NewMenu(
 MenuArchivo(nil)),
 NewSubMenu('~E~ditar', hcEditar, NewMenu(
 MenuEdicion(
 NewLine(
 NewItem('~M~ostrar clipboard', '', kbNoKey, cmMostrarClip, hcMostrar-
Clip,
 nil))))),
 NewSubMenu('A~g~enda', hcMAgenda, NewMenu(
 NewItem('~N~ueva Ficha', 'F9', kbF9, cmNuevaFicha, hcNuevaFicha,
 NewItem('~G~uardar', '', kbNoKey, cmSalvarFicha, hcAGuardar,
 NewLine(
 NewItem('Siguiete', 'AvPg', kbPgDn, cmSigFicha, hcASiguiete,
 NewItem('Anterior', 'RePg', kbPgUp, cmAntFicha, hcAPrevia,
 nil)))))),
 NewSubMenu('~O~pciones', hcOpciones, NewMenu(
 NewItem('Cambiar modo ~v~ideo', '', kbNoKey, cmOpcionesVideo, hcVi-
deo,
 NewItem('Cambiar ~c~olores', '', kbNoKey, cmColor,
hcColores,
 NewItem('Ra~t~ón', '', kbNoKey, cmRaton, hcRa-
ton,
 NewLine(
 NewItem('~S~alvar desktop...', '', kbNoKey, cmOpcionesSalvar, hcSal-
varDesktop,
 NewItem('~R~ecuperar desktop...', '', kbNoKey, cmOpcionesCargar,
hcRecuperarDesktop,
 nil)))))),
 NewSubMenu('~V~entana', hcVentana, NewMenu(
 NewItem('~A~genda', '', kbNoKey, cmVentanaFicha, hcVAgenda,

```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
NewItem('~C~calculadora', '', kbNoKey, cmCalculadora, hcVCalculadora,
NewItem('Ca~l~endario', '', kbNoKey, cmCalendario, hcVCalendario,
NewLine(
MenuVentana(nil))))) ,
NewSubMenu('Ay~u~da', hcAyuda, NewMenu(
NewItem('~I~ndice', 'Shift+F1', kbShiftF1, cmIndiceAyuda, hcIndice,
NewLine(
NewItem('~A~cerca de este Programa', '', kbNoKey, cmAcerca, hcAcerca,
nil))))) ,
nil))))) ;
```

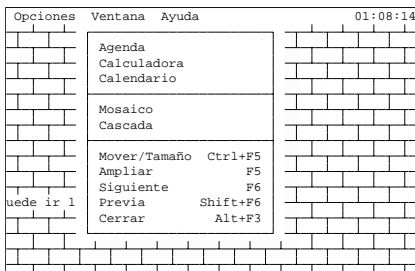


Fig. 14.39 Menú de ventanas

El ejemplo 14.17 construye los menús del programa AGENDA. Este ejemplo es un fragmento del programa RECURSOS.PAS, que genera algunos de los objetos que se utilizarán en AGENDA. La función `NewLine` que aparece en el listado permite realizar una separación meramente visual mediante una franja representada entre dos componentes de la lista de elementos de un menú. Se puede ver un ejemplo de su utilización en la figura 14.39.

## LINEAS DE ESTADO

El constructor por defecto del objeto aplicación llama a un método virtual `InitStatusLine` para construir e inicializar el objeto línea de estado. Para crear una línea de estado personalizada, se necesita redefinir `InitStatusLine` para construir un nuevo objeto línea de estado y asignarlo a la variable global `StatusLine`. La línea de estado cumple tres funciones importantes en la aplicación:

- Mostrar los comandos que el usuario puede pinchar con el ratón.
- Vincular *hot keys* a los comandos.
- Proporcionar al usuario indicaciones sensibles al contexto.

Las dos primeras se establecen cuando se construye el objeto línea de estado. La tercera la estudiaremos más adelante en el apartado **Ayuda sensible al contexto**.

El constructor del objeto línea de estado tiene dos parámetros: el rectángulo delimitador y un puntero a una lista enlazada de definiciones de estado. Una *definición de estado* es un registro que almacena un rango de contextos de ayuda y la lista de claves de estado que la línea de estado visualiza cuando el contexto de ayuda actual de la aplicación se encuentra dentro de este rango. Las *claves de estado* son registros que almacenan comandos, cadenas de texto y las *hot keys* que generan los comandos.

La función `NewStatusDef` permite crear los registros definición de estado. La creación de una lista enlazada de registros se realiza por medio de llamadas anidadas a `NewStatusDef`. Esta función tiene cuatro parámetros:

- El límite inferior del rango de contextos de ayuda
- El límite superior del rango de contextos de ayuda

- Un puntero a la lista enlazada de claves de estado
- Un puntero al siguiente registro de definición de estado, si lo hubiera

En la figura 14.40 se puede apreciar en la parte inferior de las dos pantallas las claves de estado en dos contextos distintos de la aplicación AGENDA.

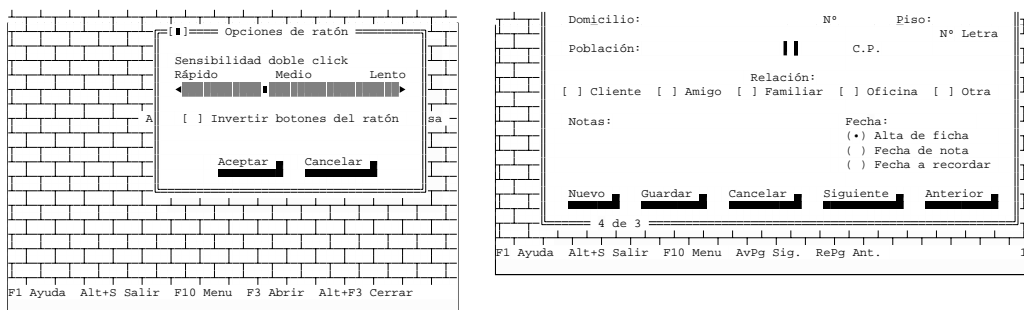


Fig. 14.40 Claves de estado en diferentes contextos de ayuda

Cada una de las definiciones de estado inicializadas en un programa necesita una lista de claves de estado. Un registro clave de estado consta de cuatro campos:

- Una cadena de texto que aparece en la línea de estado.
- Un *código scan* del teclado para una *hot key*.
- Un comando a generar.
- Un puntero al siguiente registro clave de estado, si lo hubiera.

La función *NewStatusKey* permite crear una lista de claves de estado realizando llamadas anidadas. En el ejemplo 14.18 se utiliza para implementar una función *TeclasEstado* que se podrá utilizar luego para definir claves de estado en distintos contextos que tengan en común las claves que define dicha función.

#### Ejemplo 14.18.

```
{ Definición de línea de estado estándar }
function TeclasEstado(Mas: PStatusItem): PStatusItem;
begin
 TeclasEstado :=
 NewStatusKey('', kbAltS, cmQuit,
 NewStatusKey('', kbF10, cmMenu,
 NewStatusKey('', kbAltF3, cmClose,
 NewStatusKey('', kbF5, cmZoom,
 NewStatusKey('', kbCtrlF5, cmResize,
 NewStatusKey('', kbF6, cmNext,
 NewStatusKey('', kbShiftF6, cmPrev,
 NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
 Mas))))))));
end;
```

El ejemplo 14.19 lista la definición de la línea de estado del programa AGENDA que contiene cinco rangos de contextos de ayuda.

#### Ejemplo 14.19.

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
var
 . . .
 LineaDeEstado: PLineaEstado;
 . . .

begin
 . . .
 { Definición de la línea de estado para cada contexto }
 R.Assign(0, 24, 80, 25);
 New(LineaDeEstado, Init(R,
 NewStatusDef(0, 200,
 TeclasEstado(nil),
 NewStatusDef(201, 400,
 TeclasEstado(
 NewStatusKey('~Alt+S~ Salir', kbAltS, cmQuit,
 NewStatusKey('~F10~ Menu', kbF10, cmMenu,
 NewStatusKey('~F3~ Abrir', kbF3, cmOpen,
 NewStatusKey('~Alt+F3~ Cerrar', kbAltF3, cmClose,
 nil))))),
 NewStatusDef(401, 600,
 TeclasEstado(
 NewStatusKey('~Alt+S~ Salir', kbAltS, cmQuit,
 NewStatusKey('~F10~ Menu', kbF10, cmMenu,
 NewStatusKey('~F9~ Nueva ficha', kbF9, cmNuevaFicha,
 NewStatusKey('~F4~ Nuevo archivo', kbF4, cmNew,
 nil))))),
 NewStatusDef(601, 700,
 TeclasEstado(
 NewStatusKey('~Alt+S~ Salir', kbAltS, cmQuit,
 NewStatusKey('~F10~ Menu', kbF10, cmMenu,
 NewStatusKey('~AvPg~ Sig.', kbPgDn, cmSigFicha,
 NewStatusKey('~RePg~ Ant.', kbPgUp, cmAntFicha,
 nil))))),
 NewStatusDef(700, $FFFF,
 TeclasEstado(
 NewStatusKey('~Alt+S~ Salir', kbAltS, cmQuit,
 NewStatusKey('~F10~ Menu', kbF10, cmMenu,
 NewStatusKey('~F3~ Abrir', kbF3, cmOpen,
 NewStatusKey('~Alt+F3~ Cerrar', kbAltF3, cmClose,
 nil))))),
 nil))))));
```

## VENTANAS

Los tipos objeto ventana son vistas grupo especializadas que permiten instanciar las ventanas (con título, solapables y con marco distintivo) que las aplicaciones Turbo Vision tienen en el desktop. En la figura 14.41 se pueden ver ventanas típicas insertadas en el desktop de la aplicación AGENDA.

El constructor del objeto ventana por defecto lleva tres parámetros: un rectángulo delimitador, una cadena para el título, y un número de ventana. La ventana por defecto crea una vista grupo con esos límites, hace que su campo título apunte a una copia de la cadena título de los parámetros, y establece su estado y sus *flags* de opción que le proporcionan una sombra y la hacen seleccionable. Como se puede observar en el ejemplo 14.20 se pueden derivar tipos ventana cuyo constructor requiera un número de parámetros y tipo distinto.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION



Fig. 14.41 Ejemplos de ventanas

### Ejemplo 14.20

```

procedure Calculadora;
var
 P: PCalculator;
begin
 P := New(PCalculator, Init);
 P^.HelpCtx := hcCalculadora;
 InsertWindow(P);
end;

procedure TGestionFichas.NuevaVentana;
var
 R: TRect;
 Ventana: PEditWindow;
begin
 R.Assign(0, 0, 60, 20);
 Ventana := New(PEditWindow, Init(R, '', wnNoNumber));
 Ventana^.HelpCtx := hcEditor;
 InsertWindow(Ventana);
end;

```

### Ejemplo 14.21

```

{ Inicializar visor de reloj y heap }
GetExtent(R);
R.A.X := R.B.X - 9; R.B.Y := R.A.Y + 1;
Reloj := New(PVistaReloj, Init(R));
Insert(Reloj);

GetExtent(R);
Dec(R.B.X);
R.A.X := R.B.X - 9; R.A.Y := R.B.Y - 1;
Heap := New(PVistaDeHeap, Init(R));
Insert(Heap);

```

El objeto aplicación hereda un método denominado *InsertWindow* que lleva un objeto ventana como parámetro, y comprueba que la ventana es correcta antes de insertarla en el desktop. Usar *InsertWindow* en vez de insertar las ventanas directamente (con *Insert* como en el ejemplo 14.21) asegura que cualquier ventana del desktop ha pasado dos tests de validación, de esta manera se puede confiar razonablemente en la ausencia de problemas. Estos dos test son:

- Llama a *ValidView* para comprobar que la construcción de la ventana no provoque que el gestor de memoria penetre en el área de seguridad.



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

- Llama al método *Valid* del objeto ventana, pasando el parámetro *cmValid*, que devuelve *True* sólo si la ventana y todas sus subvistas fueron construidas correctamente.

Si ambos, *Valid* y *ValidView*, indican que la ventana es correcta, *InsertWindow* llama al método *Insert* del objeto desktop para que inserte la ventana. Si la ventana no supera los dos tests, *InsertWindow* no la inserta, la libera, y devuelve *False*.

En el ejemplo 14.22 se llama a *ValidView* antes de insertar la ventana con *InsertWindow*, ya que se precisa cambiar un campo de la ventana *VentanaClipboard* para ocultarla si se ha construido correctamente antes de que sea insertada en el desktop. El ejemplo 14.23 presenta un caso similar, el que se debe validar un desktop temporal *DesktopTemp* leído de un fichero de recursos. Sólo cuando nos hemos asegurado que el desktop leído es correcto, eliminaremos del desktop anterior (guardando antes el portapapeles) asignando a la variable *Desktop* el desktop leído (e insertando el portapapeles que teníamos).

### Ejemplo 14.22

```
Desktop^.GetExtent(R);
VentanaClipboard := New(PEditWindow, Init(R, '', wnNoNumber));
if ValidView(VentanaClipboard) <> nil then
begin
 VentanaClipboard^.Hide;
 InsertWindow(VentanaClipboard);
 Clipboard := VentanaClipboard^.Editor;
 Clipboard^.CanUndo := False;
end;
```

### Ejemplo 14.23

```
procedure TGestionFichas.CargarDesktop;
var
 FicheroDesktop: TBufStream;
 DesktopTemp: PDesktop;
 Marca: string[LongMarcaD];
 R: TRect;
begin
 FicheroDesktop.Init('AGENDA.DSK', stOpenRead, 1024);
 if LowMemory then OutOfMemory
 else if FicheroDesktop.Status <> stOk then
 MessageBox('No se puede abrir el fichero de desktop', nil, mfOkButton +
mfError)
 else
 begin
 Marca[0] := Char(LongMarcaD);
 FicheroDesktop.Read(Marca[1], LongMarcaD);
 if Marca = MarcaDesktop then
 begin
 DesktopTemp := PDesktop(FicheroDesktop.Get);
 if FicheroDesktop.Status <> stOk then
 MessageBox('Error al leer el fichero de desktop ', nil,
mfOkButton + mfError);
 end
 else
 MessageBox('Error: Fichero de desktop no válido.', nil,
mfOkButton + mfError);
 end;
 FicheroDesktop.Done;
 if ValidView(DesktopTemp) <> nil then
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
begin
 Desktop^.Delete(VentanaClipboard);
 Delete(Desktop);
 Dispose(Desktop, Done);
 Desktop := DesktopTemp;
 Insert(Desktop);
 GetExtent(R);
 R.Grow(0, -1);
 Desktop^.Locate(R);
 InsertWindow(VentanaClipboard);
 VentanaFicha := Message(Desktop, evBroadcast, cmBuscarFichaVentana, nil);
 if VentanaFicha <> nil then MostrarFicha(FichaActiva);
end;
end;
```

## CUADROS DE DIALOGO

Los cuadros de diálogo son ventanas especializadas. La principal diferencia con las ventanas está en que los objetos cuadro de diálogo tienen diferentes atributos por defecto, soporte para la operación modal, y adaptaciones para la gestión de objetos de control.

El método *ExecuteDialog* del objeto aplicación es muy parecido al *InsertWindow*. La diferencia está en que después de determinar la validez del objeto ventana, *ExecuteDialog* llama al método *Execute* del objeto desktop para hacer la ventana modal, en vez de insertarla. Como su propio nombre indica, *ExecuteDialog* se diseñó teniendo en mente los cuadros de diálogo, pero se puede pasar cualquier objeto ventana que se quiera hacer modal.

*ExecuteDialog* lleva además un segundo parámetro, un puntero a un buffer de datos usado por *GetData* y *SetData*. Si el puntero es **nil**, *ExecuteDialog* se salta los procesos *GetData/SetData*. Si no es **nil**, *ExecuteDialog* llama a *SetData* antes de ejecutar la ventana y llama a *GetData* si el usuario no canceló el cuadro de diálogo.

### Ejemplo 14.24





```
procedure Raton;
var
 D: PDialog;
begin
 D := New(PDialogoRaton, Init);
 D^.HelpCtx := hcDialogoRaton;
 ExecuteDialog(D, @MouseReverse);
end;
```

### Ejemplo 14.25

```
..
if ExecuteDialog(D, Application^.GetPalette) <> cmCancel then
begin
 DoneMemory; { Elimina todos los buffers de grupo }
 ReDraw; { Redibuja la aplicación con la nueva paleta }
end;
end;
```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

Los objetos cuadro de diálogo tienen dos métodos que perfilan su uso como vistas modales: *HandleEvent* y *Valid*. Gestionan la mayoría de los eventos como cualquier objeto ventana, pero realizan dos cambios que sólo se notarán al usarlos como vistas modales:

-  y el  son manejados de una manera especial.
-  envía un mensaje *cmDefault* al cuadro de diálogo, lo que equivale a haber pulsado el botón por defecto.  se transforma en un comando *cmCancel*.
- Ciertos comandos terminan automáticamente el estado modal.
- Los comandos *cmOk*, *cmCancel*, *cmYes*, y *cmNo* provocan llamadas a *EndModal*, pasando el comando como parámetro.

Los objetos cuadro de diálogo contienen habitualmente *controles*. Estos son vistas especializadas que permiten la interacción del usuario presionando botones, cuadros de lista, y barras de desplazamiento. Aunque se pueden insertar controles en un objeto ventana, los cuadros de diálogo están especialmente adaptados para ello.

La incorporación de controles a una ventana es igual que en cualquier subvista. Después de llamar al constructor de la ventana, se pueden construir e insertar objetos control, como se muestra en el listado del ejemplo 14.26.

### Ejemplo 14.26

```
var
 R: TRect;
 . . .
 VentanaFicha, CajaAbout: PDialog;
 Campo: PInputLine;
 Historia: PHistory;
 Cluster: PCluster;
 Memo: PMemo;

begin
 . . .

 { Dialogo "Acerca de ..." }
 R.Assign(0, 0, 50, 14);
 CajaAbout := New(PDialog, Init(R, 'Acerca de este programa'));
 with CajaAbout^ do
 begin
 Options := Options or ofCentered;
 R.Assign(2, 2, 48, 4);
 Insert(New(PStaticText, Init(R, #3'INTRODUCCION A LA PROGRAMACION'+
 ' ESTRUCTURADA'#13#3'Y ORIENTADA A OBJETOS CON PASCAL')));
 R.Assign(2, 5, 48, 7);
 Insert(New(PStaticText, Init(R, #3'Copyright (c) Noviembre 1993'#13#3+
 'Dpto. de Matemáticas. Universidad de Oviedo.')));
 . . .
 R.Assign(20, 12, 31, 14);
 Insert(New(PButton, Init(R, '~A~ceptar', cmOk, bfDefault)));
 end;

 { Ventana para las fichas de la agenda }
 R.Assign(0, 0, 66, 21);
 VentanaFicha := New(PDialog, Init(R, 'Agenda'));
 with VentanaFicha^ do
 begin
 Options := Options or ofCentered;
```

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

HelpCtx := hcAgenda;
 { Definición de campos de apellidos }
R.Assign(19, 2, 35, 3);
Campo := New(PInputLine, Init(R, 20));
Campo^.SetValidator(New(PValidadorNombre, Init));
Insert(Campo);
R.Assign(16, 2, 19, 3);
Insert(New(PLabel, Init(R, '1º', Campo)));
R.Assign(3, 2, 14, 3);
Insert(New(PLabel, Init(R, 'Apellido~s~:', Campo)));
R.Assign(40, 2, 56, 3);
Campo := New(PInputLine, Init(R, 20));
Campo^.SetValidator(New(PValidadorNombre, Init));
Insert(Campo);
R.Assign(37, 2, 40, 3);
Insert(New(PLabel, Init(R, '2º', Campo)));

 { Definición de las casillas de verificación de la relación del
 usuario con la persona de la ficha }
R.Assign(2, 11, 63, 12);
Cluster := New(PCheckBoxes, Init(R,
 NewSItem('Cliente',
 NewSItem('Amigo',
 NewSItem('Familiar',
 NewSItem('Oficina',
 NewSItem('Otra', nil)))))));
Insert(Cluster);
R.Assign(28, 10, 38, 11);
Insert(New(PLabel, Init(R, 'Re~l~ación:', Cluster)));
. . .

 { Definición de campo de notas }
R.Assign(2, 14, 40, 17);
Memo := New(PMemo, Init(R, nil, nil, nil, 255));
Insert(Memo);
R.Assign(3, 13, 10, 14);
Insert(New(PLabel, Init(R, 'Notas:', Memo)));
. . .

{ Definición de campo de fecha y tipo de fecha }
R.Assign(48, 13, 60, 14);
Campo := New(PInputLine, Init(R, 10));
Campo^.SetValidator(New(PValidadorFecha, Init));
Insert(Campo);
R.Assign(41, 13, 48, 14);
Insert(New(PLabel, Init(R, 'Fec~h~a:', Campo)));
. . .
 { Definición de botones de radio para indicar significado de fecha }
R.Assign(41, 14, 63, 17);
Cluster := New(PRadioButtons, Init(R,
 NewSItem('Alta de ficha ',
 NewSItem('Fecha de nota ',
 NewSItem('Fecha a recordar ', nil))));
Insert(Cluster);
. . .
. . .

. . .
 { definición de botones de control }
R.Assign(2, 18, 11, 20);
Insert(New(PButton, Init(R, '~N~uevo', cmNuevaFicha, bfNormal)));
R.Assign(12, 18, 23, 20);
Insert(New(PButton, Init(R, 'Gua~r~dar', cmSalvarFicha, bfDefault)));
R.Assign(24, 18, 36, 20);
Insert(New(PButton, Init(R, '~C~ancelar', cmCancelarFicha, bfNormal)));
R.Assign(37, 18, 50, 20);

```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
Insert(New(PButton, Init(R, 'Siguiente', cmSigFicha, bfNormal)));
R.Assign(51, 18, 63, 20);
Insert(New(PButton, Init(R, 'Anterior', cmAntFicha, bfNormal)));
SelectNext(False);
end;
```

La figura 14.42 presenta los diálogos construidos en el ejemplo 14.26.

Es necesario ser consciente del orden en el que se insertan los controles. Este orden establece el orden-Z de las vistas, el cual a su vez determina el *orden tab* de los controles. El orden tab es aquel en el cual los controles reciben el foco en una ventana cuando el usuario presiona **TAB**.

El orden tab es importante debido a que determina:

- El orden de la interacción del usuario
- El orden de la inicialización de controles

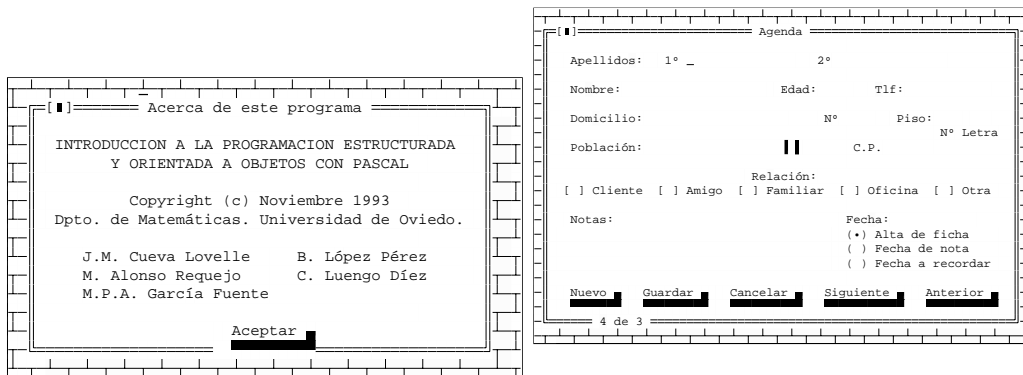


Fig. 14.42 Cuadros de diálogo

En cualquier momento después de haber construido un objeto ventana con controles, se pueden establecer o leer los valores de todos ellos usando los métodos *SetData* y *GetData*. Estos métodos son diferentes para los controles y para las otras vistas. Todos los grupos, incluyendo ventanas y cuadros de diálogo, heredan los métodos *GetData* y *SetData* que se repiten a través de sus subvistas en orden-Z, llamando a los métodos *GetData* o *SetData* de las mismas.

En el caso de una ventana con controles, la llamada a su *SetData* provoca la llamada al método *SetData* de cada control (ejemplo 14.27), en orden, así, en vez de tener que inicializar manualmente cada control, puede ser la ventana misma quien lo haga. El parámetro que se pasa a *SetData* es un registro que contiene un campo por cada control de la ventana.

Para definir un registro de datos para una ventana o para un cuadro de diálogo, hay que hacer lo siguiente:

- Listar cada control en orden-Z
- Determinar el registro de datos para cada control
- Crear un registro con un campo para cada control

Un método *SetData* del objeto ventana llama a los métodos *SetData* de cada una de sus subvistas en orden-Z. El registro de datos pasado a cada subvista es un subconjunto del que se le pasa al *SetData* de la ventana. El primer control en orden-Z coge todo el registro. Si lee varios bytes del registro (informado por su método *DataSize*), *SetData* pasa sólo el resto del registro a la siguiente subvista. Así, en el método *MostrarFicha* del ejemplo 14.27 el primer control de la ventana *VentanaFicha* (definida en el ejemplo 14.26) lee 20 bytes del registro *FichaInfo* de tipo *TFicha*, el *SetData* de la ventana da a la segunda subvista un registro que empieza 20 bytes después del original (2º apellido).

La lectura de los valores de los controles de un cuadro de diálogo se realiza de forma análoga al establecimiento de los mismos. El método *GetData* de los objetos cuadro de diálogo llama a *GetData* para cada subvista en orden-Z. Cada subvista tiene la oportunidad de escribir varios bytes (determinados por su método *DataSize*) en el registro de datos para el cuadro de diálogo.

### Ejemplo 14.27

```

type
 PFicha = ^TFicha;
 TFicha = record
 Apell1, Apell2: string[20];
 Nombre: string[25];
 Edad: shortint;
 Tlf: string[10];
 Domicilio: string[35];
 Portal: integer;
 Piso: shortint;
 Letra: char;
 Ciudad: string[20];
 CP: string[5];
 Relacion, LongNotas: Word;
 Notas: array[0..255] of Char;
 Fecha: string[10];
 TipoFecha: Word;
 end;

procedure TGestionFichas.SalvarDatosFicha;
begin
 if VentanaFicha^.Valid(cmClose) then
 begin
 VentanaFicha^.GetData(FichaInfo);
 if FichaActiva = ColecFichas^.Count then
 begin
 FichaTemporal^.RegisTransferencia := FichaInfo;
 ColecFichas^.Insert(FichaTemporal);
 VentanaFicha^.ContadorFichas^.IncContador;
 end
 else
 PObjetoFicha(ColecFichas^.At(FichaActiva))^RegisTransferencia :=
FichaInfo;
 SalvaFichas;
 MostrarFicha(FichaActiva);
 end;
end;

```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
procedure TGestionFichas.MostrarFicha(ANumFicha: Integer);
begin
 if ColecFichas^.Count > 0 then
 begin
 FichaActiva := ANumFicha;
 FichaInfo := PObjetoFicha(ColecFichas^.At(FichaActiva))^ .Re-
 gisTransferencia;
 with VentanaFicha^ do
 begin
 SetData(FichaInfo);
 ContadorFichas^.PonerActivo(FichaActiva + 1);
 end;
 if FichaActiva > 0 then EnableCommands([cmAntFicha])
 else DisableCommands([cmAntFicha]);
 if ColecFichas^.Count > 0 then
 EnableCommands([cmSigFicha]);
 if FichaActiva >= ColecFichas^.Count - 1 then
 DisableCommands([cmSigFicha]);
 EnableCommands([cmSalvarFicha, cmNuevaFicha]);
 DisableCommands([cmCancelarFicha]);
 end;
end;
```

Si el segundo parámetro para *ExecuteDialog* no es **nil**, la aplicación establece los valores iniciales de los controles del cuadro de diálogo y los lee cuando cierra el cuadro de diálogo modal.

El segundo parámetro para *ExecuteDialog* es un puntero a un registro de datos para el cuadro de diálogo. Como con todos los registros de datos de establecimiento y lectura de valores de control, el programador es el responsable de que el registro indicado incluya los datos en el orden correcto. Después de la construcción del cuadro de diálogo y de realizar las comprobaciones de validación, *ExecuteDialog* llama al método *SetData* de la ventana si el puntero al registro de datos no es **nil**. Cuando el usuario acaba con el estado modal de la ventana, *ExecuteDialog* lee los valores de los controles devueltos en el mismo registro llamando a *GetData*, a menos que el estado modal finalice mediante el comando *cmCancel*. En ese caso, la transferencia de datos no se lleva a cabo.

La tabla 14.6 muestra el tamaño de datos para cada uno de los controles estándar de Turbo Vision. Así en el ejemplo 14.27 el registro *TFicha* tiene como último campo *TipoFecha* de 2 bytes (*Word*), ya que el último control de *VentanaFicha* que ocupa memoria es un botón de radio *RadioButtons* que como se puede comprobar ocupa 2 bytes.

Los tipos de control de Turbo Vision son de propósito general, por lo que puede que no sean la mejor herramienta para una determinada aplicación. Se pueden derivar tipos de control que usen registros de datos más especializados para establecer y leer sus valores.

En el ejemplo 14.28 se define un tipo objeto de campo de edición para entradas numéricas, ya que no es muy eficiente tener que transferir toda una cadena hacia y desde el objeto. Es mucho más sensato usar un valor numérico. Este tipo de control se utiliza en la ventana de la agenda del programa *AGENDA* por ejemplo para la entrada y presentación de la edad de una persona.

| <b>Tipo de control</b>                         | <b>Tamaño de datos<br/>(en bytes)</b> | <b>Interpretación de los datos</b> |
|------------------------------------------------|---------------------------------------|------------------------------------|
| <i>Botón</i>                                   | 0                                     | Ninguna                            |
| <i>Casilla de verificación</i>                 | 2                                     | Bit por casilla                    |
| <i>Campo de edición</i>                        | <i>Maxlen + 1</i>                     | String Pascal                      |
| <i>Etiqueta</i>                                | 0                                     | Ninguna                            |
| <i>Cuadro de lista</i>                         | 6                                     | Puntero a lista y selección        |
| <i>Casilla de verificación<br/>multiestado</i> | 4                                     | Depende de los flags               |
| <i>Texto parametrizado</i>                     | <i>ParamCount * 4</i>                 | Parámetros a sustituir             |
| <i>Botón de radio</i>                          | 2                                     | Nº casilla seleccionada            |
| <i>Barra desplazamiento</i>                    | 0                                     | Ninguna                            |
| <i>Texto estático</i>                          | 0                                     | Ninguna                            |

Tabla 14.6 Tamaño de los registros de transferencia de datos para los objetos de control

### Ejemplo 14.28

```

PEntradaInt = ^TEntradaInt;
TEntradaInt = Object(TInputLine) { Entrada de enteros(2 bytes) }
 function DataSize: Word; virtual;
 procedure GetData(var Reg); virtual;
 procedure SetData(var Reg); virtual;
end;

{ TEntradaInt }
function TEntradaInt.DataSize: Word;
begin
 DataSize := SizeOf(integer);
end;

procedure TEntradaInt.GetData(var Reg);
var
 CodError : Integer;
begin
 Val(Data^, integer(Reg), CodError); { da valor a Reg desde Data }
end;

procedure TEntradaInt.SetData(var Reg);
begin
 Str(Integer(Reg), Data^); { da valor a Data desde Reg }
end;

```

### VALIDADORES DE CAMPO

Turbo Vision proporciona varias formas de validar la información que un usuario introduce en un campo de edición por medio de la asociación de objetos de validación con los objetos campo de edición.



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

La validación de datos es realizada por el método *Valid* de los objetos vista. Se pueden validar los contenidos de cualquier campo de edición o dato de la pantalla en cualquier momento llamando al método *Valid* del objeto, pero Turbo Vision también proporciona mecanismos para automatizar la validación.

Existen tres clases de validación de datos que no se excluyen entre sí:

- **Filtro de la entrada.** Permite restringir los caracteres a introducir por el usuario. Por ejemplo, un campo de entrada numérico puede limitar los caracteres admisibles sólo a los dígitos.
- **Validación de cada elemento.** Se garantiza que la entrada del usuario en un determinado campo es válida antes de pasar al siguiente. Eso se denomina generalmente *validación por Tab*, debido a que la manera habitual de desplazar el foco en una pantalla de entrada de datos es presionando la tecla **TAB**.
- **Validación de pantallas completas.** Se puede gestionar de tres formas diferentes:
  - *Validación de ventanas modales.* Una ventana modal valida automáticamente todas sus subvistas antes de cerrarse, a menos que el comando de cierre sea *cmCancel*.
  - *Validación sobre el cambio de foco.* Si se usa una ventana de entrada de datos no modal, se la puede forzar a que valide sus subvistas cuando pierda el foco, lo que se produce cuando se selecciona. La activación del flag *ofValidate* de una ventana evita que el usuario se desplace hacia otra antes de que los datos de entrada sean validados.
  - *Validación por demanda.* Se puede decir a una ventana que valide sus subvistas en cualquier momento llamando a su método *Valid*, pasando *cmClose* como parámetro.

Cada tipo objeto de validación hereda cuatro importantes métodos del tipo de objeto de validación abstracto *TValidator*. Redefiniendo estos métodos de diferentes maneras, los tipos objeto de validación descendientes realizan sus validaciones específicas.

Los cuatro métodos de validación son:

- **Valid.** Devuelve *True* si el método *IsValid* devuelve *True*; de lo contrario llama a *Error* para dar cuenta al usuario del error y devuelve *False*.
- **IsValid.** Sólo lleva una cadena como parámetro y devuelve *True* si los datos que contiene son válidos. *IsValid* es el método que realiza la validación, así que si se crean objetos de validación, se deberá redefinir *IsValid* la mayor parte de las veces.
- **IsValidInput.** Cuando un objeto campo de edición reconoce un evento de teclado que tiene significado para él, llama al método *IsValidInput* de su objeto de validación para garantizar que el carácter introducido es válido. Por defecto, el método *IsValidInput* siempre devuelve *True*. Algunos descendientes de los objetos de validación redefinen *IsValidInput* para deshechar las pulsaciones no deseadas.
- **Error.** Avisa al usuario que el contenido del campo de edición no ha superado el control de validación.

Los únicos métodos que se llaman desde fuera del objeto son *Valid* y *IsValidInput*. *Error* y *IsValid* son llamados sólo por otros métodos del objeto de validación.

Turbo Vision incluye cinco tipos de objetos de validación además de *TValidator*:

- Objeto de validación por filtro. **TFilterValidator**
- Objeto de validación por rango. **TRangeValidator** (descendiente del anterior)
- Objeto de validación por búsqueda. **TLookupValidator** (abstracto)
- Objeto de validación por búsqueda de cadena. **TStringLookupValidator** (descendiente del anterior)
- Objeto de validación por patrón. **TPXPictureValidator**

El ejemplo 14.29 lista la unit ValidaEntrada del fichero VALIDAENT.PAS del directorio TPU. Esta unit define validadores específicos para los campos de la ventana de fichas de agenda como teléfono, código postal, edad, ...

#### Ejemplo 14.29 validaent.pas

```
Unit ValidaEntrada;
interface
. . .

PValidadorEdad = ^TValidadorEdad;
TValidadorEdad = Object(TRangeValidator)
 constructor Init;
 procedure Error; virtual;
end;

PValidadorNombre = ^TValidadorNombre;
TValidadorNombre = Object(TPXPictureValidator)
 constructor Init;
 procedure Error; virtual;
end;

PValidadorTlf = ^TValidadorTlf;
TValidadorTlf = Object(TFilterValidator)
 constructor Init;
 procedure Error; virtual;
end;

PValidadorNum = ^TValidadorNum;
TValidadorNum = Object(TRangeValidator)
 constructor Init;
 procedure Error; virtual;
end;

PValidadorLetra = ^TValidadorLetra;
TValidadorLetra = Object(TPXPictureValidator)
 constructor Init;
 procedure Error; virtual;
end;

PValidadorCP = ^TValidadorCP;
TValidadorCP = Object(TPXPictureValidator)
 constructor Init;
 procedure Error; virtual;
end;

PValidadorFecha = ^TValidadorFecha;
TValidadorFecha = Object(TPXPictureValidator)
 constructor Init;
 function IsValid(const S: string): Boolean; virtual;
```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
 procedure Error; virtual;
 end;
 . . .

implementation

{ TValidadorEdad }
constructor TValidadorEdad.Init;
begin
 inherited Init(0, 120);
end;

procedure TValidadorEdad.Error;
type
 rango = record
 min, max: Longint;
 end;
var
 datos:rango;
begin
 datos.min:=Min;
 datos.max:=Max;
 MessageBox('Edad no válida. Edades admisibles entre %d y %d',
 @datos, mfError + mfOkButton);
end;

{ TValidadorNombre }
constructor TValidadorNombre.Init;
begin
 inherited Init('*@', False);
end;
procedure TValidadorNombre.Error;
begin
 MessageBox('Sólo se permiten caracteres alfabéticos.',
 nil, mfError + mfOkButton);
end;

{ TValidadorTlf }
constructor TValidadorTlf.Init;
begin
 inherited Init(['0'..'9', '-']);
end;
procedure TValidadorTlf.Error;
begin
 MessageBox('Formato de número de teléfono no válido. #13#3+
 'Se admite sólo dígitos y el carácter "-".',
 nil, mfError + mfOkButton);
end;
 . . .

{ TValidadorFecha }
function Bisiesto(anio: integer): Boolean;
begin
 Bisiesto:= (((anio mod 4) = 0) AND ((anio mod 100) <> 0)
 OR ((anio mod 400) = 0));
end;

function FechaValida(dia, mes, anio: integer): Boolean;
const
 DiasMes: array[1..12] of Byte =
 (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
var
 DiaMas: Byte;
begin
 if (mes in [1..12]) then
 begin
 if (mes=2) AND Bisiesto(anio) then
```

MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

 DiaMas:=1
 else DiaMas:=0;
 if (dia>=1) AND (dia<=(DiasMes[mes]+DiaMas)) then
 FechaValida:=True
 else
 FechaValida:=False;
 end
else
 FechaValida:=False;
end;

constructor TValidadorFecha.Init;
begin
 inherited Init('{#[#]}/{#[#]}/{##[##]}', True);
end;

function TValidadorFecha.IsValid(const S: string): Boolean;
var
 d, m, a: integer;
 cad1, cad2: string[12];
 err, desde,
 cuantos : integer;
begin
 if Not (TPXPictureValidator.IsValid(S)) then
 IsValid:=False
 else
 if Length(S)=0 then
 IsValid:=True
 else
 begin { Día }
 cuantos:=pos('/',S)-1;
 cad1:=copy(S,1,cuantos);
 val(cad1,d,err);
 if (err <> 0) then
 IsValid:=False
 else
 begin { Mes }
 desde:=pos('/',S)+1;
 cuantos:=Length(S)-desde+1;
 cad2:=copy(S,desde,cuantos);
 cuantos:=pos('/',cad2)-1;
 cad1:=copy(cad2,1,cuantos);
 val(cad1,m,err);
 if (err <> 0) then
 IsValid:=False
 else { Año }
 begin
 desde:=pos('/',cad2)+1;
 cuantos:=Length(cad2)-desde+1;
 cad1:=copy(cad2,desde,cuantos);
 val(cad1,a,err);
 if (err <> 0) then
 IsValid:=False
 else { Año }
 IsValid:=FechaValida(d,m,a);
 end;
 end;
 end;
 end;
 end;
 end;
 end;
 end;

procedure TValidadorFecha.Error;
begin
 MessageBox('Introducir fecha(dd/mm/aa) correcta.', nil, mfError + mfOkBut-
ton);
end;
. . .

```

UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

La figura 14.43.muestra el mensaje de error producido al validar un campo fecha con el validador TValidadorFecha de la unit del ejemplo anterior (*ValidaEntrada*).

Un *patrón* o plantilla describe el formato válido de la entrada en un validador por patrón.

Los patrones admisibles por *TPXPictureValidator* son compatibles con los que se utilizan en las bases de datos relacionales Paradox de Borland para controlar las entradas de usuario. La tabla 14.7 muestra el significado de los caracteres dentro de estos patrones.

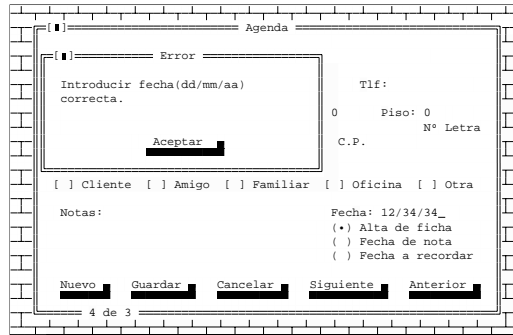


Fig. 14.43 Validación de una fecha

| Tipo de carácter   | Carácter | Descripción                                      |
|--------------------|----------|--------------------------------------------------|
| <i>Especial</i>    | #        | Acepta sólo un dígito                            |
|                    | ?        | Acepta sólo una letra                            |
|                    | &        | Acepta sólo una letra, pasándola a mayúscula     |
|                    | @        | Acepta cualquier carácter                        |
| <i>Combinación</i> | !        | Acepta cualquier carácter, pasándola a mayúscula |
|                    | ;        | Toma el siguiente carácter literalmente          |
|                    | *        | Repetición                                       |
|                    | []       | Opcional                                         |
|                    | {}       | Agrupamiento                                     |
| <i>Otros</i>       | ()       | Conjunto de alternativas                         |
|                    | ,        | Se toman literalmente                            |

Tabla 14.7 Caracteres de una plantilla en un validador por patrón

El uso de un objeto de validación de datos con un campo de edición se realiza mediante dos pasos:

- Construcción del objeto de validación
- Asignación del objeto de validación a un campo de edición

Todos los objetos campo de edición tienen un campo denominado *Validator*, asignado por defecto a **nil**, que puede apuntar a un objeto de validación. En el ejemplo 14.30 se utiliza el método *SetValidator* para de los campos de entrada para asignarles distintos tipos de validadores definidos en el ejemplo 14.29.

**Ejemplo 14.30**

```
var
 R: TRect;
 .
 .
 .
 VentanaFicha, CajaAbout: PDialog;
 Campo: PInputLine;
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
Historia:PHistory;
Cluster: PCluster;
Memo: PMemo;

begin
 . . .

 { Definición de campos de apellidos }
 R.Assign(19, 2, 35, 3);
 Campo := New(PInputLine, Init(R, 20));
 Campo^.SetValidator(New(PValidadorNombre, Init));
 Insert(Campo);
 R.Assign(16, 2, 19, 3);
 Insert(New(PLabel, Init(R, '1º', Campo)));
 R.Assign(3, 2, 14, 3);
 Insert(New(PLabel, Init(R, 'Apellido~s~:', Campo)));
 R.Assign(40, 2, 56, 3);
 Campo := New(PInputLine, Init(R, 20));
 Campo^.SetValidator(New(PValidadorNombre, Init));
 Insert(Campo);
 R.Assign(37, 2, 40, 3);
 Insert(New(PLabel, Init(R, '2º', Campo)));
 . . .

 { Definición de campos Edad y Teléfono }
 R.Assign(38, 4, 43, 5);
 Campo := New(PInputLine, Init(R, 3));
 Campo^.SetValidator(New(PValidadorEdad, Init));
 Insert(Campo);
 R.Assign(32, 4, 38, 5);
 Insert(New(PLabel, Init(R, 'E~d~ad:', Campo)));

 R.Assign(50, 4, 62, 5);
 Campo := New(PInputLine, Init(R, 10));
 Campo^.SetValidator(New(PValidadorTlf, Init));
 Insert(Campo);
 R.Assign(45, 4, 50, 5);
 Insert(New(PLabel, Init(R, '~T~lf:', Campo)));
 . . .

 { Definición de campo de fecha y tipo de fecha }
 R.Assign(48, 13, 60, 14);
 Campo := New(PInputLine, Init(R, 10));
 Campo^.SetValidator(New(PValidadorFecha, Init));
 Insert(Campo);
 R.Assign(41, 13, 48, 14);
 Insert(New(PLabel, Init(R, 'Fec~h~a:', Campo)));
 . . .
```

## COLECCIONES

*TCollection* es un tipo objeto que almacena una colección de punteros y proporciona un conjunto de métodos para manipularlos.

El tamaño de un array estándar en Turbo Pascal se fija en tiempo de compilación, lo cual está bien si se conoce siempre el tamaño que debe tener el array, pero en el momento en que alguien ejecute el programa puede no ser un buen ajuste. El cambio del tamaño de un array implica la modificación del código y su recompilación. Sin embargo, con una colección, se establece un

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

tamaño inicial, pero ésta puede crecer dinámicamente en tiempo de ejecución para acomodar los datos almacenados en ella. Esto hace que la aplicación sea mucho más flexible en su versión compilada.

Los objetos colección son polimórficos. Eliminan la limitación de los arrays, en los que cada elemento debe ser del mismo tipo, y éste debe determinarse cuando se compila el código. Para ello las colecciones utilizan punteros sin tipo. No sólo es más rápido y eficiente, sino que además una colección puede almacenar objetos de diferentes tipos y tamaños. Esta característica no cumple la fuerte comprobación de tipos tradicional del lenguaje Pascal.

Se puede añadir elementos a una colección que no se derive de *TObject*, pero esto puede producir un problema. Las colecciones esperan recibir punteros sin tipo, pero algunos de los métodos de *TCollection* trabajan específicamente con una colección de instancias derivadas de *TObject*. Esto incluye los métodos de acceso a stream *PutItem* y *GetItem* y también el procedimiento estándar *FreeItem*. Si se desea utilizar elementos que no sean *TObject* habrá que redefinir estos métodos.

Una colección se puede cargar de un stream (como `FicheroFichas`), o construir un objeto colección indicando al constructor el nº inicial de elementos y el incremento de elementos que se establece cada vez que sea preciso aumentar el tamaño de la colección `New(PCollection, Init(10, 10))`.

El método `Done` de los objetos de la colección (en el ejemplo 14.31 `PObjetoFicha`) es llamado para cada elemento cuando se libera una colección. Para liberar una colección se llama al procedimiento `Dispose`: `Dispose(objeto_colección, Done)`.

Las colecciones tienen tres métodos iteradores: *ForEach*, *FirstThat*, y *LastThat*, que permiten tratar o bien todos, o bien el primer o el último elemento de la colección que satisface una determinada condición. Cada uno de ellos lleva un puntero a un procedimiento o función que es su único parámetro. En el caso de *ForEach* es un procedimiento que se aplica a todos los elementos de la colección. En el caso de *FirstThat* y *LastThat* es una función booleana que indicará si un elemento cumple o no una determinada característica. La dirección del primer elemento para el cual devuelva la función *True*, o el último en el caso de *LastThat*, será regresado por el iterador. En el caso de que ningún elemento satisfaga la función el valor devuelto será **nil**.

El método `At` en el método `MostrarFicha` del ejemplo 14.31 permite posicionarnos en el elemento `FichaActiva` de la colección para obtener la información del registro de transferencia y presentarlo en la ventana de fichas de agenda. El método `SalvarDatosFicha` inserta una nueva ficha en la colección `ColecFichas` si nos hemos posicionado después del último elemento de la colección, y en caso contrario modifica el elemento de la posición activa de la colección con los datos extraídos de la ventana `VentanaFicha`.

### Ejemplo 14.31

```

type
 PFicha = ^TFicha;
 TFicha = record
 Apell1, Apell2: string[20];
 Nombre: string[25];
 Edad: shortint;
 Tlf: string[10];
 Domicilio: string[35];
 Portal: integer;
 Piso: shortint;
 Letra: char;
 Ciudad: string[20];
 CP: string[5];
 Relacion, LongNotas: Word;
 Notas: array[0..255] of Char;
 Fecha: string[10];
 TipoFecha: Word;
 end;

var
 FichaInfo: TFicha;
 ColecFichas: PCollection;
 FichaActiva: Integer;
 FichaTemporal: PObjetoFicha;
 . . .

{ Procedimientos para cargar y salvar Fichas de la Agenda }
procedure TGestionFichas.CargaFichas;
begin
 . . .
 ColecFichas := PCollection(FicheroFichas.Get);
 . . .
 ColecFichas := New(PCollection, Init(10, 10));
 . . .
end;

. . .

procedure TGestionFichas.SalvarDatosFicha;
begin
 if VentanaFicha^.Valid(cmClose) then
 begin
 VentanaFicha^.GetData(FichaInfo);
 if FichaActiva = ColecFichas^.Count then
 begin
 FichaTemporal^.RegisTransferencia := FichaInfo;
 ColecFichas^.Insert(FichaTemporal);
 VentanaFicha^.ContadorFichas^.IncContador;
 end
 else
 PObjetoFicha(ColecFichas^.At(FichaActiva))^RegisTransferencia :=
FichaInfo;
 SalvaFichas;
 MostrarFicha(FichaActiva);
 end;
 end;

. . .

procedure TGestionFichas.MostrarFicha(ANumFicha: Integer);
begin
 if ColecFichas^.Count > 0 then
 begin
 FichaActiva := ANumFicha;

```



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
FichaInfo := PObjetoFicha(ColecFichas^.At(FichaActiva))^Re-
gisTransferencia;
with VentanaFicha^ do
begin
 SetData(FichaInfo);
 ContadorFichas^.PonerActivo(FichaActiva + 1);
end;
. . .
end;
```

El tipo especial de colección *TSortedCollection* (descendiente de *Tcollection*) permite ordenar los datos de cualquier manera que se desee. Este tipo de colección comprueba automáticamente la colección cuando se añade un miembro nuevo y rechaza los duplicados.

*TSortedCollection* es un tipo abstracto. Para utilizarlo, se debe primero decidir qué tipo de datos se almacenarán y definir dos métodos que determinen los criterios de ordenación. Para hacer ésto, es necesario derivar un nuevo tipo colección de *TSortedCollection* redefiniendo los métodos *KeyOf* y *Compare*, que permiten definir la clave de ordenación y la forma de comparar dos elementos de la colección.

Además de *TSortedCollection* Turbo Vision tiene otros dos tipos objeto de colecciones: de cadenas *TStringCollection* y recursos *TResourceCollection*.

Cada elemento que se añade a una colección ocupa sólo cuatro bytes ya que se almacena como un puntero. Si no hay memoria disponible cuando se añade un elemento, se llama al método *TCollection.Error* generándose un error en tiempo de ejecución. Se puede redefinir este método para personalizar el mensaje o proporcionar un mecanismo de recuperación.

## STREAMS

Un flujo o *stream* de Turbo Vision es una colección de objetos que se dirige a: generalmente un fichero, memoria EMS, un puerto serie, o cualquier otro dispositivo. Los streams manejan la E/S al nivel del objeto, en vez de al nivel de los datos. Cuando se amplía un objeto Turbo Vision, es necesario proporcionarle mecanismos para la manipulación de los campos de datos adicionales que se definan. Toda la complejidad de la gestión de la representación de los objetos es responsabilidad del programador.

En Pascal se especifica el tipo de datos que se van a leer o escribir en un fichero en tiempo de compilación. Turbo Pascal permite saltarse esta regla mediante los procedimientos *BlockWrite* y *BlockRead* (dejando al programador la responsabilidad de comprobar los tipos) los cuales realizan E/S binaria a mayor velocidad. No obstante, Turbo Pascal presenta otro inconveniente al no permitir crear ficheros con objetos. El almacenar objetos fuera del programa carece de sentido así como

su lectura<sup>66</sup>. Una solución a este problema, algo engorrosa, consistiría en copiar los datos de los objetos fuera y almacenar esa información con la que más tarde se podrán reconstruir los objetos siguiendo el camino inverso.

Los streams proporcionan un solución sencilla para almacenar objetos fuera del programa. La comprobación de tipos se mantiene, pero no se realiza en tiempo de compilación. Se deben definir los objetos que un stream puede manejar, de manera que se puedan emparejar los datos de los objetos con sus tablas de métodos virtuales.

Los tipos stream que dispone Turbo Vision son *TStream* (stream abstracto) y sus tipos objeto derivados: *TDosStream*, que proporciona E/S a disco, *TBufStream*, que ofrece E/S a disco con buffer, y *TEmsStream*, un stream que envía objetos a memoria EMS. También implementa un stream indexado, con un puntero a una posición del mismo. Reasignando ese puntero, se pueden realizar accesos aleatorios al stream.

A continuación vamos a ver las distintas operaciones que se debe realizar con un stream para poder utilizarlo correctamente.

#### • Inicialización

Un stream se inicializa con su constructor *Init*. La sintaxis del constructor depende del tipo de stream. Si se abre un stream DOS (*TDosStream*) se pasa el nombre de un fichero DOS y el modo de acceso (lectura, escritura o lectura/escritura):

```
{ Modos de acceso a TStream }
 stCreate = $3C00; { Crear nuevo fichero }
 stOpenRead = $3D00; { Acceso de sólo lectura }
 stOpenWrite = $3D01; { Acceso de sólo escritura }
 stOpen = $3D02; { Acceso de lectura y escritura }
```

El campo *status* de un stream nos permite saber si la última operación realizada sobre él se ha concluido satisfactoriamente o no.

Para eliminar un stream (equivalente a cerrar un fichero) se llama a su método *Done*.

#### • Introducción y recuperación de objetos

La introducción de objetos en un stream que haya sido inicializado convenientemente se realiza con el método *Put* al que se le pasa el objeto que se quiere añadir en el stream por medio de un puntero. El objeto tiene que estar registrado para que el stream pueda conocer su ID y cuantos datos se escribirán a continuación. Cuando el objeto es un grupo con subvistas, estas son enviadas al stream automáticamente.

---

<sup>66</sup> Los objetos tienen una tabla de métodos virtuales (VMT) que contiene las direcciones de estos métodos (calculadas en tiempo de ejecución).

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

La recuperación de objetos se realiza con la función `Get`, que devuelve un puntero al objeto que se haya almacenado al stream. Si el puntero devuelto no es del mismo tipo que el del puntero al que se asigna se obtendrá información alterada. Si el objeto es un grupo se recuperará con todas sus subvistas.

En el ejemplo 14.32 se presentan dos funciones para salvar las fichas de la colección `ColecFichas` del programa AGENDA con `SalvaFichas` y para recuperar un desktop con `CargarDesktop`. En ambos casos se utilizan cadenas de marca al principio del fichero para verificar que realmente se va a recuperar el fichero deseado. Para ello se utiliza el procedimiento de stream `Write` para escribir directamente la cadena y `Read` para recuperarla.

### Ejemplo 14.32

```
{ Información para dejar marca en el fichero de Desktop y Fichas }
LongMarcaD = 23;
LongMarcaF = 17;
MarcaDesktop: string[LongMarcaD] = 'Fichero Desktop Agenda'#26;
MarcaFichas: string[LongMarcaF] = 'Fichas de Agenda'#26;
. . .

procedure TGestionFichas.SalvaFichas;
var
 FicheroFichas: TBufStream;
 F: File;
begin
 FicheroFichas.Init('AGENDA.DAT', stCreate, 1024);
 if (FicheroFichas.Status = stOk) then
 begin
 FicheroFichas.Write(MarcaFichas[1], LongMarcaF);
 FicheroFichas.Put(ColecFichas);
 if FicheroFichas.Status <> stOk then
 begin
 MessageBox('No se puede crear el fichero AGENDA.DAT.', nil, mfOkButton
+ mfError);
 {$I-}
 FicheroFichas.Done;
 Assign(F, 'AGENDA.DAT');
 Erase(F);
 Exit;
 end;
 end;
 FicheroFichas.Done;
 EnableCommands([cmVentanaFicha]);
end;
procedure TGestionFichas.CargarDesktop;
var
 FicheroDesktop: TBufStream;
 DesktopTemp: PDesktop;
 Marca: string[LongMarcaD];
 R: TRect;
begin
 FicheroDesktop.Init('AGENDA.DSK', stOpenRead, 1024);
 if LowMemory then OutOfMemory
 else if FicheroDesktop.Status <> stOk then
 MessageBox('No se puede abrir el fichero de desktop', nil, mfOkButton +
mfError)
 else
 begin
 Marca[0] := Char(LongMarcaD);
 FicheroDesktop.Read(Marca[1], LongMarcaD);
 if Marca = MarcaDesktop then
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
begin
 DesktopTemp := PDesktop(FicheroDesktop.Get);
 if FicheroDesktop.Status <> stOk then
 MessageBox('Error al leer el fichero de desktop ', nil,
 mfOkButton + mfError);
 end
 else
 MessageBox('Error: Fichero de desktop no válido.', nil,
 mfOkButton + mfError);
 end;
FicheroDesktop.Done;
if ValidView(DesktopTemp) <> nil then
begin
 Desktop^.Delete(VentanaClipboard);
 Delete(Desktop);
 Dispose(Desktop, Done);
 Desktop := DesktopTemp;
 Insert(Desktop);
 GetExtent(R);
 R.Grow(0, -1);
 Desktop^.Locate(R);
 InsertWindow(VentanaClipboard);
 VentanaFicha := Message(Desktop, evBroadcast, cmBuscarFichaVentana, nil);
 if VentanaFicha <> nil then MostrarFicha(FichaActiva);
end;
end;
```

### • Manejo de errores

En caso de error el procedimiento *Error* determina qué ocurre. Por defecto, *TStream.Error* simplemente da valor a los campos *Status* y *ErrorInfo* del stream y llama al procedimiento indicado por la variable de la aplicación *StreamError*, que por defecto vale *nil*. Para obtener un comportamiento distinto se debe redefinir *Error* o asignar un procedimiento de gestión de mensajes de error a *StreamError*.

```
{ Códigos de error de TStream }
stOk = 0; { Sin error }
stError = -1; { Error de acceso }
stInitError = -2; { No se puede inicializar stream }
stReadError = -3; { Lectura después de fin de stream }
stWriteError = -4; { No se puede expandir el stream }
stGetError = -5; { Se quiere obtener un tipo de objeto no registrado}
stPutError = -6; { Se quiere escribir un tipo de objeto no registrado}
```

En el ejemplo 14.33 se implementa un procedimiento de gestión de errores que se asigna a *StreamError*.

### Ejemplo 14.33

```
{ Procedimiento de gestión de errores de Stream }
procedure AgendaStreamError(var S: TStream); far;
var
 MensajeError: String;
begin
 case S.Status of
 stError: MensajeError := 'Error en el stream ' ;
 stInitError: MensajeError := ' No se puede inicializar el stream';
 stReadError: MensajeError := ' Lectura después de fin de stream';
 stWriteError: MensajeError := ' No se puede añadir en el stream';
 stGetError: MensajeError := 'Tipo sin registrar para leer del stream';
```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
 stPutError: MensajeError := 'Tipo sin registrar para escribir en stream';
end;
DoneVideo;
PrintStr('Error: ' + MensajeError);
Halt(Abs(S.Status));
end;
. . .
constructor TGestionFichas.Init;
begin
. . .
StreamError := @AgendaStreamError;
. . .
```

El ejemplo 14.34 contiene la unit *ErrorStream* que implementa un stream que cuando se produce un error en la inicialización no llama al método *Error* del ascendiente, de forma que no se realiza la detección de la aplicación permitiendo verificar que si el fichero no existe se pueda crear. En el ejemplo 14.32 la función *CargarDesktop* si el fichero de desktop no existe, se detiene el programa al utilizar el tipo *TBufStream*. En cambio el método *TGestionFichas.CargaFichas* que carga el stream de fichas, al utilizar el tipo *ErrorStream* ante un fallo de la inicialización de la colección *ColecFichas* a partir del stream de fichas puede recuperarse creando una nueva colección.

### Ejemplo 14.34

```
{ ***** }
{ Unit de adaptación del manejador de errores de TBufStream. }
{ ***** }
```

```
Unit ErrorStream;

interface
uses Objects;

type
PErrorStream = ^TErrorStream;
TErrorStream = Object(TBufStream)
 procedure Error(Code, Info: Integer); virtual;
end;

implementation

{ TErrorStream }
{ Se evita que se llama a la función de errores cuando se produce error
en la apertura de un stream de forma que se puede saber si existe sin
que se pase el control a la rutina de error que se indica en la vble.
global StreamError que en la implementación del programa "Agenda"
realiza un "halt" y termina el programa }
procedure TErrorStream.Error(Code, Info: Integer);
begin
 if Code = stInitError then
 begin
 Status:=Code;
 ErrorInfo:=Info;
 end
 else
 TBufStream.Error(Code, Info);
end;
end.
```

### • Registro de objetos para utilizarlos con streams

Para poder utilizar un tipo objeto con streams se debe registrar y además definir dos métodos para cargar y almacenar en el stream. Todos los objetos de Turbo Vision está preparados ya para poder utilizarlos con streams con tan sólo llamar al método de registro de la unit en la que están implementados (*Register...*) en la inicialización de la aplicación.

#### Ejemplo 14.35

```

const
 RObjetoFicha: TStreamRec = (
 ObjType: 15000;
 VmtLink: ofs(KindOf(TObjetoFicha)^);
 Load: @TObjetoFicha.Load;
 Store: @TObjetoFicha.Store
);
 . . .

constructor TObjetoFicha.Load(var S: TStream);
begin
 inherited Init;
 S.Read(RegisTransferencia, SizeOf(RegisTransferencia));
end;

procedure TObjetoFicha.Store(var S: TStream);
begin
 S.Write(RegisTransferencia, SizeOf(RegisTransferencia));
end;
 . . .

constructor TGestionFichas.Init;
 . . .
begin
 . . .
 RegisterMenus;
 RegisterObjects;
 RegisterViews;
 RegisterApp;
 RegisterEditors;
 RegisterDialogs;
 RegisterValidar;
 RegisterCalc;
 RegisterCalendario;
 RegisterContador;
 RegisterFondoMuro;
 RegisterAyuda;
 RegisterLineaEstado;
 RegisterType(RObjetoFicha);
 RegisterType(RVentanaFicha);
 RegisterType(RDesktopConFondo);
 . . .

```

La lectura y escritura real de objetos en un stream es gestionada por los métodos *Load* y *Store*. Aunque cada objeto debe tener estos métodos para que sea utilizable con streams, nunca deben ser llamados directamente. (Estas llamadas las realizan *Get* y *Put*.) Sólo es necesario garantizar que el objeto sabe enviarse a sí mismo al stream cuando se le indique. Si se modifican los campos de un objeto hay que asegurarse de actualizar los métodos *Load* y *Store*. En el ejemplo

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

14.35 el tipo objeto `TObjetoFicha` llama a los métodos del ascendiente y sólo debe añadir la lectura/escritura del campo de `RegisTransferencia` que se le ha añadido respecto a `TObject`. Recuérdese que los objetos manipulados por streams deben ser descendientes de este tipo objeto.

Para realizarla declaración o registro de un tipo objeto se debe definir primero un *record* de declaración de stream y luego pasárselo al procedimiento global `RegisterType`.

Los registros de declaración de stream son registros Pascal de tipo `TStreamRec`:

```
TStreamRec = record
 ObjType: Word;
 VmtLink: Word;
 Load: Pointer;
 Store: Pointer;
 Next: Word;
end;
```

Por convenio, a todos los registros de declaración de stream Turbo Vision se les da el mismo nombre que el correspondiente tipo de objeto, donde la inicial **T** se reemplaza por una **R**. Los tipos abstractos como `TObject` y `TView` no tienen registros de declaración ya que nunca habrá instancias de ellos que se almacenen en los streams.

El campo *ObjType* es parte del registro con la que hay que tener más precaución. Cada tipo nuevo que se defina necesita su propio y único número identificador de tipo. Turbo Vision reserva los números de declaración 0 a 99 para los objetos estándar, mientras que los números de declaración de usuario pueden ser cualquiera desde 100 hasta 65.535.

El programador debe crear y mantener una librería de números ID para todos los objetos nuevos que sean usados en la E/S con streams, así como garantizar que los IDs estarán disponibles para los usuarios en las units. Los números que se asignen pueden ser completamente arbitrarios, mientras sean únicos.

El campo *VmtLink* es un enlace a la tabla de métodos virtuales (VMT) del objeto, y se le asigna al desplazamiento del tipo del objeto:

```
RObjetoFicha.VmtLink := ofs(TypeOf(TObjetoFicha)^);
```

Los campos *Load* y *Store* contienen las direcciones de estos métodos.

```
RObjetoFicha.Load := @TObjetoFicha.Load;
RObjetoFicha.Store:= @TObjetoFicha.Store;
```

Por último, el campo *Next* es asignado por `RegisterType`, y no requiere intervención por parte del programador.

Una vez que se ha construido el registro de declaración de stream, se llama a `RegisterType` con el registro como parámetro:

```
RegisterType(RObjetoFicha);
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

El constructor de la aplicación `TGestionFichas.Init` en el ejemplo 14.35 llama a los procedimientos *Register...* que declaran los objetos de las units de Turbo Vision que se utilizan, así como a los procedimientos de declaración de otras units de usuario y registra además los propios tipos objeto del programa con *RegisterType*.

### • Acceso aleatorio

Los streams vistos funcionan como dispositivos secuenciales: se ponen (*Put*) los objetos al final del stream, y se recuperan (*Get*) en el mismo orden. Turbo Vision permite tratar un stream como un dispositivo virtual de acceso aleatorio. Como complemento a *Get* y *Put*, que corresponden a *Read* y *Write* en ficheros, los streams proporcionan características análogas al *Seek*, *FilePos*, *FileSize* y *Truncate* para uso con ficheros.

- Seek*** Este procedimiento mueve el puntero actual del mismo a una posición especificada (en bytes desde el comienzo del stream), tal como el procedimiento estándar *Seek* de Turbo Pascal.
- GetPos*** Esta función es la inversa del procedimiento *Seek*. Devuelve un *Longint* con la posición actual del stream.
- GetSize*** Devuelve el tamaño del stream en bytes.
- Truncate*** Borra todos los datos a partir de la posición actual del stream, haciendo que la posición actual sea el final del mismo.

El acceso aleatorio a streams requiere el mantenimiento de un índice fuera del stream, que registre la posición de comienzo de cada objeto en el mismo. Se puede utilizar una colección para realizar esta tarea. Este es lo que hace Turbo Vision con los ficheros de recursos.

## RECURSOS

Un fichero de recursos funciona como un stream de acceso aleatorio, con objetos a los que se accede mediante *claves*, cadenas únicas que identifican los recursos. Por lo tanto, los objetos guardados en un recurso pueden después recuperarse por su nombre. De esta forma una aplicación puede recuperar desde un recurso los objetos que use en vez de inicializarlos. El programa `RECURSOS.PAS` genera algunos de los objetos del programa `AGENDA`. Este último no tiene que crear esos objetos ni inicializarlos, basta con que los tome del recurso.

Los recursos permiten personalizar una aplicación sin cambiar el código, pudiéndose mantener versiones específicas de una aplicación para distintos idiomas o versiones con distintas capacidades. También ahorra código en la aplicación al ponerse los constructores de objetos en el programa que genera los recursos.



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

Un tipo objeto recurso *TResourceFile* contiene una colección ordenada de cadenas y un stream. Las cadenas de la colección son las claves de los objetos que hay en el stream. *TResourceFile* tiene un método *Init* que inicializa un stream al que asocia una colección para efectuar el acceso al mismo, y un método *Get* que toma una cadena y devuelve un objeto del stream.

El ejemplo 14.36 es parte del programa RECURSOS.PAS. En este ejemplo se utiliza el recurso *ResFile* para almacenar los objetos (barra de menús, línea de estado, ...) en el stream por medio del método *Put* (e.j. *ResFile.Put(LineaDeEstado, 'ESTADO');*) asociando a cada objeto una cadena.

### Ejemplo 14.36 recursos.pas

```
var
 R: TRect;
 ResFile: TResourceFile;
 MenuPrincipal: PMenuBar;
 LineaDeEstado: PLineaEstado;
 VentanaFicha, CajaAbout: PDialog;
 . . .
begin
 RegisterViews;
 RegisterDialogs;
 RegisterMenus;
 RegisterValidar;
 RegisterEditors;
 RegisterLineaEstado;

 { Crear Stream para almacenar recursos }
 ResFile.Init(New(PBufStream, Init('AGENDA.TVR', stCreate, 2048)));
 . . .

 { Definición del Menu Principal }
 R.Assign(0, 0, 80, 1);
 New(MenuPrincipal,
 . . .
 ResFile.Put(MenuPrincipal, 'MENUPRINCIPAL');

 { Definición de la línea de estado para cada contexto }
 R.Assign(0, 24, 80, 25);
 New(LineaDeEstado,
 . . .
 ResFile.Put(LineaDeEstado, 'ESTADO');

 { Dialogo "Acerca de ..." }
 R.Assign(0, 0, 50, 14);
 CajaAbout := New(PDialog,
 . . .
 ResFile.Put(CajaAbout, 'ACERCA');

 { Ventana para las fichas de la agenda }
 R.Assign(0, 0, 66, 21);
 VentanaFicha := New(PDialog,
 . . .
 ResFile.Put(VentanaFicha, 'FICHAS');

 ResFile.Done;
end.
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

Para recuperar los objetos de un recurso se utiliza el método *Get*. En el ejemplo 14.37 (programa AGENDA.PAS) se recuperan los objetos almacenados por el programa RECURSOS.PAS. (e.j. `StatusLine := PLineaEstado(FicheroRecursos.Get('ESTADO'));`)

### Ejemplo 14.37 recursos.pas

```
var
 FicheroRecursos: TResourceFile;
 FichaInfo: TFicha;
 ColecFichas: PCollection;
 FichaActiva: Integer;
 FichaTemporal: PObjetoFicha;
 . . .

constructor TGestionFichas.Init;
begin
 . . .
 FicheroRecursos.Init(New(PBufStream, Init('AGENDA.TVR', stOpenRead, 1024)));
 . . .
end;

. . .

procedure TGestionFichas.InitMenuBar;
begin
 MenuBar := PMenuBar(FicheroRecursos.Get('MENUPRINCIPAL'));
end;

procedure TGestionFichas.CajaAbout;
begin
 ExecuteDialog(PDialog(FicheroRecursos.Get('ACERCA')), nil);
end;

procedure TGestionFichas.InitStatusLine;
var
 R: TRect;
begin
 StatusLine := PLineaEstado(FicheroRecursos.Get('ESTADO'));
 GetExtent(R);
 StatusLine^.MoveTo(0, R.B.Y - 1);
end;
. . .

destructor TGestionFichas.Done;
begin
 FicheroRecursos.Done;
 inherited Done;
end;
```

Si cuando se lee un objeto de un recurso el índice pasado a *Get* no es válido este método devuelve **nil**. Se puede leer repetidas veces un objeto de un recurso, como por ejemplo un cuadro de diálogo que sea solicitado numerosas veces a lo largo de la ejecución de un programa. Para acelerar estos accesos se puede pasar el stream a un stream EMS.

## AYUDA SENSIBLE AL CONTEXTO

Para facilitar la utilización de una aplicación y acelerar el correcto uso de la misma es conveniente incorporar la posibilidad de acceder a un sistema de *ayuda sensible al contexto*,

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

disponible en cualquier ámbito del programa. Un sistema de ayuda sensible al contexto debe permitir presentar información sobre la parte de una aplicación que se esté usando en un determinado momento. Aparte de disponer de esta prestación también se obtendría un valor añadido, si el programa dispusiera de una vista activa durante toda la ejecución (*e.j.*, línea de estado), en la cual se hiciesen indicaciones sobre las operaciones que se están realizando o se presentasen mensajes al usuario. En la figura 14.44 se pueden observar dos pantallas de la aplicación AGENDA; en la de la derecha en la línea de estado se presenta una indicación sobre el elemento de menú que se ha seleccionado y en la derecha un pantalla de ayuda solicitada sobre otro elemento de menú.

Turbo Vision no da un soporte directo para el desarrollo de ayudas (por mediante de ventanas) sensibles al contexto, pero sí para la presentación de indicaciones que sirvan de guía al usuario. Esta prestación se consigue mediante la creación de un tipo objeto descendiente de *TStatusLine* en el cual se redefine un método heredado denominado *hint*. Este método, por defecto, devuelve un string vacío. En los objetos descendientes de la línea de estado *TStatusLine* se redefine para devolver una cadena con una indicación sensible al contexto por medio del parámetro de tipo *Word* que se le pasa. Si la cadena que retorna *hint* no está vacía se escribe en la línea de estado de la aplicación después de una barra de división que la separa de las combinaciones de teclas y su cometido, presentadas también en la línea de estado.

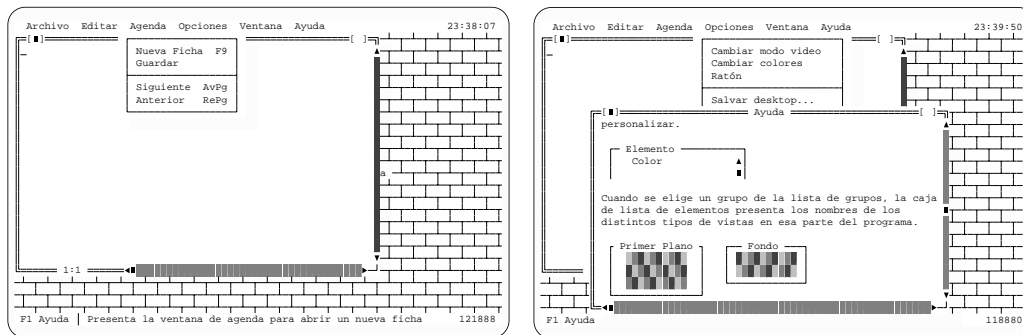


Fig. 14.44 Indicación en línea de estado y ventana de ayuda

En el ejemplo 14.38 contiene el código de la unit LINEAEST.TPU, donde se define el tipo objeto *TLineaEstado* que es un descendiente de *TStatusLine*, al que se redefine el método *hint* para que presente indicaciones en la aplicación AGENDA según el contexto del programa que se pasa en el parámetro *ContextoAyuda*. Este parámetro se utiliza para indexar el array de cadenas de caracteres *TextoAviso*, que contiene todas las indicaciones del programa.

### Ejemplo 14.38

```

}
 Esta Unit define un nuevo tipo de línea de estado que
} permite presentar avisos o consejos (hints) en la propia
}

```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

{ línea de estado para el programa de agenda.
{*****}
}

unit LineaEstado;

interface
uses Objects, Drivers, Menus, Views, App, AyudaAgenda;

type
 PLineaEstado = ^TLineaEstado;
 TLineaEstado = object(TStatusLine)
 function Hint(ContextoAyuda: Word): String; virtual;
 end;

const
 RLineaEstado: TStreamRec = (
 ObjType: 11011;
 VmtLink: Ofs(KindOf(TLineaEstado)^);
 Load: @TLineaEstado.Load;
 Store: @TLineaEstado.Store
);
procedure RegisterLineaEstado;

implementation
{ Los avisos o indicaciones se definen para los menús, sus opciones
 y las ventanas , cuyos contextos de ayuda son los siguiente :
 Archivo = 3 Video = 23
 Nuevo = 4 Colores = 24
 Abrir = 5 Raton = 25
 Guardar = 6 SalvarDesktop = 26
 GuardarComo = 7 RecuperarDesktop = 27
 ShellDos = 8 Ventana = 28
 Salir = 9 VAgenda = 29
 Editar = 10 VCalculadora= 30
 Deshacer = 11 VCalendario=31
 Cortar = 12 Mosaico = 32
 Copiar = 13 Cascada = 33
 Pegar = 14 Redimensionar = 34
 Limpiar = 15 Ampliar = 35
 MostrarClip = 16 Siguiete = 36
 MAgenda = 17 Previa = 37
 NuevaFicha = 18 Cerrar = 38
 AGuardar = 19 Ayuda = 39
 ASiguiete = 20 Indice = 40
 APrevia = 21 Acerca=41
 Opciones = 22
}
const
 TextoAviso: array[hcArchivo..hcAcerca] of string[57] =
 ('Comandos para gestión de ficheros',
 'Crea un nuevo fichero en una ventana de editor',
 'Carga un fichero dado en una ventana de editor',
 'Salva el fichero de la ventana de editor activa',
 'Salva el fichero activo con diferente nombre',
 'Salida temporal al DOS',
 'Salir de AGENDA',
 'Comandos de edición',
 'Deshace la última operación de edición',
 'Elimina el texto seleccionado y lo lleva al portapapeles',
 'Copia el texto seleccionado al portapapeles',
 'Inserta el texto seleccionado desde el portapapeles',
 'Borra el texto seleccionado',
 'Abre la ventana del portapapeles',
 'Comandos de gestión de agenda',
 'Presenta la ventana de agenda para abrir un nueva ficha',
 'Almacena la agenda',

```

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
'Presenta la siguiente ficha',
'Presenta la ficha anterior',
'Configuración del programa',
'Cambia el modo de video entre 25 y 43/50 líneas',
'Personaliza los colores de ventanas, menús, ...',
'Especifica los parámetros de funcionamiento del ratón',
'Salva el estado de las ventanas abiertas',
'Recupera el estado de las ventanas abiertas grabado antes',
'Abrir y gestionar ventanas',
'Abrir la ventana agenda',
'Abrir la ventana calculadora',
'Abrir la ventana calendario',
'Distribuye las ventanas sin que se solapan',
'Distribuye las ventanas en cascada',
'Cambia el tamaño y posición de la ventana activa',
'Amplia o restaura el tamaño de la ventana activa',
'Activa la siguiente ventana',
'Activa la ventana anterior',
'Cierra la ventana activa',
'Obtiene ayuda en línea',
'Presenta un índice de la ayuda en línea',
'Muestra copyright y autores del programa');

function TLineaEstado.Hint(ContextoAyuda: Word): String;
begin
 case ContextoAyuda of
 hcArchivo .. hcAcerca: Hint := TextoAviso[ContextoAyuda];
 else
 Hint := '';
 end;
end;

{ Registrar objetos }
procedure RegisterLineaEstado;
begin
 RegisterType(RLineaEstado);
end;

end.
```

El valor pasado en ContextoAyuda al método hint se obtiene de la vista activa en ese momento (menú, ventana o diálogo). Si es un diálogo o una ventana ese valor está almacenado en el campo HelpCtx que habrá sido asignado en la inicialización del objeto.

### Ejemplo 14.39.

```
procedure Raton;
var
 D: PDialog;
begin
 D := New(PDialogoRaton, Init);
 D^.HelpCtx := hcDialogoRaton;
 ExecuteDialog(D, @MouseReverse);
end;

procedure TGestionFichas.NuevaVentana;
var
 R: TRect;
 Ventana: PEditWindow;
begin
 R.Assign(0, 0, 60, 20);
 Ventana := New(PEditWindow, Init(R, '', wnNoNumber));
```

```
Ventana^.HelpCtx := hcEditor;
 InsertWindow(Ventana);
end;
```

Si es un elemento de menú, se tomará el valor que se pasó al método `NewItem`, con el que se definió dicho elemento, en el último de los parámetros (*ejemplo 14.40* `hcGuardar`). Si no se quiere definir un contexto de ayuda concreto para un elemento de menú, se le asigna el valor 0 o la constante `hcNoContext` definida en la Unit `VIEWS` (`hcNoContext = 0;`). Este es el valor que se asigna por defecto al campo `HelpCtx` de los objetos `TView`. Si se define una ayuda para `hcNoContext`, esta será la que aparecerá por defecto en aquellos elementos que no tengan un contexto de ayuda propio. Los valores de contexto de ayuda de los objetos vistas se suelen definir como constantes que se les asigna valores de tipo `Word` y cuyos identificadores, por convenio, comienzan por "*hc...*".

#### Ejemplo 14.40.

```
NewItem('~G~uardar', 'F2', kbF2, cmSave, hcGuardar,
```

Como hemos dicho Turbo Vision no da soporte directo para la creación de un sistema de pantallas de ayuda sensibles al contexto dentro de un programa (utilizando la jerarquía de tipos de objeto de Turbo Vision). No obstante con la librería de Turbo Vision se distribuye la Unit `HELPPFILE.TPU` y su código fuente `HELPPFILE.PAS`<sup>67</sup>. Esta unit aporta los tipos de objeto y métodos necesarios para gestionar un fichero de contextos de ayuda y poder presentar la información de estos contextos en una ventana dentro de una aplicación desarrollada con Turbo Vision. El *fichero de ayuda* debe tener una estructura concreta, que se puede obtener por medio del programa `TVHC 1.0` (compilador de ficheros de ayuda de Turbo Vision). Este programa toma un fichero de descripción de contextos de ayuda y genera un fichero de ayuda (`.HLP`) y un fichero de contextos de ayuda (`.PAS`).

En el directorio `TVHC` de los ejemplos del capítulo 14, se encuentra el fichero `TVHC.PAS` que es una copia del compilador de ficheros de ayuda de Borland modificado para que todos los mensajes al usuario aparezcan en castellano. De igual forma el fichero `AYUDA.PAS` del directorio `TPU` de este capítulo es una adaptación<sup>68</sup> del fichero `HELPPFILE.PAS` y con el se genera la Unit `AYUDA.TPU` que se utiliza en el programa agenda.

Veamos el formato que debe tener un fichero de descripción de pantallas de ayuda para poder generar el fichero de ayuda del programa.

A cada contexto o ayuda para un punto del programa se le da un nombre simbólico (*e.j.* `Salir`) que será añadido en el fichero de contextos de ayuda (`.PAS`) (*e.j.* `hcSalir`). El texto que sigue a la línea de tópico se pone en el fichero de ayuda (`.HLP`). Como la ventana de ayuda puede ser redimensionada, se necesitará reagrupar o redistribuir el texto de las líneas (en nuevas líneas) para

---

<sup>67</sup> El fichero `HELPPFILE.PAS` se encuentra en el directorio "*examples\tvdemo*" del compilador.

<sup>68</sup> Los mensajes que pueden aparecer en pantalla se ha traducido a castellano así como algunos identificadores.

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

adaptarse al nuevo tamaño de la ventana. Si una línea de texto comienza sin ningún carácter de espacio en blanco, ésta podrá ser reagrupada o reajustada si es necesario. Si la línea comienza por un espacio en blanco no será reagrupada con otras líneas. Por ejemplo, el texto siguiente define un tópico de ayuda para un elemento de menú `Archivo | Salir` que finalice la ejecución del programa:

```
.topic Salir
 Archivo | Salir (Alt-S)
 =====
El comando Salir finaliza
```

La línea `Archivo | Salir (Alt-S)` no se reagrupará con la línea `=====` ya que ambas comienzan con un carácter espacio, en cambio la línea `El comando Salir finaliza ....` si será agrupada con la(s) siguientes si es necesario.

La sintaxis para un línea de tópico es:

```
.topic símbolo [=número] [, símbolo [=número] [...]]
```

que como se puede observar, un tópico puede tener múltiples símbolos asociados, por lo que puede ser utilizado en varios contextos. El *número* es opcional y será el valor que se le dé al contexto *hc...*, donde los puntos son sustituidos por el símbolo, en el fichero de contexto (.PAS). Una vez que se haya asignado un número, al resto de los símbolos de tópicos se les asociará un número secuencial a partir del dado. Así

```
.topic AbrirArchivo=5, ArchivoNuevo, Abrir
```

genera las siguientes definiciones de números de contexto de ayuda:

```
AbrirArchivo = 5;
ArchivoNuevo = 6;
Abrir = 7;
```

Para incluir *referencias cruzadas*<sup>69</sup> en el texto de un tópico de ayuda se utiliza la siguiente sintaxis:

```
{texto [:alias]}
```

donde el *texto* que va entre llaves es el que aparecerá en la ventana de ayuda resaltado sobre el fondo, pudiendo ser seleccionado por el usuario para acceder al tópico cuyo nombre es *texto*. Cuando el texto a presentar en la referencia cruzada no sea el mismo que el nombre del símbolo del tópico al que se desea referenciar, se puede utilizar la sintaxis alternativa con *alias*. En este caso el símbolo del tópico va después del *texto* y ':'. En el ejemplo 14.41, que presenta fragmentos del fichero AYUDAAGE.TXT que define los tópicos de ayuda de la aplicación AGENDA, se pueden observar la utilización de referencias cruzadas.

---

<sup>69</sup> Permiten al usuario acceder rápidamente a tópicos relacionados

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

El usuario puede utilizar la tecla **TAB** o el ratón para seleccionar más información a través de las referencias cruzadas. El compilador de ayuda maneja referencias avanzadas de forma que un tópico no necesita estar definido antes de ser referenciado. Si un tópico se referencia, pero no se define, el compilador dará un error leve (*warning*) creando no obstante un fichero de ayuda utilizable. Si se utiliza una referencia no definida en el fichero de ayuda se da un mensaje ("*Ayuda no disponible ...*") en la ventana de ayuda.

### Ejemplo 14.41

```
.topic NoContext=0
AGENDA
=====
```

Bienvenido a AGENDA. Este es un ejemplo de aplicación construida con la utilización del marco de aplicación Turbo Vision. Todos los menús del programa son desplegados por medio de la pulsación de Alt-Z, donde Z es el carácter resaltado del nombre de menú. Así, por ejemplo, el menú "Archivo" es desplegado con Alt-A.

Pulsar ESC para que desaparezca esta ventana de ayuda.

```
{IndiceAyuda}
```

```
.topic Archivo = 3
Archivo (Alt-A)
=====
```

El menú de Archivo tienen opciones para abrir y salvar ficheros, salida temporal al shell del sistema operativo DOS y finalizar la ejecución del programa.

```
{Abrir} {Salir} {IndiceAyuda}
```

```
.topic Nuevo = 4
Archivo|Nuevo (F4)
=====
```

El comando Archivo Nuevo crea una nueva ventana de editor. El nombre que se da a la ventana es "Untitled" (Sin\_Título, en inglés), y ésta estará vacía.

```
{IndiceAyuda}
```

```
.topic IndiceAyuda = 42
Indice (Shift+F1)
=====
```

```
{Abrir Archivo:Abrir} {Menu Agenda:MAgenda}
{Acerca del programa:Acerca} {Menu Archivo:Archivo}
{Agenda} {Menu Ayuda:Ayuda}
{Ampliar ventana:Ampliar} {Menu Editar:Editar}
{Apertura de ficheros:FicherosAF} {Modo Video:Video}
{Apertura Fichero:DialogoAbrirFichero} {Mostrar Portapapeles:MostrarClip}
{Archivo Nuevo:Nuevo} {Nombre de un fichero:NombreAF}
{Borrar texto seleccionado:Limpiar} {Nueva Ficha:NuevaFicha}
{Boton Cancel:BotonCancel} {Opciones}
{Boton Open:BotonOpenAF} {Pegar texto:Pegar}
{Calculadora} {Raton}
{Calendario} {Recuperar Desktop:RecuperarDeskTop}
{Cerrar Ventana:Cerrar} {Salir}
{Colores} {Salir al Dos:ShellDos}
{Copiar texto:Copiar} {Salvar Desktop:SalvarDeskTop}
{Cortar texto:Cortar} {Siguiete ficha:ASiguiete}
{Deshacer} {Siguiete Ventana:Siguiete}
```



## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

```
{Dialogo Colores:DialogoColores} {Tamño/Mover ventana:Redimensio-
nar}
{Dialogo Raton:DialogoRaton} {Ventana}
{Editor de ficheros:Editor} {Ventana Agenda:VAgenda}
{Ficha anterior:APrevia} {Ventana Calculadora:VCalculadora}
{Guardar Archivo:Guardar} {Ventana Calendario:VCalendario}
{Guardar Archivo Como:GuardarComo} {Ventana Previa:Previa}
{Guardar ficha:AGuardar} {Ventanas en Cascada:Cascada}
{Indice} {Ventanas en Mosaico:Mosaico}
```

A partir del fichero AYUDAAGE.TXT se genera el fichero de ayuda del programa AYUDAAGE.HLP y el fichero de contextos de ayuda AYUDAAGE.PAS. Este último es el que se muestra en el ejemplo 14.42. Para ello bastará con ejecutar el programa TVHC: TVHC AYUDAAGE.

### Ejemplo 14.42

```
unit AyudaAgenda;

interface
const
 hcAbrir = 5;
 hcAcerca = 41;
 hcAgenda = 601;
 hcAGuardar = 19;
 hcAmpliar = 35;
 hcAPrevia = 21;
 hcArchivo = 3;
 hcASiguiente = 20;
 hcAyuda = 39;
 hcBotonCancel = 105;
 hcBotonOpenAF = 104;
 hcCalculadora = 401;
 hcCalendario = 501;
 hcCascada = 33;
 hcCerrar = 38;
 hcColores = 24;
 hcCopiar = 13;
 hcCortar = 12;
 hcDeshacer = 11;
 hcDialogoAbrirFichero = 101;
 hcDialogoColores = 301;
 hcDialogoRaton = 201;
 hcEditar = 10;
 hcEditor = 65400;
 hcFicherosAF = 103;
 hcGuardar = 6;
 hcGuardarComo = 7;
 hcIndice = 40;
 hcIndiceAyuda = 42;
 hcLimpiar = 15;
 hcMAgenda = 17;
 hcMosaico = 32;
 hcMostrarClip = 16;
 hcNoContext = 0;
 hcNombreAF = 102;
 hcNuevaFicha = 18;
 hcNuevo = 4;
 hcOpciones = 22;
 hcPegar = 14;
 hcPrevia = 37;
 hcRaton = 25;
 hcRecuperarDeskTop = 27;
 hcRedimensionar = 34;
 hcSalir = 9;
 hcSalvarDeskTop = 26;
```

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```

hcShellDos = 8;
hcSiguiete = 36;
hcVAgenda = 29;
hcVCalculadora = 30;
hcVCalendario = 31;
hcVentana = 28;
hcVideo = 23;
implementation
end.

```

Una vez que se tiene el fichero de ayuda y cada vista del programa (ventanas, elementos de menú, ...) tiene definido su contexto de ayuda falta incorporar algún método que se encargue de recibir las peticiones de ayuda del usuario y facilitarle la información que demande. Para esta tarea haremos uso del tipo objeto *PHelpWindow* definido en la Unit *AYUDA.TPU*<sup>70</sup>.

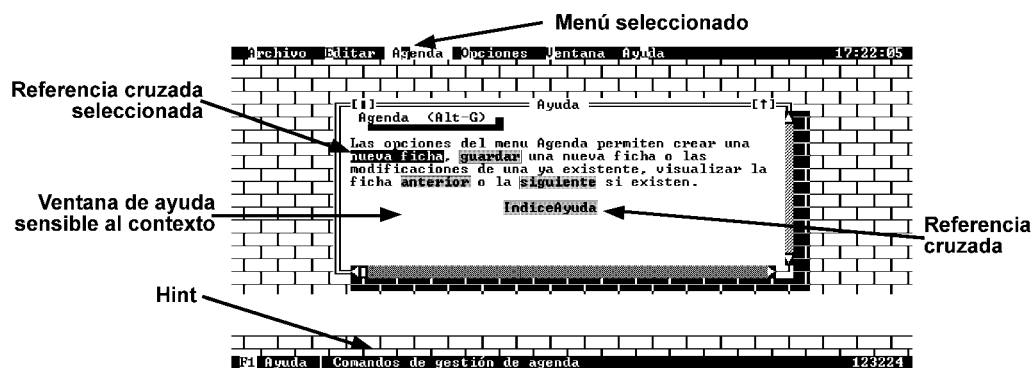


Fig. 14.45 Ventana de ayuda sensible al contexto.

En el método *GetEvent* del tipo objeto *TGestionFichas* (ejemplo 14.43), aparte de generar los eventos a partir de las entradas que recibe el programa, se gestionan las solicitudes de ayuda. Para ello se comprueba si no se está utilizando ya una ventana de ayuda, en cuyo caso no se realiza ninguna acción. Si no se ha abierto ninguna ventana de ayuda (controlado por *AyudaEnUso*) entonces se abre un stream de tipo *THelpFile* asociado con el fichero que devuelve la función *FicheroAyuda* (ejemplo 14.44). Esta función toma el nombre del programa, bien del argumento de índice 0 pasado desde el sistema operativo a través de *ParamStr* si trabajamos con una versión del DOS superior o igual a la 3.00, o bien buscando el programa *AGENDA.EXE* en los directorios indicados en la vble. de entorno *PATH* si la versión es anterior. Del nombre del programa *NombreProgr* se toma el directorio y se añade el nombre del fichero de ayuda *AYUDAAGE.HLP*, con lo que se obtiene el nombre completo (directorio, nombre y extensión) del fichero de ayuda. Si no es posible inicializar el stream con este fichero se presenta una caja de diálogo con un mensaje de error y si se puede realizar la inicialización, se crea una ventana de ayuda del tipo *PHelpWindow* que presenta el tópico *ContextoAyuda* del stream direccionado por *FichAyuda*.

<sup>70</sup> Versión con pequeñas modificaciones de la Unit *HELPPFILE.TPU* de Turbo Vision.

## UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION

ContextoAyuda contendrá el valor del campo HelpCtx de la vista activa o hcIndiceAyuda si lo que se ha solicitado es un acceso al índice de tópicos de ayuda. Finalmente se anula el evento de solicitud de ayuda. En la figura 14.45 contiene la ventana de ayuda sensible al contexto que se presenta cuando está seleccionado el menú **Agenda**, así como la indicación (*hint*) que aparece en la línea de estado para este menú. Esta figura presenta dos instantes diferentes de la aplicación, ya que la indicación del menú agenda no aparecerá en la línea de estado cuando se solicite ayuda. En este caso se visualizaría la indicación correspondiente a la ventana de ayuda si estuviese definida.

### Ejemplo 14.43.

```
procedure TGestionFichas.GetEvent(var Evento: TEvent);
const
 AyudaEnUso: Boolean = False;
var
 V: PWindow;
 ContextoAyuda: Word;
 FichAyuda: PHelpFile;
 StrmAyuda: PDosStream;
begin
 inherited GetEvent(Evento);
 if Evento.What = evCommand then
 case Evento.Command of
 cmAyuda, cmIndiceAyuda:
 if not AyudaEnUso then
 begin
 AyudaEnUso := True;
 StrmAyuda := New(PDosStream, Init(FicheroAyuda, stOpenRead));
 FichAyuda := New(PHelpFile, Init(StrmAyuda));
 if StrmAyuda^.Status <> stOk then
 begin
 MessageBox('No se puede abrir el fichero de ayuda.',
 nil, mfError + mfOkButton);
 Dispose(FichAyuda, Done);
 end
 else
 begin
 if (evento.command = cmAyuda) then
 ContextoAyuda := GetHelpCtx
 else
 ContextoAyuda := hcIndiceAyuda;
 V := New(PHelpWindow, Init(FichAyuda, ContextoAyuda));
 if ValidView(V) <> nil then
 begin
 ExecView(V);
 Dispose(V, Done);
 end;
 ClearEvent(Evento);
 end;
 AyudaEnUso := False;
 end; { Fin de CmAyuda, cmIndiceAyuda }
 end; { Fin CASE }
 end;
end;
```

### Ejemplo 14.44.

## MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVENTOS CON TURBO VISION

```
{ Función para calcular el nombre del fichero de Ayuda }
function FicheroAyuda: PathStr;
var
 NombreProgr: PathStr;
 Dir: DirStr;
 Nombre: NameStr;
 Ext: ExtStr;
begin
 if Lo(DosVersion) >= 3 then NombreProgr := ParamStr(0)
 else NombreProgr := FSearch('AGENDA.EXE', GetEnv('PATH'));
 FSplit(NombreProgr, Dir, Nombre, Ext);
 if Dir[Length(Dir)] = '\' then Dec(Dir[0]);
 FicheroAyuda := FSearch('AYUDAAGE.HLP', Dir);
end;
```

### 14.9 EJERCICIOS PROPUESTOS

- 14.1 Añadir un comando al programa AGENDA que permita almacenar en un recurso los colores de la aplicación.
- 14.2 Cambiar las opciones del programa sin modificarlo para que permita:
  - Presentar los menús en inglés.
  - Obtener una versión reducida de la aplicación.
- 14.3 Heredar un tipo objeto del calendario de la unit *calendar.tpu* que tenga asociada una colección para poder definir las fiestas de un año. Asociarle los métodos necesarios para poder actualizar las fiestas cada año, manteniendo en un histórico las fiestas de años anteriores. Presentar las fiestas en un color que resalte. Los domingos se incluirán automáticamente como festivos.
- 14.4 Añadir a la ventana de fichas del programa AGENDA.PAS un campo lista que permita seleccionar la provincia de la persona que se introduce en la agenda. Utilizar el ejemplo del fichero LISTA.PAS.
- 14.5 Examinar el fichero EDITOR3.PAS que implementa un editor de ficheros de texto en el cual el nº de ventana de edición que se habrá es el siguiente al mayor de las ventanas abiertas, teniendo en cuenta si se ha cerrado alguna. Incorporar el mecanismo a las ventanas editor del programa AGENDA.PAS.
- 14.6 Modificar el sistema de ayuda sensible al contexto de AGENDA asociándole un histórico que permita recorrer las ventanas de ayuda que se hayan desplegado desde que se inició la aplicación. Modificar también que la ventana de ayuda no sea modal, permitiendo seleccionar otros elementos de la aplicación mientras una ayuda está activa. No debe haber más de una ventana de ayuda abierta simultáneamente.
- 14.7 Añadir un control a la ventana de fichas que permita borrar un elemento de la colección. Cambiar el método que almacena la colección, para que no se almacena toda la colección cada vez que se salva la agenda de fichas, sino que se salven sólo las modificadas o añadidas.

## AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

- 14.8 Crear una aplicación que no tenga barra de menús. Los comandos se activarán en una caja de menú que pueda situarse en cualquier parte del desktop a instancias del usuario.
- 14.9 Construir una aplicación que utilice el método *Idle* para recoger caracteres de una puerta serie y presentarlos en una vista diseñada a tal efecto. Construir un tipo objeto vista que envíe por una puerta serie todos los caracteres que se envíen a esa vista. Integrar estos tipos objeto en un editor, de forma que disponga de correo electrónico mediante una vista de emisión y otra de recepción que coexisten con otras vistas de edición de ficheros.
- 14.10 En el ejercicio anterior utilizar un objeto *Clipboard* para poder llevar(trazer) de él texto a(de) la vista de emisión(recepción). Hacer uso de esta posibilidad para poder enviar y recibir ficheros.
- 14.11 Mediante la utilización de colecciones ordenadas de cadenas, modificar el editor de EDITOR3.PAS para que las palabras que se escriban en las vistas de edición de ficheros sean corregidas. El tipo objeto colección tendrá métodos que permitan introducir directamente nuevas palabras, así como editar y corregir las palabras almacenadas, insertar nuevas palabras que aparezcan en cualquier fichero (con confirmación o no del usuario) así como deshacer las últimas correcciones mediante un histórico. La colección se podrá almacenar y recuperar en un recurso, de forma que se pueden tener varios diccionarios (técnicos y de distintos idiomas).

### 14.10 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

En la obra *Object-Oriented Programming: An Introduction* (Greg Voss, McGraw-Hill 1991) se puede profundizar en programación orientada a objetos, y su aplicación en el desarrollo de aplicaciones con librerías de clases y marcos de aplicación (cubre C++, Turbo Pascal, Smalltalk/V, Actor, Turbo Vision, C++ Views y ObjectWindows).

En castellano, sobre marcos de aplicación y programación en Turbo Vision la bibliografía existente es mínima. El libro *Turbo Pascal 7. Manual de referencia* (traducción de la obra *Turbo Pascal 7: The Complete Reference*. Stephen K. O'Brien y Steve Nameroff, McGraw-Hill 1993) incluye un capítulo sobre Turbo Vision. No se explica ni los fundamentos de los marcos de aplicación ni las posibilidades de la librería Turbo Vision. Básicamente se limite a enumerar alguno de los tipos objeto de Turbo Vision y sus métodos principales, sin interrelacionar los objetos dentro de una jerarquía ni explicar la potencia de la aplicación de la programación orientada a objetos a la jerarquía de partida que ofrece el marco de aplicación.

Para llegar a los más oscuros rincones y profundizar en todos los aspectos del Turbo Vision, lo mejor es acudir al manual de Turbo Vision que acompaña a la versión 7.0 de Turbo Pascal, y del cual ya existe traducción desde enero de 1994. En él se puede estudiar desde los fundamentos e implementación de los tipos objetos, como la forma de utilizar hasta llegar al capítulo de referencia

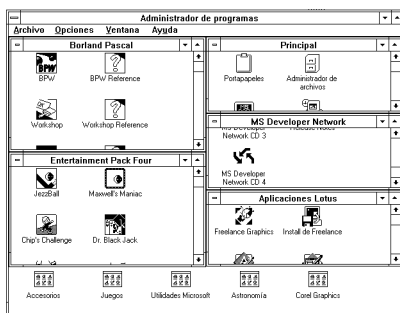
técnica que describe uno a uno, todos los tipos objeto, sus métodos, y otras rutinas disponibles. Quizá adolece de falta de pequeños ejemplos completos que ayuden a clarificar los distintos (y numerosos) aspectos que explica.

En el contexto de los lenguajes de programación, **persistencia** se refiere a la habilidad para conservar los valores de los datos o el estado de los objetos entre ejecuciones sucesivas de un programa. El primer lenguaje que utilizó esta idea fue probablemente *APL*, con su concepto de espacio de trabajo (*workspace*), y más tarde fue adoptado por otro lenguaje interactivo, *Smalltalk*. En estos dos casos, la inclusión de persistencia en el lenguaje fue un intento de crear un entorno de programación totalmente interactivo. Con *PS\_Algol*, se demostró que este principio no debía estar restringido a lenguajes interactivos, sino que, podría ser aplicado a una variedad de lenguajes que soportasen *memoria heap*. En el caso de Turbo Pascal, que permite la gestión de la memoria heap, con las extensiones de Turbo Vision se podría implementar la persistencia haciendo uso de los *streams* y los *recursos* (e.j. en el programa AGENDA se puede recuperar el estado del *desktop* que se tenía en una ejecución previa del programa, pudiéndose hacer lo mismo, si así se implementase, con cualquier otro objeto de la aplicación).

En los artículos *Layered implementation of persistent object stores* publicado en *Software Engineering Journal* (Marzo 1989) de *P. Balch, W.P. Cockshott y P.W. Foulk* y *PS\_Algol: An Algol with a persistent heap* publicado en *ACM TOPLAS* 17(7): 24-31, 1981 de *M.P. Atkinson, K.J. Chisholm y W.P. Cockshott* se puede profundizar en aspectos relativos al concepto de *persistencia*.

En este capítulo se ha analizado a fondo el marco de aplicación Turbo Vision en modo texto y se han mencionado otros en modo gráfico. Existen marcos de aplicación que permiten desarrollar programas tanto en modo texto como en modo gráfico y en bajo distintas plataformas o sistemas operativos con compatibilidad de código. Por ejemplo **Zinc Application Framework**® 3.5 de *Zinc Software Incorporated* permite desarrollar aplicaciones en C++ para Microsoft Windows, Windows NT, OS/2 2.0, UNIX Motif, DOS en modo gráfico y en modo texto con arquitectura orientada a objetos. **XVT**® de *XVT Software Inc* utilizando como lenguaje de base C/C++ pone al alcance la posibilidad de implementar programas bajo Windows, Windows NT, OS/2 2.0, Macintosh, OPEN LOOK, OSF/Motif y sistemas en modo carácter.

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS



## CAPITULO 15

# PROGRAMACION EN ENTORNO WINDOWS®

### CONTENIDOS

- 15.1 Interfaces gráficas de usuario
- 15.2 El entorno Windows
- 15.3 Programación dirigida por eventos
- 15.4 Transición rápida a Windows
- 15.5 Tipos de datos en Windows
- 15.6 La biblioteca ObjectWindows
- 15.7 Los recursos
- 15.8 Las funciones API de Windows
- 15.9 Los mensajes
- 15.10 El portapapeles
- 15.11 Las bibliotecas de enlace dinámico (DLL)
- 15.12 Intercambio dinámico de datos (DDE)
- 15.13 Objetos de enlace e inclusión (OLE)
- 15.14 Ejercicios resueltos
- 15.15 Ejercicios propuestos
- 15.16 Ampliaciones y notas bibliográficas



### 15.1 INTERFACES GRAFICOS DE USUARIO

Los interfaces gráficos de usuario o GUI (siglas en inglés de *Graphics User Interface*) permiten el manejo de los programas de una forma más intuitiva y cómoda a los usuarios, por medio de gráficos, iconos y menús de ayuda. En la figura 15.1 se muestra un ejemplo de GUI.

Los conceptos básicos de los GUI fueron desarrollados a mediados de los 70 en el centro de Investigación de Xerox en Palo Alto (PARC, *Palo Alto Research Center*), dentro del proyecto liderado por *Alan Kay*, y denominado *Dynabook*, dentro del cual también se desarrollo el ratón y el lenguaje orientado a objetos puro *Smalltalk*.

Las ideas desarrolladas por Xerox fueron aprovechadas por *Apple* a principios de los 80 para su ordenador *Lisa*, pero su popularización no llegó hasta el año 1984 con el lanzamiento al mercado del ordenador *Macintosh*. Desde entonces los GUI han proliferado en todos los entornos y tipos de ordenadores. Ejemplos de GUI son: Windows en entornos MS-DOS, Windows/NT, OS/2, GEM para ATARI, X-Windows para UNIX, MOTIF para OSF/1, NeXT para NextStep, etc...

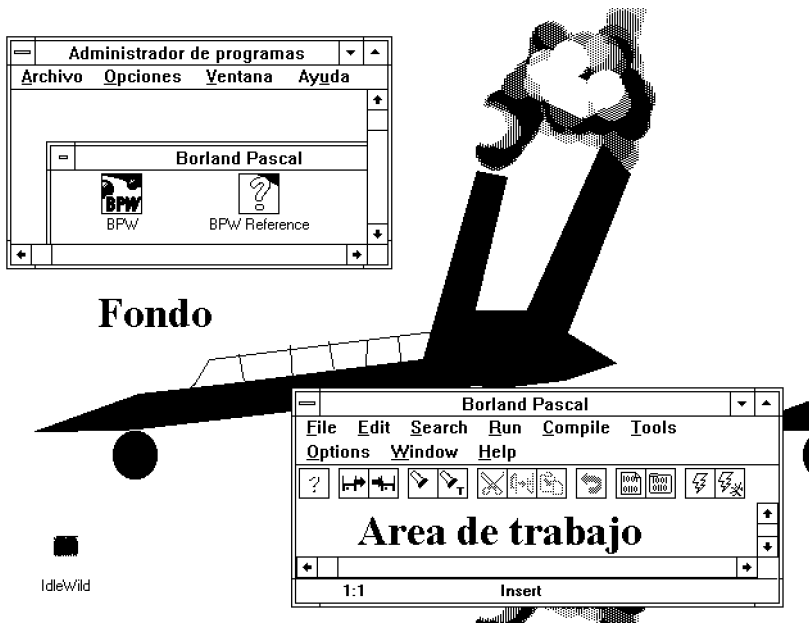


Figura 15.1 Interfaz gráfico de usuario de Windows<sup>71</sup>

<sup>71</sup> Fondo diseñado por Guillermo Cueva de 7 años

Actualmente el interfaz gráfico de usuario es el consenso más importante en la industria de los ordenadores. Aunque los distintos entornos gráficos difieren en detalles, casi todos tienen características similares, habitualmente dadas por las especificaciones IBM SAA/CUA (*Systemes Application Architecture/Common User Access*). Las especificaciones IBM SAA/CUA dictan distintas normas sobre diseño de menús, aceleradores de teclado, cajas de diálogo, y otros aspectos de las GUI.

Este capítulo trata sobre la programación en Borland® Pascal con el interfaz gráfico de usuario Windows®, desarrollado por Microsoft®.

## 15.2 EL ENTORNO WINDOWS

Windows ofrece al usuario un entorno de ventanas multitarea basado en gráficos, que ejecuta programas especialmente diseñados para Windows, y también programas diseñados para MS-DOS.

Los programas escritos para Windows tienen una apariencia y estructura de menús muy similar, además son más fáciles de usar y de aprender que los programas convencionales para MS-DOS. También pueden intercambiar datos entre ellos, por medio del uso del portapapeles (*clipboard*), de las bibliotecas de enlace dinámico (*DLL, Dynamic Link Libraries*), del intercambio dinámico de datos (*DDE, Dynamic Data Exchange*), y de los objetos de enlace e inclusión (*OLE, Object Linking and Embedding*).

El interfaz con el usuario se compone de varios elementos gráficos en forma de iconos, fondos, ventanas y dispositivos de entrada, como pueden ser teclas, cajas de diálogo, barras de menú y barras de desplazamiento. Utilizando el teclado o el ratón, el usuario puede manejar directamente estos elementos gráficos en la pantalla. Las ventanas y otros elementos gráficos pueden ser arrastrados, redimensionados, y colocados en distintas posiciones de la pantalla. En la figura 15.2 se muestran los distintos componentes de una ventana Windows.

Aunque Windows está diseñado principalmente para ejecutar aplicaciones Windows, también puede ejecutar programas para MS-DOS. Aunque estos últimos se ejecutan dentro de una ventana de Windows, sin poder utilizar todas las características de Windows. En algunos casos pueden ser desplegados en ventanas y usados en multitarea junto con programas Windows. El *Manual del Usuario* de Windows se refiere a estos programas como aplicaciones *No-Windows*. Los programas MS-DOS desde el punto de vista de su utilización en Windows se pueden dividir en dos categorías: las aplicaciones de "buen comportamiento en Windows", y las de "mal comportamiento en Windows".

Las aplicaciones MS-DOS de "buen comportamiento" en Windows son aquellas que utilizan las interrupciones software del MS-DOS y de la ROM BIOS (*Basic Input/Output System*: sistema básico de entrada/salida) para leer del teclado o escribir en pantalla, que es el método que usan las *units* de Turbo Pascal y Borland Pascal. Estos programas pueden funcionar generalmente en una ventana de Windows y con multitarea.

## PROGRAMACION DIRIGIDA POR EVENTOS

Las aplicaciones MS-DOS de "mal comportamiento" en Windows son aquellas que escriben directamente en la pantalla de video o toman directamente el control del teclado por medio de interrupciones hardware. Estos programas pueden tener problemas para ejecutarse en una ventana o hacer funcionar la multitarea.

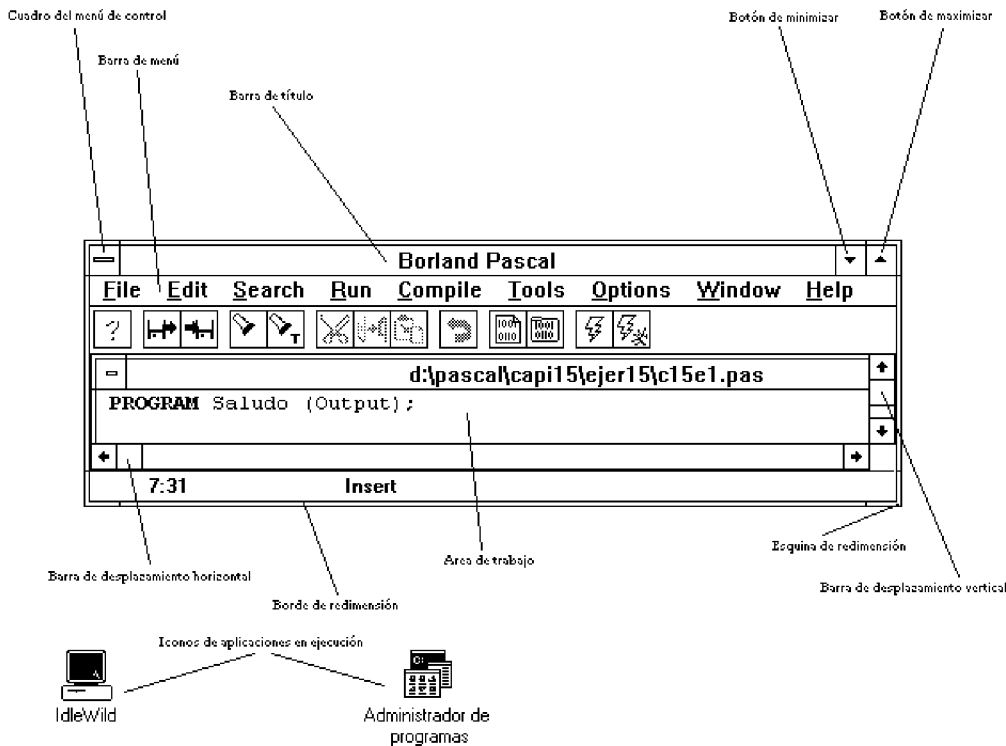


Figura 15.2 Elementos de una ventana de Windows

### 15.3 PROGRAMACION DIRIGIDA POR EVENTOS

Los conceptos básicos de la programación dirigida por eventos ya se introdujeron en el capítulo 14, y son válidos para la programación del entorno Windows, dado que Windows es un entorno dirigido por eventos. Uno de los principales problemas para la comprensión de los programas en el entorno Windows, es el cambio de la programación tradicional a la programación dirigida por eventos.

La programación tradicional se basa en las estructuras de control de flujo secuenciales, alternativas y repetitivas. Sin embargo la programación en el entorno Windows está dirigida por eventos. Es decir las acciones de los programas no se realizan siguiendo las estructuras de control tradicionales, sino que dichas acciones se ejecutan según ocurran o no una serie de acontecimientos (o eventos) generados por el usuario del programa o por el propio sistema Windows.

Un evento es cualquier acción que se efectúa en el entorno Windows. Por ejemplo la pulsación de una tecla, los movimientos del ratón, el *click* del ratón, etc... El entorno Windows captura los eventos y los traduce en mensajes.

Los mensajes son estructuras de datos que contienen información relacionada con un evento. Los mensajes son enviados por Windows a las distintas aplicaciones cargadas en el entorno. Cada aplicación procesa los mensajes que le afectan.

Como conclusión se puede decir que en la programación dirigida por eventos la secuencia de ejecución de instrucciones de cada aplicación depende de los mensajes que recibe la aplicación.

#### 15.4 TRANSICION RAPIDA A WINDOWS

Borland Pascal contiene tres *units* (*WinCrt*, *WinDos*, y *WinPrn*) que permiten una transición rápida de los programas desarrollados para DOS, para su ejecución en Windows. El inconveniente de estas *units* es que tan sólo permiten la emulación de la pantalla del DOS con una ventana de Windows. Si se desea el manejo completo del entorno Windows (ratón, menús, etc...) deben utilizarse las *units* de la biblioteca *ObjectWindows* o las funciones API (*Application Program Interface*) de Windows.

| Igual para ambas | Sólo en <i>Crt</i>  | Sólo en <i>WinCrt</i> |
|------------------|---------------------|-----------------------|
| AssignCrt        | Delay <sup>72</sup> | CursorTo              |
| ClrEol           | DellLine            | DoneWinCrt            |
| ClrScr           | HighVideo           | InitWinCrt            |
| GotoXY           | InsLine             | ReadBuf               |
| KeyPressed       | LowVideo            | ScrollTo              |
| ReadKey          | NormVideo           | TrackCursor           |
| WhereX           | NoSound             | WriteBuf              |
| WhereY           | Sound               | WriteChar             |
|                  | TextBackground      |                       |
|                  | TextColor           |                       |
|                  | TextMode            |                       |
|                  | Window              |                       |

Tabla 15.1 Diferencias entre las *units Crt* y *WinCrt*

<sup>72</sup> Véase una versión de *Delay* para Windows en el ejercicio resuelto 15.1.

## LA UNIT WINCRT

La *unit WinCrt* utiliza una ventana con desplazamientos horizontales y verticales para emular la pantalla del DOS. Esta *unit* contiene procedimientos y funciones compatibles con los de la *unit Crt* usada en las aplicaciones DOS. En la tabla 5.1 se muestra una comparación entre la *units Crt* y *WinCrt*.

### Ejemplo 15.1

Escribir el mensaje *Hola a todos*, en una ventana de Windows.

**Solución.** Utilizando la *unit WinCrt* se escribe el siguiente programa para Borland Pascal para Windows.

```
PROGRAM Saludo (Output);
USES WinCrt;
BEGIN
 Writeln ('Hola a todos');
END.
```

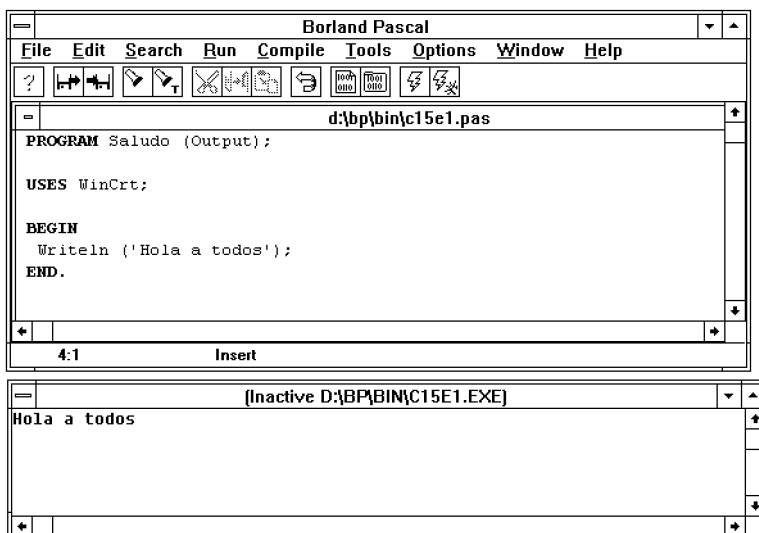


Figura 15.3 Ejecución del programa del ejemplo 15.1

El entorno de desarrollo de Borland Pascal para Windows y la ejecución del programa se muestran en la figura 15.3. La ejecución del programa se puede explicar siguiendo los pasos que se indican a continuación:

- *Apertura de la ventana.* El procedimiento *WriteIn* con la *unit WinCrt* abre una ventana implícitamente. También se puede abrir implícitamente con los procedimientos *Write*, *Read*, y *Readln* sobre la entrada y salida estándar. Si se desea abrir explícitamente la ventana se debe usar el procedimiento *InitWinCrt*. La posición y el tamaño inicial de la pantalla están predeterminados por unos valores por omisión en las variables *WindowOrg*, y *WindowSize* de la *unit WinCrt*. La ventana se puede modificar de tamaño interactivamente utilizando el ratón.
- *Se establece automáticamente el título de la ventana con el path y el nombre del fichero ejecutable.*
- *Se realiza la ejecución del programa.*
- *Cuando el programa finaliza su ejecución, la ventana queda inactiva.* En el título de la ventana aparece la palabra *inactive*. También se podría definir el título de la ventana inactiva con la variable *InactiveTitle* de la *unit WinCrt*.

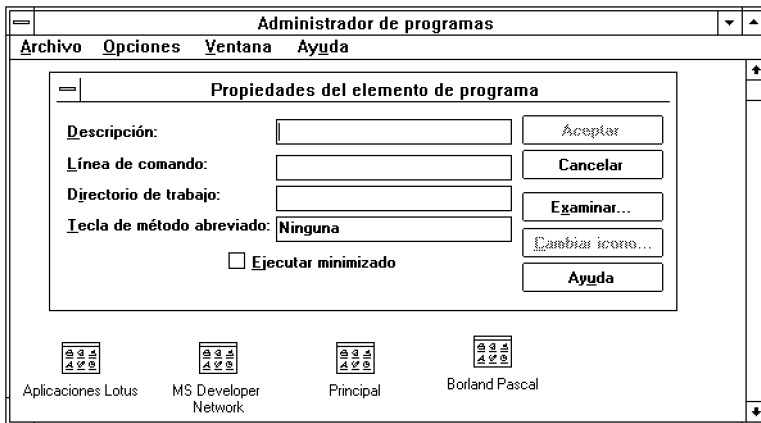


Figura 15.4 Definición de las propiedades de un programa Windows

- *Recorrer la ventana.* El usuario puede desplazar y recorrer los contenidos de la pantalla utilizando las barras de desplazamiento horizontal y vertical.

TRANSICION RAPIDA A WINDOWS

- *Cierre de la ventana.* La ventana se puede cerrar, y con ello el programa, haciendo *click* con el ratón en la esquina superior izquierda, y eligiendo cerrar. También se puede hacer directamente pulsando las teclas **(Alt)+F4** o añadiendo el procedimiento *DoneWinCrt* en alguna parte del programa.
- *Asignar un icono al programa ejecutable.* Se puede asociar un icono al programa con las herramientas estándar del entorno Windows, para lo cual situados en la ventana de Windows *Administrador de programas*, se elige *Archivo*, después *Nuevo*, y luego *Elemento de programa*. Entonces aparece la ventana de la figura 15.4, se debe rellenar como mínimo la caja de *Línea de comando*, indicando en ella el *path* y el nombre del programa ejecutable. Para asignarle un icono, se pulsa el botón de **Cambiar icono...**, y se elige el que se desee. Por último se elige **Aceptar**.
- *Ejecutar el programa pulsando el icono.* Tan sólo debe hacerse *click* con el ratón sobre el icono que representa la aplicación.

| Igual o similar para ambas | Sólo en <i>Dos</i> | Sólo en <i>WinDos</i> |
|----------------------------|--------------------|-----------------------|
| DiskFree                   | DosExitCode        | CreateDir             |
| DiskSize                   | EnvCount           | FileExpand            |
| DosVersion                 | EnvStr             | FileSearch            |
| FindFirst                  | Exec               | FileSplit             |
| FindNext                   | Fexpand            | GetArgCount           |
| GetCBreak                  | FSearch            | GetArgStr             |
| GetDate                    | Fsplit             | GetCurDir             |
| GetEnvVar                  | Keep               | RemoveDir             |
| GetFAttr                   | SwapVectors        | SetCurDir             |
| GetFTime                   |                    |                       |
| GetIntVec                  |                    |                       |
| GetTime                    |                    |                       |
| GetVerify                  |                    |                       |
| Intr                       |                    |                       |
| MsDos                      |                    |                       |
| PackTime                   |                    |                       |
| SetCBreak                  |                    |                       |
| SetDate                    |                    |                       |
| SetFAttr                   |                    |                       |
| SetIntVec                  |                    |                       |
| SetTime                    |                    |                       |
| SetVerify                  |                    |                       |
| UnpackTime                 |                    |                       |
| SetFtime                   |                    |                       |

Tabla 15.2 Diferencias y similitudes entre las *units Dos* y *WinDos*

## LA UNIT WINDOS

La *unit WinDos* sustituye a la *unit Dos*, cuando se escriben aplicaciones que se van a ejecutar en el entorno Windows. La *unit WinDos* no incluye todos los procedimientos y funciones de la *unit Dos*, consultar la tabla 15.2.

La *unit WinDos* se ha desarrollado siguiendo la forma de las funciones de Windows, por tanto muchos nombres de tipos y constantes son diferentes, aunque se han mantenido nombres muy parecidos. Las cadenas de caracteres declaradas en la *unit Dos* como de tipo *string*, en la *unit WinDos* son del tipo *PChar* (puntero a carácter).

### Ejemplo 15.2

Escribir un programa que muestre el espacio total y libre del disco actual por defecto.

**Solución.** Utilizando Borland Pascal para Windows, con las *units WinCrt* y *WinDos*.

```
PROGRAM EspacioEnDisco(Output);
USES WinCrt, WinDos;
BEGIN
 GotoXY(10,2);
 Writeln ('Situación actual del disco');
 GotoXY(3,4);
 Writeln ('Espacio libre = ',DiskFree(0) div 1024,' Kbytes');
 GotoXY(3,6);
 Writeln ('Tamaño total del disco = ',DiskSize(0) div 1024, ' Kbytes');
 REPEAT
 UNTIL KeyPressed; (* Espera a que se pulse una tecla *)
 DoneWinCrt; (* Cierra la ventana *)
END.
```

La ejecución puede verse en la figura 15.5.

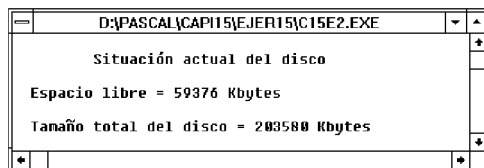


Figura 15.5 Ejecución del programa del ejemplo 15.2



## TRANSICION RAPIDA A WINDOWS

### LA UNIT WINPRN

La *unit WinPrn* sustituye a la *unit Printer* del Dos, y permite enviar la salida de un programa en Windows a la impresora. Se puede elegir la impresora, los fuentes y el título de la tarea.

| Función      | Descripción                                       |
|--------------|---------------------------------------------------|
| AbortPrn     | Detiene la impresión                              |
| AssignDefPrn | Asigna un fichero a la impresora por defecto      |
| AssignPrn    | Asigna un fichero de texto a una impresora        |
| SetPrnFont   | Inicia la impresión con la fuente elegida         |
| TitlePrn     | Da un título de tarea para el gestor de impresión |

Tabla 15.3 Funciones de la *unit WinPrn*

### Ejemplo 15.3

Escribir un programa que lea el nombre de un fichero de texto y lo imprima por la impresora estándar de Windows.

**Solución.** Se define la situación inicial y el tamaño de la ventana, el directorio actual, y se imprime un fichero de texto. Véase que para el manejo de cadenas de caracteres con *Pchar*, es necesario reservar memoria dinámica *heap* con *GetMem*.

```
PROGRAM ImpresionDeFicheros(Input,Output,fichero);
USES WinPrn, WinCrt, WinDos;
VAR
 fichero, impresora: Text;
 nombre: ARRAY[0..80] OF Char;
 linea: STRING;
 dirActual: Pchar;

BEGIN
 (* Define la situación inicial de la ventana *)
 WindowOrg.x:=100;
 WindowOrg.y:=100;

 (* Define el tamaño inicial de la ventana *)
 WindowSize.x:=500;
 WindowSize.y:=20;

 (* Abre la ventana *)
 InitWinCrt;

 (* Determina el directorio actual *)
 (* Reserva de memoria heap para dirActual *)
 GetMem(dirActual, 80);
 GetCurDir(dirActual,0);
 Writeln('El directorio actual: ',dirActual);
 Write('Nombre del fichero a imprimir: ');
 Readln(nombre);
 Assign(fichero, nombre);
 Reset(fichero);
```

```

(* Abre la impresora por defecto *)
AssignDefPrn(impresora);
TitlePrn(impresora, nombre);
Rewrite(impresora);
Writeln('Imprimiendo ', nombre);

WHILE NOT Eof(fichero) DO
BEGIN
 Readln(fichero, linea);
 Writeln(impresora, linea);
 (* Fin de impresión si el usuario pulsa ESC *)
 IF KeyPressed AND (ReadKey = #27) THEN
 BEGIN
 AbortPrn(impresora);
 Break;
 END;
END;

Writeln('Fin de impresión');
Close(fichero); (* Cierra ficheros *)
Close(impresora);
DoneWinCrt; (* Cierra la ventana *)
END.

```

## 15.5 TIPOS DE DATOS DE WINDOWS

El entorno de programación Windows añade nuevos tipos de datos a los ya manejados en los capítulos anteriores. Es necesario una visión preliminar de estos tipos de datos, para comprender las declaraciones de los tipos *object* de *ObjectWindows*. En la tabla 15.3 se muestran los tipos de datos simples de Windows más utilizados por *Borland Pascal* para Windows, tanto para la biblioteca *ObjectWindows* como para las *funciones API*<sup>73</sup> de Windows. Para utilizar estos tipos de datos es necesario incluir la *unit WinTypes*. Para obtener una relación de todos los tipos de datos simples y estructuras de datos de *WinTypes* consultar la ayuda en línea de la *unit WinTypes*. Borland también suministra los fuentes de esta *unit*, en el directorio `\bp\rtl\win\wintypes.pas`.

Un *handle* o manejador es un número entero que se asocia con la interfaz de un objeto, tal como una ventana, una caja de diálogo, o cualquier objeto de control, que se corresponde con elemento sobre la pantalla.

Los *handles* de Windows se pueden comparar con los manejadores de ficheros de Pascal, así en el capítulo 11 se vio como para manejar un fichero en Pascal era necesario declarar una variable de tipo *file*, que debía asociarse a un nombre de fichero con la instrucción *Assign* de Turbo Pascal. Esta variable de tipo *file* sería el manejador de fichero en MS-DOS. En Windows los *handles* se generalizan y se utilizan para manejar cualquier objeto de control del entorno Windows.

El valor del número entero del *handle* no tiene interés para el programador, es como en el caso de la variable de tipo *file*, en el ejemplo de manejo de ficheros en MS-DOS, no interesa el valor que contiene, sino lo que representa.

---

<sup>73</sup> Las funciones API (*Applications Programming Interface*) se estudian en el apartado 15.8 de este capítulo.

## TIPOS DE DATOS DE WINDOWS

Las aplicaciones Windows trabajan con ventanas, cada ventana de Windows se identifica con un *handle*, de tipo *HWnd* (*handle* de ventana). Muchas de las funciones de Windows requieren como primer argumento un valor de tipo *HWnd*, para indicar a qué ventana se aplica la función. Si un programa crea varias ventanas, cada ventana tiene un *handle* distinto.

| Tipo     | Significado                                 |
|----------|---------------------------------------------|
| Bool     | Tipo booleano                               |
| HBitmap  | <i>Handle</i> de un mapa de bits            |
| HBrush   | <i>Handle</i> de un pincel                  |
| HCursor  | <i>Handle</i> de un cursor                  |
| HDC      | <i>Handle</i> de un dispositivo de contexto |
| HFont    | <i>Handle</i> de una fuente de caracteres   |
| HIcon    | <i>Handle</i> de un icono                   |
| HMenu    | <i>Handle</i> de un menú                    |
| HPalette | <i>Handle</i> de una paleta de colores      |
| HPen     | <i>Handle</i> de una pluma                  |
| HRgn     | <i>Handle</i> de una región                 |
| HStr     | <i>Handle</i> de una cadena de caracteres   |
| HWnd     | <i>Handle</i> de una ventana                |
| LPVoid   | Puntero largo genérico                      |
| LPHandle | Puntero largo a un <i>handle</i>            |
| PBool    | Puntero a tipo booleano                     |
| PByte    | Puntero a byte                              |
| PHandle  | Puntero a <i>handle</i>                     |
| PInteger | Puntero a entero                            |
| PLong    | Puntero a entero largo                      |
| PStr     | Puntero a <i>String</i>                     |
| PWord    | Puntero a <i>Word</i>                       |
| THandle  | Tipo genérico de <i>handle</i>              |

Tabla 15.4 Tipos de datos de Windows

Para escribir texto o crear gráficos dentro de una ventana se necesita manejar un *handle* de tipo *HDC* (*handle* de contexto de representación). El contexto de representación es el área donde se va a dibujar o escribir. Los contextos de representación tienen tres funciones:

- Asegurar que no se escribe texto o se dibujan gráficos fuera de la ventana
- Gestionar la selección de herramientas de dibujo (plumas, brochas,...) y las fuentes de escritura.

- Definir un área de trabajo independiente del dispositivo o periférico utilizado. Así por ejemplo se utilizan las mismas instrucciones para dibujar en una ventana o en una impresora.

Además de las ventanas y los contextos de representación, Windows tiene más elementos en su interfaz gráfica (mapas de bits, pinceles, cursores, fuentes de caracteres, iconos, menús, paletas de colores, etc...) que también se manejan con *handles*.

En Windows 3.x<sup>74</sup> los *handles* son enteros sin signo de 16 bits, es decir tienen un rango entre 0 y 65535, por lo que pueden manejar 65536 valores distintos. Sin embargo en Windows/NT<sup>75</sup> los *handles* son enteros largos sin signo de 32 bits, cuyo rango está entre cero y cuatro mil millones.

## LAS UNITS PARA PROGRAMAR EN WINDOWS

Los tipos de datos de Windows, la biblioteca de tipos objeto *ObjectWindows* y las funciones API de Windows están en las *units* mostradas en las tablas 15.5 y 15.6.

| Unit            | Contenidos                                                                        |
|-----------------|-----------------------------------------------------------------------------------|
| <i>Objects</i>  | Tipos objeto <i>TObject</i> , <i>TCollection</i> , <i>TStream</i>                 |
| <i>OWindows</i> | Tipos objeto <i>TApplication</i> , <i>TWindow</i> , <i>TScroller</i> y <i>MDI</i> |
| <i>ODialogs</i> | Tipos objeto de cajas, ventanas y controles de diálogo.                           |
| <i>OPrinter</i> | Tipos objeto de control de impresión                                              |
| <i>Validate</i> | Tipos objeto de validación de datos                                               |
| <i>BWCC</i>     | Controles al estilo Borland Windows                                               |
| <i>OStdDlgs</i> | Tipos objeto de cajas de diálogo, entradas, etc...                                |
| <i>OStdWnds</i> | Tipos objeto de edición de texto y ficheros                                       |
| <i>WinTypes</i> | Todos los tipos de datos manejados por las funciones API de Windows 3.0           |
| <i>WinProcs</i> | Todas las funciones y procedimientos de las funciones API de Windows              |

Tabla 15.5 *Units* de *ObjectWindows* y funciones API de Windows 3.0

La tabla 15.5 contiene las *units* necesarias para manejar *ObjectWindows* y las funciones API de Windows 3.0. Las *units* *OStdDlgs*, *OStdWnds*, y *OPrinter* de la biblioteca *ObjectWindows* tienen ficheros de recursos<sup>76</sup> asociadas a ellas. Los recursos de cada una de estas *units* es un fichero

<sup>74</sup> El conjunto de funciones API soportado por Windows 3.x se denomina Win16.

<sup>75</sup> Windows/NT soporta un conjunto de funciones API denominado Win32. Un subconjunto de Win32 es el Win32s (s de subset), que permite crear programas de 32 bits que se ejecutan bajo Windows 3.x. Win32s es una versión de Win16 con 32 bits, y sin las extensiones de Win32.

<sup>76</sup> Los recursos se estudian en el apartado 15.7 de este capítulo.

con el mismo nombre que la *unit*, pero con la extensión *.RES*. Los recursos se incluyen de forma automática en la *unit* cuando se usa. Una ventaja de los recursos es que para cambiarlos de idioma sólomente es necesario editar el fichero *\*.RES* con el taller de recursos (*resource workshop*) y traducirlo al lenguaje deseado, sin tener que manejar para nada los programas fuentes. Los recursos incorporados por defecto en las *units* de *ObjectWindows* traen los textos en inglés, si se desea traducirlos al castellano, uno de los caminos es traducir los ficheros de recursos (*\*.RES*) al castellano. Otro camino para poner los diálogos estándar en castellano es utilizar las funciones API de la *unit CommDlg* de Windows 3.1, que manejan directamente los diálogos del entorno Windows instalado.

La tabla 15.6 contiene las 11 *units* necesarias para manejar las funciones API incorporadas por Windows 3.1.

Una idea de la magnitud y potencia de estas *units* es que las funciones API de Windows 3.0 son más de 600 y las de Windows 3.1 son del orden de 400.

| Unit            | Contenidos                                      |
|-----------------|-------------------------------------------------|
| <i>CommDlg</i>  | Cajas de diálogos comunes                       |
| <i>DDEML</i>    | Mensajes de intercambio dinámico de datos (DDE) |
| <i>Dlgs</i>     | Constantes de caja de diálogo                   |
| <i>LZExpand</i> | Expansión de ficheros LZ                        |
| <i>MMSystem</i> | Extensiones multimedia                          |
| <i>OLE</i>      | Enlace e inclusión de objetos (OLE)             |
| <i>ShellAPI</i> | Funciones API de la <i>shell</i> de Windows     |
| <i>Stress</i>   | Comprobación estricta de tipos                  |
| <i>ToolHelp</i> | Depuración y otras herramientas                 |
| <i>Ver</i>      | Manejo de versiones                             |
| <i>Win31</i>    | Extensiones de Windows 3.1                      |

Tabla 15.6 *Units* de las funciones API de Windows 3.1

## 15.6 LA BIBLIOTECA OBJECTWINDOWS®

La biblioteca *ObjectWindows*<sup>77</sup> es una poderosa herramienta que facilita el desarrollo de aplicaciones Windows. Sin la biblioteca Windows el desarrollo de aplicaciones Windows es más duro para el programador, debido a la necesidad de desarrollar un código mucho más amplio y complejo.

<sup>77</sup> También denominada librería *ObjectWindows*, según se traduzca la palabra *library*.

La biblioteca *ObjectWindows* utiliza una combinación de programación orientada a objetos y programación dirigida por eventos para el desarrollo de aplicaciones Windows.

### LA JERARQUIA DE OBJECTWINDOWS

La biblioteca *ObjectWindows* es una jerarquía de tipos *object* que puede ser utilizada para manejar la mayor parte de las tareas de una aplicación Windows. Un esquema de la jerarquía de *ObjectWindows* puede verse en la figura 15.6, en la cual también se indican las *units* donde se encuentran los tipos *object*.

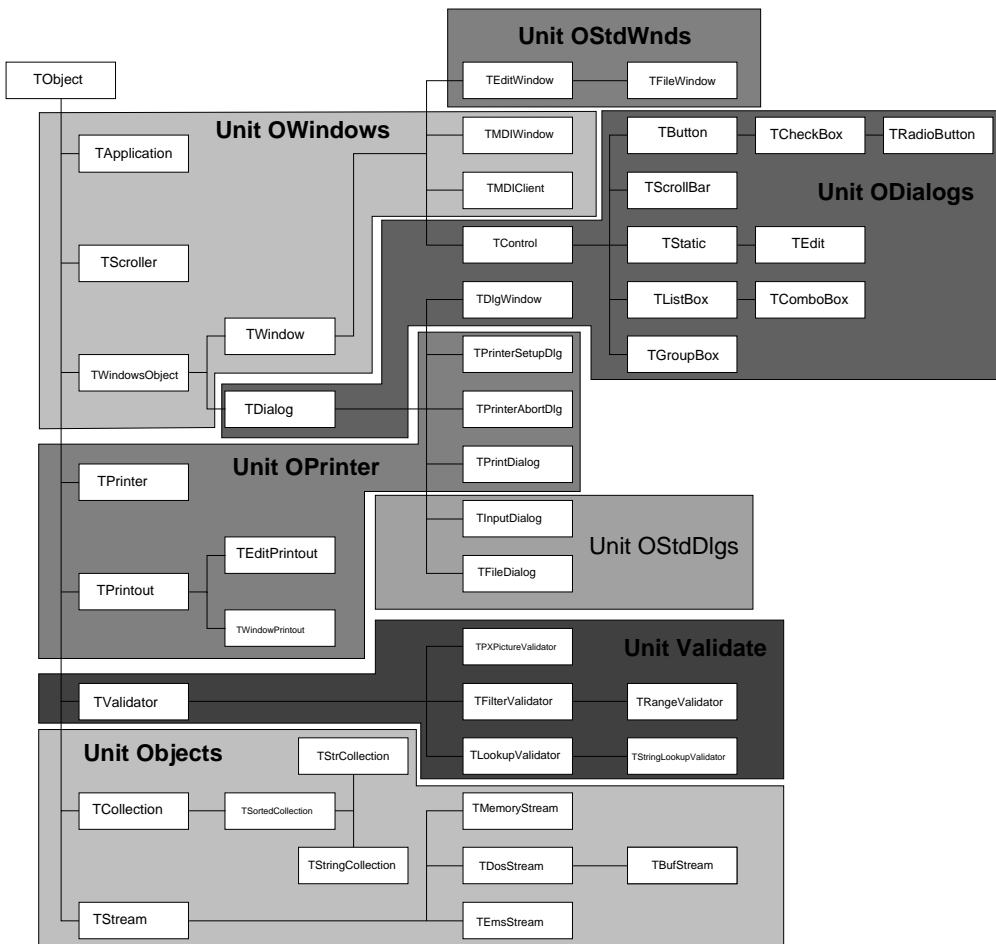


Figura 15.6 Esquema de la jerarquía *ObjectWindows*

### • El tipo objeto TObject

La jerarquía *ObjectWindows* tiene como tipo base el tipo *TObject*, es decir *TObject* es el antepasado común de todos los objetos de *ObjectWindows*. La declaración del tipo objeto *TObject* es la siguiente:

```
TObject = OBJECT
 CONSTRUCTOR Init;
 PROCEDURE Free;
 DESTRUCTOR Done; VIRTUAL;
END;
```

La declaración anterior indica que el tipo *TObject* es una clase abstracta, es decir no está diseñada para ser utilizada directamente, dado que no tiene campos y el número de métodos es mínimo. Este tipo objeto *TObject* está en la *unit Objects*.

### • El tipo objeto TApplication

El tipo objeto *TApplication* es un tipo derivado de *TObject*, y es también el tipo padre de cualquier aplicación desarrollada con *ObjectWindows*, se encuentra en la *unit OWindows*. Los campos y los métodos de *TApplication* son los componentes básicos para soportar una aplicación Windows mínima. La declaración del tipo objeto *TApplication* es la siguiente:

```
TApplication = ^TApplication;
TApplication = OBJECT(TObject)
 Status: Integer;
 Name: PChar;
 MainWindow: PWindowsObject;
 HAccTable: THandle;
 KBHandlerWnd: PWindowsObject;
 CONSTRUCTOR Init(AName: PChar);
 DESTRUCTOR Done; VIRTUAL;
 FUNCTION IdleAction: Boolean; VIRTUAL;
 PROCEDURE InitApplication; VIRTUAL;
 PROCEDURE InitInstance; VIRTUAL;
 PROCEDURE InitMainWindow; VIRTUAL;
 PROCEDURE Run; VIRTUAL;
 PROCEDURE SetKBHandler(AWindowsObject: PWindowsObject);
 PROCEDURE MessageLoop; VIRTUAL;
 FUNCTION ProcessAppMsg(var Message: TMsg): Boolean; VIRTUAL;
 FUNCTION ProcessDlgMsg(var Message: TMsg): Boolean; VIRTUAL;
 FUNCTION ProcessAccels(var Message: TMsg): Boolean; VIRTUAL;
 FUNCTION ProcessMDIAccels(var Message: TMsg): Boolean; VIRTUAL;
 FUNCTION MakeWindow(AWindowsObject: PWindowsObject): PWindowsObject; VIRTUAL;
 FUNCTION ExecDialog(ADialog: PWindowsObject): Integer; VIRTUAL;
 FUNCTION ValidWindow(AWindowsObject: PWindowsObject): PWindowsObject; VIRTUAL;
 PROCEDURE Error(ErrorCode: Integer); VIRTUAL;
 FUNCTION CanClose: Boolean; VIRTUAL;
END;
```

### Ejemplo 15.4

Desarrollo de una aplicación Windows mínima, con la biblioteca *ObjectWindows*.

**Solución.** Una aplicación mínima con *ObjectWindows* debe hacer tres cosas:

- Inicializarse

- Manejar mensajes
- Finalizar cuando se le indique

El tipo objeto *TApplication* maneja estas tareas por medio de tres métodos: *Init*, *Run* y *Done*. El programa principal de cualquier aplicación Windows desarrollada con *ObjectWindows* consiste exactamente en estos tres métodos.

A continuación se muestra el programa que construye la aplicación mínima:

```
PROGRAM AplicacionMinima(Output);
USES OWindows;
VAR
 MiAplica:TApplication;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

La ejecución del programa se muestra en la figura 15.7.

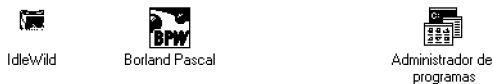
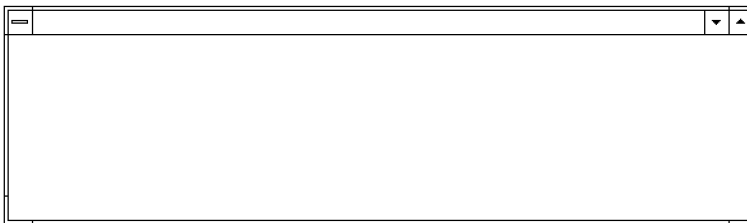


Figura 15.7 Ejecución del programa del ejemplo 15.4

A continuación se explicará lo que realmente ocurre cuando se ejecuta la aplicación, un esquema general se muestra en la figura 15.8.

- *El constructor Init*. La llamada al constructor *Init* implica:
  - § Construye el objeto *MiAplica*
  - § Inicializa los campos de datos del objeto *MiAplica*



§ También lleva a cabo dos tipos de inicializaciones:

- Llama al método *InitApplication* si no hay otras instancias de esta aplicación ejecutándose.
- Llama al método *InitInstance* siempre, que a su vez llama al método *InitMainWindow* para inicializar la ventana principal de la aplicación.

Cuando el método *Init* finaliza, la ventana principal de la aplicación está en la pantalla.

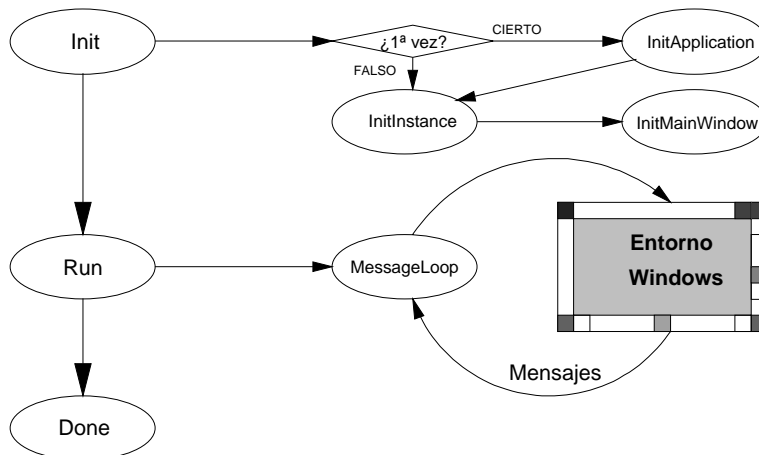


Figura 15.8 Esquema general de funcionamiento del ejemplo 15.4

- *El método Run.* La llamada al método *Run*, pone la aplicación en ejecución al llamar a su vez al método *MessageLoop*. El método *MessageLoop* procesa los mensajes procedentes del entorno *Windows*, es decir procesa las instrucciones que afectan directamente a cualquier aplicación en ejecución. *MessageLoop* es, como su nombre indica, un bucle que se ejecuta continuamente hasta que la aplicación finaliza. *MessageLoop* llama a su vez a varios métodos en función de los mensajes recibidos. Recordar que el entorno *Windows*, está dirigido por eventos, es decir cada uno de estos eventos envía un mensaje que es procesado por *MessageLoop*.
- *El destructor Done.* Cuando finaliza el método *Run*, se supone que debido a un mensaje enviado por el usuario para finalizar la aplicación (por ejemplo al pulsar las teclas **Alt**+**F4**). Entonces el destructor *Done* libera el objeto *MiAplica* de la memoria y cierra la aplicación.

**Ejemplo 15.5**

Modificar el ejemplo 15.4 para que aparezca un título de ventana en la ejecución de la aplicación.

**Solución.** El ejemplo 15.4 es el mínimo absoluto de una aplicación *ObjectWindows*, y no requiere la definición de nuevos tipos *object*. Sin embargo cuando se desarrolla una aplicación con *ObjectWindows*, es necesario definir un nuevo tipo objeto derivado del tipo objeto *TApplication*, y se redefinen algunos de los métodos de *TApplication*. Así para que aparezca un título de ventana es necesario redefinir el método *InitMainWindow* en el nuevo tipo objeto derivado de *TApplication*.

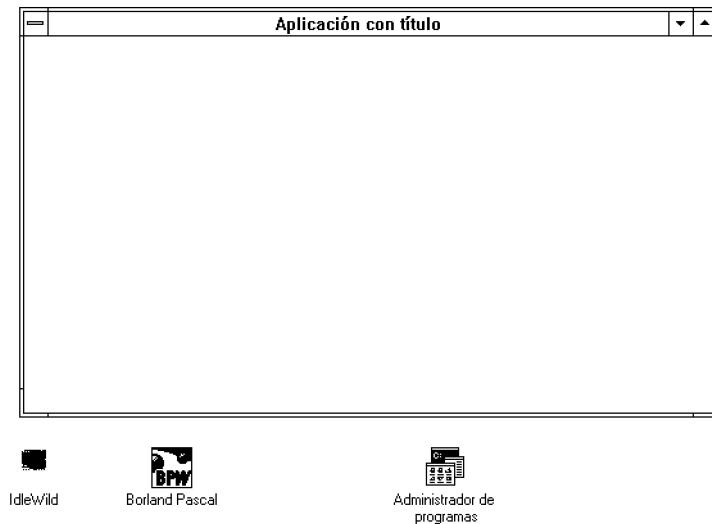


Figura 15.9 Ejecución del ejemplo 15.5

El código del programa se presenta a continuación:

```
PROGRAM AplicacionConTitulo(Output);
USES OWindows;
TYPE
 TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;
PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
 MainWindow:=New(PWindow, Init(NIL,'Aplicación con título'));
END;
```

## LA BIBLIOTECA OBJECTWINDOWS®

```
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

La ejecución se presenta en la figura 15.9.

En la instrucción:

```
MainWindow:=New(PWindow, Init(NIL,'Aplicación con título'));
```

se crea una nueva instancia del tipo objeto *TWindow*, donde *PWindow* es un puntero al tipo *TWindow*, e *Init* es un constructor de *TWindow*. El tipo *TWindow* es un descendiente del tipo *TWindowsObject* en la jerarquía de *ObjectWindows*. En el siguiente apartado se estudiará el tipo *TWindowsObject* y a continuación sus descendientes.

### Ejemplo 15.6

Modificar el ejemplo 15.5 para que el programa distinga la primera instancia del resto de las instancias.

**Solución.** La multitarea de Windows permite que una misma aplicación sea cargada más de una vez. Windows trata de economizar memoria de un modo bastante simple, pero eficaz. A cada aplicación le asigna un valor de *instancia*, así el programa puede detectar si hay instancias (es decir copias) previas de la misma aplicación que están ejecutándose. Si es así, no se carga nuevamente en memoria el segmento de código (que estaría duplicado innecesariamente). Esta forma de trabajo de Windows implica que el código de una aplicación debe permanecer invariable mientras se ejecuta la aplicación, ya que podría afectar a otras *instancias* de la misma aplicación. A continuación se escribe un programa que coloca el título de ventana '*Primera instancia*' para la primera vez que se ejecuta el programa, y el título '*Otra instancia*' para el resto de las instancias distintas de la primera. El código se presenta a continuación:

```
PROGRAM VariasInstancias(Output);
USES OWindows;
TYPE
 TMiAplicacion=OBJECT(TApplication)
 PrimeraVez: Boolean;
 PROCEDURE InitMainWindow; VIRTUAL;
 PROCEDURE InitApplication; VIRTUAL;
END;
PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
 IF PrimeraVez THEN
 MainWindow:=New(PWindow, Init(NIL,'Primera instancia'))
 ELSE
 MainWindow:=New(PWindow, Init(NIL,'Otra instancia'));
END;
```

```

PROCEDURE TMiAplicacion.InitApplication;
BEGIN
 PrimeraVez:= true;
END;

VAR
 MiAplica:TMiAplicacion;

BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.

```

La ejecución del programa se puede ver en la figura 15.10.

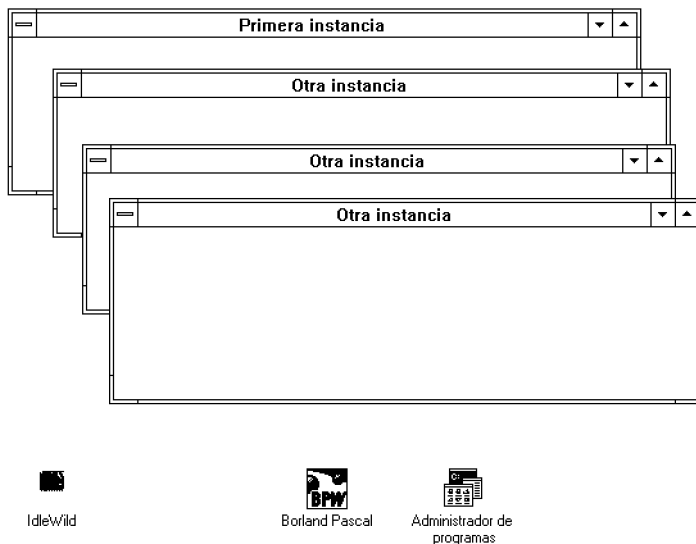


Figura 15.10 Ejecución del ejemplo 15.6

En el programa se define el tipo objeto *TMiAplicacion* como descendiente de *TApplication*, con un campo *PrimeraVez* de tipo booleano y se redefinen los métodos virtuales *InitMainWindow* e *InitApplication*. Según como se explicó en el ejemplo 15.5 *Init* llama a *InitApplication* la primera vez que se ejecuta, en el resto de las instancias *Init* llama a *InitInstance* que a su vez llama a *InitMainWindow* (véase figura 15.8).

#### • El tipo objeto *TWindowsObject*

El tipo objeto *TWindowsObject* también es un tipo derivado de *TObject*, como todos los de *ObjectWindows*, se encuentra en la *unit OWindows*. Este tipo objeto es el padre los tipos objeto:

## LA BIBLIOTECA OBJECTWINDOWS®

*TWindow* y *TDialog*, que junto con sus descendientes manejan las ventanas, cajas de diálogo, y distintos controles de Windows. La declaración siguiente indica que *TWindowsObject* es un tipo objeto bastante complejo, con gran cantidad de métodos tanto públicos como privados.

```
PWindowsObject = ^TWindowsObject;
TWindowsObject = OBJECT(TObject)
 Status: Integer;
 HWindow: HWnd;
 Parent, ChildList: PWindowsObject;
 TransferBuffer: Pointer;
 Instance: TFarProc;
 Flags: Byte;
 CONSTRUCTOR Init(AParent: PWindowsObject);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Store(var S: TStream);
 PROCEDURE DefWndProc(var Msg: TMessage); VIRTUAL;
 PROCEDURE DefCommandProc(var Msg: TMessage); VIRTUAL;
 PROCEDURE DefChildProc(var Msg: TMessage); VIRTUAL;
 PROCEDURE DefNotificationProc(var Msg: TMessage); VIRTUAL;
 PROCEDURE SetFlags(Mask: Byte; OnOff: Boolean);
 FUNCTION IsFlagSet(Mask: Byte): Boolean;
 FUNCTION FirstThat(Test: Pointer): PWindowsObject;
 PROCEDURE ForEach(Action: Pointer);
 FUNCTION Next: PWindowsObject;
 FUNCTION Previous: PWindowsObject;
 PROCEDURE Focus;
 FUNCTION Enable: Boolean;
 FUNCTION Disable: Boolean;
 PROCEDURE EnableKBHandler;
 PROCEDURE EnableAutoCreate;
 PROCEDURE DisableAutoCreate;
 PROCEDURE EnableTransfer;
 PROCEDURE DisableTransfer;
 FUNCTION Register: Boolean; VIRTUAL;
 FUNCTION Create: Boolean; VIRTUAL;
 PROCEDURE Destroy; VIRTUAL;
 FUNCTION GetId: Integer; VIRTUAL;
 FUNCTION ChildWithId(Id: Integer): PWindowsObject;
 FUNCTION GetClassName: PChar; VIRTUAL;
 FUNCTION GetClient: PMDIClient; VIRTUAL;
 PROCEDURE GetChildPtr(var S: TStream; var P);
 PROCEDURE PutChildPtr(var S: TStream; P: PWindowsObject);
 PROCEDURE GetSiblingPtr(var S: TStream; var P);
 PROCEDURE PutSiblingPtr(var S: TStream; P: PWindowsObject);
 PROCEDURE GetWindowClass(var AWndClass: TWndClass); VIRTUAL;
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE Show(ShowCmd: Integer);
 FUNCTION CanClose: Boolean; VIRTUAL;
 FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; VIRTUAL;
 PROCEDURE TransferData(Direction: Word); VIRTUAL;
 PROCEDURE DispatchScroll(var Msg: TMessage); VIRTUAL;
 PROCEDURE CloseWindow;
 PROCEDURE GetChildren(var S: TStream);
 PROCEDURE PutChildren(var S: TStream);
 PROCEDURE AddChild(AChild: PWindowsObject);
 PROCEDURE RemoveChild(AChild: PWindowsObject);
 FUNCTION IndexOf(P: PWindowsObject): Integer;
 FUNCTION At(I: Integer): PWindowsObject;
 FUNCTION CreateChildren: Boolean;
 FUNCTION CreateMemoryDC: HDC;
 PROCEDURE WMVScroll(var Msg: TMessage); VIRTUAL wm_First + wm_VScroll;
 PROCEDURE WMHScroll(var Msg: TMessage); VIRTUAL wm_First + wm_HScroll;
 PROCEDURE WMCommand(var Msg: TMessage); VIRTUAL wm_First + wm_Command;
 PROCEDURE WMClose(var Msg: TMessage); VIRTUAL wm_First + wm_Close;
```

## PROGRAMACION EN ENTORNO WINDOWS®

```
PROCEDURE WMDestroy(var Msg: TMessage); VIRTUAL wm_First + wm_Destroy;
PROCEDURE WMNCDestroy(var Msg: TMessage); VIRTUAL wm_First + wm_NCDestroy;
PROCEDURE WMActivate(var Msg: TMessage); VIRTUAL wm_First + wm_Activate;
PROCEDURE WMQueryEndSession(var Msg: TMessage);
 VIRTUAL wm_First + wm_QueryEndSession;
PROCEDURE CMExit(var Msg: TMessage); VIRTUAL cm_First + cm_Exit;
PRIVATE
 CreateOrder: Word;
 SiblingList: PWindowsObject;
END;
```

En la declaración del tipo objeto *TWindowsObject*, los métodos que manejan mensajes tienen una sintaxis aparentemente un poco especial, de la forma *VIRTUAL wm\_First+wm\_VScroll*. Lo que aparece después de la palabra *VIRTUAL* es una expresión entera, que representa el índice de la tabla de métodos dinámicos. Recuérdese que los métodos dinámicos se utilizan para tipos objeto con gran número de métodos virtuales. Para una explicación más amplia sobre el cálculo del índice de los métodos virtuales, véase el apartado 15.9 de este capítulo.

### • El tipo objeto TWindow

El tipo objeto *TWindow* es un descendiente de *TWindowsObject* que implementa una ventana genérica elemental. Esta ventana contiene el cuadro de menú de control, los botones de maximizar y minimizar, y las barras de desplazamiento (*scroll*). Las ventanas creadas con *TWindow* se pueden mover, cambiar de tamaño, minimizar y maximizar. Este tipo se encuentra en la *unit OWindows*. La declaración del tipo *TWindow* es la siguiente:

```
PWindow = ^TWindow;
TWindow = OBJECT(TWindowsObject)
 Attr: TWindowAttr;
 DefaultProc: TFarProc;
 Scroller: PScroller;
 FocusChildHandle: THandle;
 CONSTRUCTOR Init(AParent: PWindowsObject; ATitle: PChar);
 CONSTRUCTOR InitResource(AParent: PWindowsObject; ResourceID: Word);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Store(var S: TStream);
 PROCEDURE SetCaption(ATitle: PChar);
 PROCEDURE GetWindowClass(var AWndClass: TWndClass); VIRTUAL;
 PROCEDURE FocusChild;
 PROCEDURE UpdateFocusChild;
 FUNCTION GetId: Integer; VIRTUAL;
 FUNCTION Create: Boolean; VIRTUAL;
 PROCEDURE DefWndProc(var Msg: TMessage); VIRTUAL;
 PROCEDURE WMActivate(var Msg: TMessage);
 VIRTUAL wm_First + wm_Activate;
 PROCEDURE WMMDIActivate(var Msg: TMessage);
 VIRTUAL wm_First + wm_MDIActivate;
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE WMCreate(var Msg: TMessage);
 VIRTUAL wm_First + wm_Create;
 PROCEDURE WMHScroll(var Msg: TMessage);
 VIRTUAL wm_First + wm_HScroll;
 PROCEDURE WMVScroll(var Msg: TMessage);
 VIRTUAL wm_First + wm_VScroll;
 PROCEDURE WMPaint(var Msg: TMessage);
 VIRTUAL wm_First + wm_Paint;
 PROCEDURE Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); VIRTUAL;
 PROCEDURE WMSize(var Msg: TMessage);
```

## LA BIBLIOTECA OBJECTWINDOWS®

```
VIRTUAL wm_First + wm_Size;
PROCEDURE WMMove(var Msg: TMessage);
VIRTUAL wm_First + wm_Move;
PROCEDURE WMLButtonDown(var Msg: TMessage);
VIRTUAL wm_First + wm_LButtonDown;
PROCEDURE WMSysCommand(var Msg: TMessage);
VIRTUAL wm_First + wm_SysCommand;
PRIVATE
PROCEDURE UpdateWindowRect;
END;
```

El campo *Attr* almacena los atributos de las instancias de *TWindow*. Estos atributos se definen por el tipo registro *TWindowAttr*, que tiene la declaración siguiente:

```
TWindowAttr = RECORD
 Title: PChar;
 Style: LongInt;
 ExStyle: LongInt;
 X,Y,W,H: Integer;
 Param: Pointer;
 CASE Integer OF
 0: (Menu: HMenu);
 1: (Id: Integer);
 END;
```

El campo *Scroller* de *TWindow* es un puntero a una instancia del objeto *TScroller*, que maneja las barras de desplazamiento horizontal y vertical del contenido de la ventana. El tipo objeto *TScroller* se definirá posteriormente.

El tipo objeto *TWindow* tiene un gran número de métodos, entre ellos están los que se encargan de crear, mover, y redimensionar la ventana.

### Ejemplo 15.7

Escribir un programa que dibuje un círculo en la pantalla.

**Solución.** Se define el tipo objeto *TVentanaCirculo* descendiente del tipo *TWindow*, y se redefine el procedimiento *Paint*, dentro del cual se llama a la función *Ellipse*, que es una función API de Windows, que está en la *unit WinProcs*. Se incluye la *unit WinTypes* para el manejo de las definiciones de tipos de datos como *HDC* o *TPaintStruct*.

Obsérvese como es práctica común en la programación del entorno Windows, llamar a funciones (por ejemplo en el siguiente programa *Ellipse*) como si fueran procedimientos, ignorando el valor devuelto por la función. La función *Ellipse* tiene como parámetros el *handle* del dispositivo de contexto, y los vértices superior izquierdo e inferior derecho del rectángulo que circunscribe a la elipse.

```
PROGRAM Circulo (Output);
USES WinTypes, WinProcs, OWindows;

TYPE
 PVentanaCirculo=^TVentanaCirculo;
 TVentanaCirculo= OBJECT (TWindow)
 PROCEDURE Paint (PaintDC:HDC; VAR PaintInfo:TPaintStruct); VIRTUAL;
 END;
```

## PROGRAMACION EN ENTORNO WINDOWS®

```
TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
END;
PROCEDURE TVentanaCirculo.Paint;
 BEGIN
 (* La función siguiente está en WinProcs *)
 Ellipse(PaintDC, 10, 10, 100,100);
 END;
PROCEDURE TMiAplicacion.InitMainWindow;
 BEGIN
 MainWindow:=New(PVentanaCirculo, Init(NIL,'Dibujo de un círculo'));
 END;
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

La ejecución del programa se muestra en la figura 15.11.

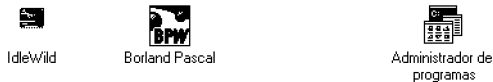
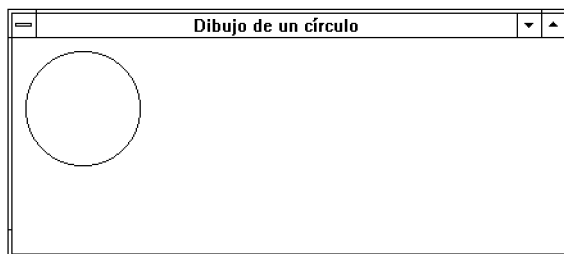


Figura 15.11 Ejecución del ejemplo 15.7

### Ejemplo 15.8

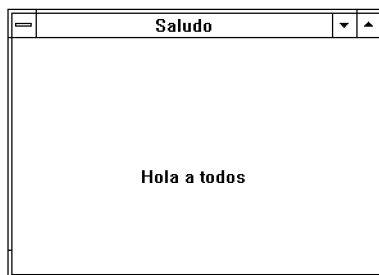
Escribir un programa que escriba el mensaje *Hola a todos* en la pantalla.

**Solución.** Es similar a la del ejemplo 15.7, con la función API *TextOut*. Los parámetros de *TextOut* son el *handle* del dispositivo de contexto, las coordenadas donde se va a comenzar a escribir, la cadena que se escribe (definida de tipo *Pchar*), y la longitud de la cadena. El código del programa se muestra a continuación, y la ejecución en la figura 15.12.



## LA BIBLIOTECA OBJECTWINDOWS®

```
PROGRAM Saludo(Output);
USES WinTypes, WinProcs, OWindows, Strings;
TYPE
 PVentanaSaludo=^TVentanaSaludo;
 TVentanaSaludo= OBJECT (TWindow)
 PROCEDURE Paint (PaintDC:HDC; VAR PaintInfo:TPaintStruct); VIRTUAL;
 END;
 TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;
 PROCEDURE TVentanaSaludo.Paint;
 VAR
 s:PChar;
 BEGIN
 GetMem(s,25);
 s:='Hola a todos';
 TextOut(PaintDC, 100, 100, s, StrLen(s));
 END;
 PROCEDURE TMiAplicacion.InitMainWindow;
 BEGIN
 MainWindow:=New(PVentanaSaludo, Init(NIL,'Saludo'));
 END;
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```



IdleWild



Borland Pascal



Administrador de programas

Figura 15.12 Ejecución del ejemplo 15.8

**Ejemplo 15.9**

Escribir un programa que escriba el mensaje *¡Hola a todos!* en la pantalla, cuando se pulsa el botón izquierdo del ratón sobre la ventana de la aplicación.

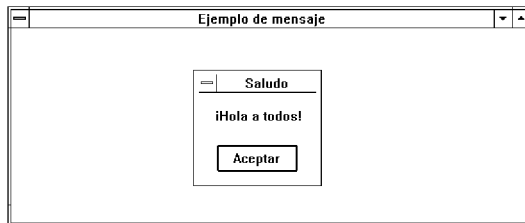


Figura 15.13 Ejecución del ejemplo 15.9

**Solución.** Es similar a la del ejemplo 15.8, pero en este caso se redefine la función *WMLButtonDown* del tipo objeto *TWindow*, para que envíe un mensaje a la pantalla cuando se pulsa el botón izquierdo del ratón. Para enviar el mensaje se utiliza la función API *MessageBox*. Los parámetros de *MessageBox* son: el *handle* de la ventana que ha recibido el *click* del ratón; la *primera* cadena es el mensaje que saldrá en el centro del cuadro; la *segunda* cadena es el título del cuadro; y la *constante mb\_OK* especifica que el cuadro debe incluir un sólo botón OK, representado en castellano por el botón **Aceptar**. Una explicación más amplia sobre el manejo de mensajes puede verse en el apartado 15.9 de este capítulo.

El código del programa se muestra a continuación, y la ejecución en la figura 15.13.

```
PROGRAM Saludo(Output);
USES OWindows, WinProcs, WinTypes;
TYPE
 PVentanaSaludo=^TVentanaSaludo;
 TVentanaSaludo= OBJECT (TWindow)
 PROCEDURE WMLButtonDown(VAR Msg:TMessage);
 VIRTUAL wm_First+wm_LButtonDown;
 END;
 TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;
 PROCEDURE TVentanaSaludo.WMLButtonDown;
 BEGIN
 MessageBox(HWindow, '¡Hola a todos!', 'Saludo', mb_OK);
 END;
 PROCEDURE TMiAplicacion.InitMainWindow;
 BEGIN
 MainWindow:=New(PVentanaSaludo, Init(NIL,'Ejemplo de mensaje'));
 END;
```

## LA BIBLIOTECA OBJECTWINDOWS®

```
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

### • El tipo objeto TEditWindow

El tipo objeto *TEditWindow* implementa un tipo objeto que soporta la entrada y edición de texto en una ventana. Las instancias *TEditWindow* incluyen la selección por menú de opciones de búsqueda y reemplazamiento de textos.

Este tipo objeto está dentro de la *unit OStdWnds*, y su declaración es la siguiente:

```
PEditWindow = ^TEditWindow;
TEditWindow = OBJECT(TWindow)
 Editor: PEdit;
 SearchRec: TSearchRec;
 CONSTRUCTOR Init(AParent: PWindowsObject; ATitle: PChar);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
 PROCEDURE WMSize(var Msg: TMessage);
 VIRTUAL wm_First + wm_Size;
 PROCEDURE WMSetFocus(var Msg: TMessage);
 VIRTUAL wm_First + wm_SetFocus;
 PROCEDURE CMEditFind(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditFind;
 PROCEDURE CMEditFindNext(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditFindNext;
 PROCEDURE CMEditReplace(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditReplace;
PRIVATE
 PROCEDURE DoSearch;
END;
```

El campo *Editor* es un puntero al tipo objeto *TEdit* de la jerarquía *ObjectWindows*. El tipo *TSearchRec* viene dado por la declaración siguiente:

```
TSearchRec = RECORD
 SearchText: array[0..80] of Char;
 CaseSensitive: Bool;
 ReplaceText: array[0..80] of Char;
 ReplaceAll: Bool;
 PromptOnReplace: Bool;
 IsReplace: Boolean;
END;
```

Los métodos del tipo *TEditWindow* se ocupan del manejo de los comandos de edición.

Las instancias de *TEditWindow* sólo pueden transferir información entre su ventana y el mundo exterior a través del portapapeles (*clipboard*). Para intercambiar datos con un fichero u otros periféricos de salida debe utilizarse tipos objeto descendientes de *TEditWindow*.

### Ejemplo 15.10

Escribir un programa que construya una pantalla de edición.

**Solución.** El código se muestra a continuación, y una ejecución del programa en la figura 15.14

```
PROGRAM PruebaEditor(Output);
USES OWindows, OStdWnds;
TYPE
 TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
END;

PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
 MainWindow:=New(PEditWindow, Init(NIL,'Prueba de editor'));
END;
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

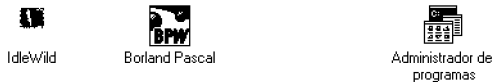
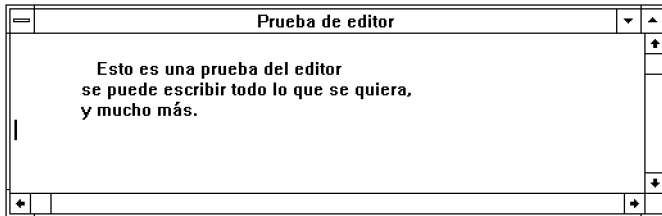


Figura 15.14 Ejecución del ejemplo 15.10

#### • El tipo objeto TFileWindow

El tipo objeto *TFileWindow* es un descendiente de *TEditWindow* y extiende las capacidades del tipo objeto padre para la edición de ficheros de texto. Los métodos que añade al tipo padre son: abrir (*Open*), leer (*Read*), escribir (*Write*), guardar (*Save*), y guardar como (*Save as*). Además las instancias de *TFileWindow* utilizan cajas de diálogo para abrir y guardar los ficheros.

Este tipo objeto está dentro de la *unit OStdWnds*, y su declaración es la siguiente:

## LA BIBLIOTECA OBJECTWINDOWS®

```
PFileWindow = ^TFileWindow;
TFileWindow = OBJECT(TEditWindow)
 FileName: PChar;
 IsNewFile: Boolean;
 CONSTRUCTOR Init(AParent: PWindowsObject; ATitle, AFileName: PChar);
 DESTRUCTOR Done; VIRTUAL;
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
 FUNCTION CanClear: Boolean; VIRTUAL;
 FUNCTION CanClose: Boolean; VIRTUAL;
 PROCEDURE NewFile;
 PROCEDURE Open;
 PROCEDURE Read;
 PROCEDURE SetFileName(AFileName: PChar);
 PROCEDURE ReplaceWith(AFileName: PChar);
 FUNCTION Save: Boolean;
 FUNCTION SaveAs: Boolean;
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE Write;
 PROCEDURE CMFileNew(var Msg: TMessage);
 VIRTUAL cm_First + cm_FileNew;
 PROCEDURE CMFileOpen(var Msg: TMessage);
 VIRTUAL cm_First + cm_FileOpen;
 PROCEDURE CMFileSave(var Msg: TMessage);
 VIRTUAL cm_First + cm_FileSave;
 PROCEDURE CMFileSaveAs(var Msg: TMessage);
 VIRTUAL cm_First + cm_FileSaveAs;
END;
```

### Ejemplo 15.11

Escribir un programa editor, que guarde los textos en un fichero.

**Solución.** Los textos se guardan siempre en el fichero *prueba.txt* al ir a cerrar la aplicación, tal y como se muestra en la figura 15.15. El código se expone a continuación:

```
PROGRAM PruebaEditorBis(Output);
USES OWindows, OStdWnds;
TYPE
 TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;
 PROCEDURE TMiAplicacion.InitMainWindow;
 BEGIN
 MainWindow:=New(PFileWindow, Init(NIL,'Prueba de editor', 'prueba.txt'));
 END;
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

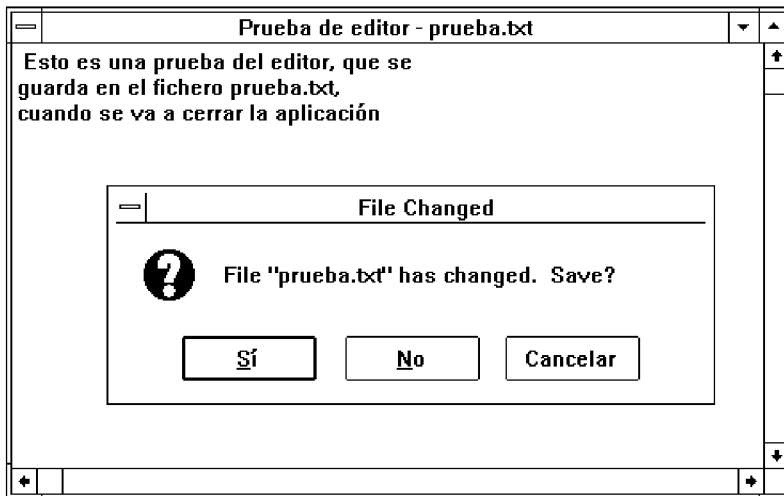


Figura 15.15 Ejecución del ejemplo 15.11

Una versión más completa de la construcción de un editor se presenta en el ejemplo 15.18.

### Ejemplo 15.12

Modificar el ejemplo 15.11 para que el mensaje de cierre de la aplicación salga en castellano.

**Solución.** Se redefine la función *CanClose* del tipo objeto *TFileWindow*, utilizando la función *Save* del mismo tipo objeto para guardar el fichero editado. Se utilizan las funciones API de Windows *MessageBox* y *MessageBeep*. La función *MessageBeep* emite un sonido de nivel 0, si el entorno Windows tiene instalado algún dispositivo de sonido. La constante *mb\_YesNoCancel* representa a los botones **Sí**, **No**, y **Cancelar**. La constante *mb\_IconQuestion* representa al icono interrogación que sale en cuadro de mensaje. La ejecución se muestra en la figura 15.16, y el código del programa se presenta a continuación:

```
PROGRAM PruebaEditor2;
USES OWindows, OStdWnds, WinProcs, WinTypes;
```

## LA BIBLIOTECA OBJECTWINDOWS®

```
TYPE
 TMiAplicacion=OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
END;
PMiFichero=^TMiFichero;
TMiFichero=OBJECT(TFileWindow)
 FUNCTION CanClose: Boolean; VIRTUAL;
END;

PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
 MainWindow:=New(PMiFichero, Init(NIL,'Prueba de editor', 'prueba.txt'));
END;
FUNCTION TMiFichero.CanClose;
VAR
 respuesta:integer;
BEGIN
 MessageBeep(0); (* Emite un sonido *)
 respuesta:=MessageBox(HWindow,
 '¿Desea guardar el fichero?',
 'Finaliza la aplicación',
 mb_YesNoCancel OR mb_IconQuestion);

 CASE respuesta OF
 id_Yes:BEGIN
 save;
 CanClose:=true;
 END;
 id_Cancel:CanClose:=false;
 id_No:CanClose:=true;
 END;
END;
VAR
 MiAplica:TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

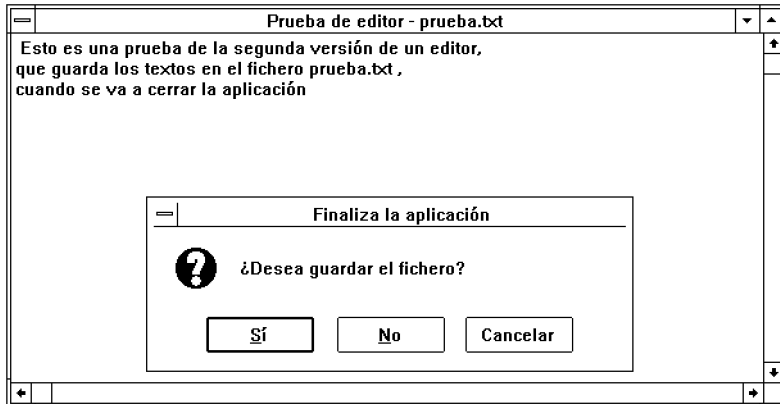


Figura 15.16 Ejecución del ejemplo 15.12

### • El tipo objeto TMDIWindow

El tipo objeto *TMDIWindow* permite manejar varias ventanas por una única aplicación. MDI son las siglas en inglés de *Multiple Document Interface*, que es una especificación para aplicaciones Windows que manejan varias ventanas simultáneamente. Como ejemplos de aplicaciones Windows que utilizan el MDI se pueden citar el *Administrador de programas de Windows*, el *Administrador de archivos de Windows*, y el entorno integrado de desarrollo (IDE) de Borland Pascal para Windows. El estándar MDI está definido dentro de las especificaciones IBM SAA/CUA.

La ventana principal (*main*) de una aplicación MDI es igual a cualquier otra aplicación windows, dicha ventana se denomina ventana marco (*frame window*) El área interior a la ventana se denomina área de trabajo (*workspace*) y puede contener ventanas hijas (*child windows*), o ventanas cliente (*client windows*). Las ventanas hijas o clientes son como la ventana marco (*frame*) con una excepción: no tienen menús. El menú principal de la ventana marco (*frame*) manipula las ventanas hijas. Sólo una ventana hija puede ser seleccionada de cada vez. Las ventanas hijas también pueden ser maximizadas y minimizadas. Una ventana hija minimizada es un icono dentro del área de trabajo.

El tipo objeto *TMDIWindow* permite la creación de ventanas marco (*frame*) y ventanas hijas o clientes. Las ventanas clientes se manejan con el tipo objeto *TMDIClient*. Este tipo objeto se encuentra en la *unit OWindows*.



La declaración del tipo objeto *TMDIWindow* es la siguiente:

```

PMDIWindow = ^TMDIWindow;
TMDIWindow = object(TWindow)
 ClientWnd: PMDIClient;
 ChildMenuPos: Integer;
 CONSTRUCTOR Init(ATitle: PChar; AMenu: HMenu);
 DESTRUCTOR Done; VIRTUAL;
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE InitClientWindow; VIRTUAL;
 FUNCTION GetClassName: PChar; VIRTUAL;
 FUNCTION GetClient: PMDIClient; VIRTUAL;
 PROCEDURE GetWindowClass(var AWndClass: TWndClass); VIRTUAL;
 PROCEDURE DefWndProc(var Msg: TMessage); VIRTUAL;
 FUNCTION InitChild: PWindowsObject; VIRTUAL;
 FUNCTION CreateChild: PWindowsObject; VIRTUAL;
 PROCEDURE CMCreateChild(var Msg: TMessage);
 VIRTUAL cm_First + cm_CreateChild;
 PROCEDURE TileChildren; VIRTUAL;
 PROCEDURE CascadeChildren; VIRTUAL;
 PROCEDURE ArrangeIcons; VIRTUAL;
 PROCEDURE CloseChildren; VIRTUAL;
 PROCEDURE CMTileChildren(var Msg: TMessage);
 VIRTUAL cm_First + cm_TileChildren;
 PROCEDURE CMCascadeChildren(var Msg: TMessage);
 VIRTUAL cm_First + cm_CascadeChildren;
 PROCEDURE CMArrangeIcons(var Msg: TMessage);
 VIRTUAL cm_First + cm_ArrangeIcons;
 PROCEDURE CMCloseChildren(var Msg: TMessage);
 VIRTUAL cm_First + cm_CloseChildren;
END;

```

Los métodos del tipo objeto *TMDIWindow* se ocupan de la gestión de las ventanas hijas o clientes, con ellos se pueden crear, cerrar, y mostrar en cascada o en forma de mosaico.

### Ejemplo 15.13

Escribir un programa sencillo que utilice ventanas MDI

**Solución.** El código del ejemplo se presenta a continuación, y una ejecución en la figura 15.17.

```

PROGRAM PruebaMDI;
{$R c15e13.RES}
USES WinTypes, WinProcs, Strings, OWindows, ODialogs;
CONST
 cm_cuentaHijas = 102;
 id_PuedeCerrarse = 201;
TYPE
 TmiAplicacionMDI = OBJECT(TApplication)
 PROCEDURE InitMainWindow; virtual;
 END;

 PMiHijaMDI = ^TmiHijaMDI;
 TmiHijaMDI = OBJECT(TWindow)
 Num: Integer;
 PuedeCerrarseCheckBox: PCheckBox;
 CONSTRUCTOR Init(UnPadre: PWindowsOBJECT; NumHija: Integer);
 PROCEDURE SetupWindow; virtual;
 FUNCTION CanClose: Boolean; virtual;
 END;

```

PROGRAMACION EN ENTORNO WINDOWS®

```

PMiVentanaMDI = ^TmiVentanaMDI;
TmiVentanaMDI = OBJECT(TMDIWindow)
 PROCEDURE SetupWindow; virtual;
 FUNCTION CreateChild: PWindowsOBJECT; virtual;
 FUNCTION CuentaHijas: Integer;
 PROCEDURE CMcuentaHijas(var Msg: TMessage);
 VIRTUAL cm_First + cm_cuentaHijas;
END;

{ El CONSTRUCTOR de TmiHijaMDI instancia una checkbox }
CONSTRUCTOR TmiHijaMDI.Init(UnPadre: PWindowsOBJECT; NumHija: Integer);
VAR
 tituloStr: array[0..12] of Char;
 NumHijaStr: array[0..5] of Char;
BEGIN
 Str(NumHija, NumHijaStr);
 StrCat(StrECopy(tituloStr, 'Hija n°'), NumHijaStr);
 INHERITED Init(UnPadre, tituloStr);
 Num := NumHija;
 New(PuedeCerrarseCheckBox, Init(@Self, id_PuedeCerrarse, 'Puede cerrarse', 10,
10, 200, 20, nil));
END;

PROCEDURE TmiHijaMDI.SetupWindow;
BEGIN
 INHERITED SetupWindow;
 PuedeCerrarseCheckBox^.Check;
END;

FUNCTION TmiHijaMDI.CanClose;
BEGIN
 CanClose := PuedeCerrarseCheckBox^.GetCheck = bf_Checked;
END;

{ SetupWindow crea la primera ventana MDI hija }
PROCEDURE TmiVentanaMDI.SetupWindow;
BEGIN
 INHERITED SetupWindow;
 CreateChild;
END;

{ Crea una nueva ventana MDI hija }
FUNCTION TmiVentanaMDI.CreateChild: PWindowsOBJECT;
VAR
 NumHija: Integer;
 FUNCTION NumberUsed(P: PMiHijaMDI): Boolean; far;
 BEGIN
 NumberUsed := NumHija = P^.Num;
 END;
BEGIN
 NumHija := 1;
 while FirstThat(@NumberUsed) <> nil do Inc(NumHija);
 CreateChild := Application^.MakeWindow(New(PMiHijaMDI,
 Init(@Self, NumHija)));
END;

{ Devuelve un contador de las ventanas hijas MDI }
FUNCTION TmiVentanaMDI.CuentaHijas: Integer;
VAR
 contador: Integer;
 PROCEDURE ContadorHijas(AChild: PWindowsOBJECT); far;
 BEGIN
 Inc(contador);
 END;

```

LA BIBLIOTECA OBJECTWINDOWS®

```

BEGIN
 contador := 0;
 ForEach(@ContadorHijas);
 CuentaHijas := contador;
END;

{ Muestra un mensaje con el número de ventanas hijas }
PROCEDURE TMiVentanaMDI.CMuentaHijas(var Msg: TMessage);
VAR
 ContadorStr: array[0..5] of Char;
BEGIN
 Str(CuentaHijas, ContadorStr);
 MessageBox(HWindow, ContadorStr, 'Total de hijas', mb_Ok);
END;

{ Construye un objeto ventana principal (Main) }
PROCEDURE TMiAplicacionMDI.InitMainWindow;
BEGIN
 MainWindow := New(PMiVentanaMDI,
 Init('Prueba de MDI', LoadMenu(HInstance, MakeIntResource(100))));
 HAccTable := LoadAccelerators(HInstance, MakeIntResource(100));
END;

VAR
 MiAplicaMDI: TMiAplicacionMDI;
BEGIN
 MiAplicaMDI.Init('Prueba MDI');
 MiAplicaMDI.Run;
 MiAplicaMDI.Done;
END.

```

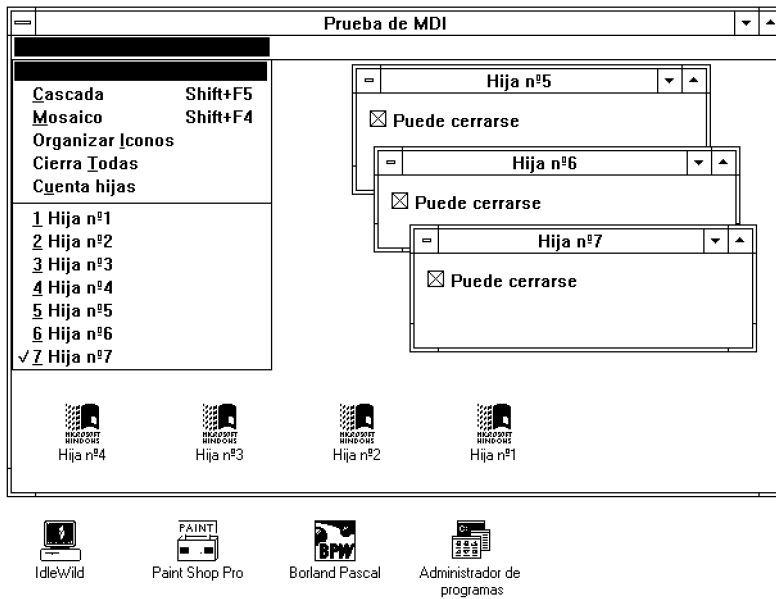
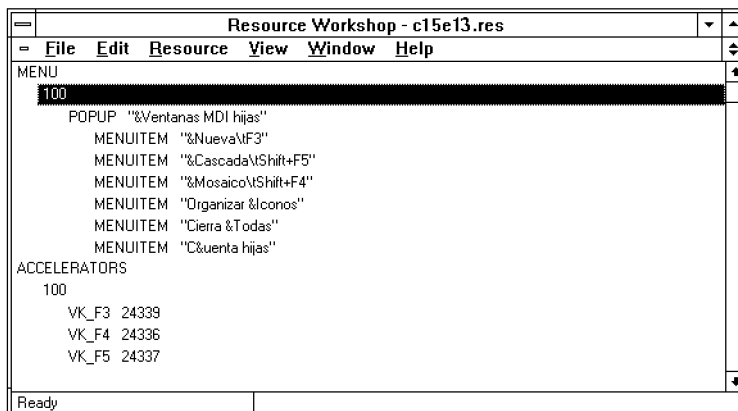


Figura 15.17 Ejecución del ejemplo 15.13

La instrucción `{$R c15e13.RES}` incluye el fichero binario de recursos *c15e13.res* creado con el taller de recursos (*resource workshop*), que es una herramienta que acompaña al *Borland Pascal*, para la creación de recursos<sup>78</sup>. Los recursos son las partes de un programa Windows, que interactúan con los usuarios. Ejemplos de recursos son los menús, las teclas aceleradoras, los mapas de bits, los iconos, los cursores, los cuadros de diálogo, etc... El objetivo de este fichero es, en este caso particular, permitir el manejo de un menú desplegable y unas teclas aceleradoras en dicho menú (véase figura 15.18). Las teclas aceleradoras usadas son las teclas de función **F3**, **F4** y **F5**, que se representan en el taller de recursos por las constantes *VK\_F3*, *VK\_F4*, y *VK\_F5*. Las funciones *LoadMenu* y *LoadAccelerators* cargan los recursos, y se explican dentro del subapartado *Programando con recursos* del epígrafe 15.7 de este capítulo. Los valores de las constantes *cm\_cuentaHijas* y *id\_PuedeCerrarse*<sup>79</sup> construidos por el programador, los valores que se han definido teniendo en cuenta la tabla 15.8.



IdleWild



Administrador de programas

Figura 15.18 Taller de recursos del ejemplo 15.13

#### • El tipo objeto *TMDIClient*

El tipo objeto *TMDIClient* es un descendiente del tipo *TWindow*, que maneja las ventanas cliente de una aplicación MDI (*Multiple Document Interface*). Los métodos de *TMDIClient*

<sup>78</sup> Los recursos se estudian en el apartado 15.7 de este capítulo

<sup>79</sup> Los mensajes se explican en el apartado 15.9 de este capítulo

## LA BIBLIOTECA OBJECTWINDOWS®

contribuyen al manejo de las ventanas clientes: creándolas, mostrándolas en cascada o en mosaico, pintando o escribiendo en ellas, o bien activándolas. Este tipo se encuentra en la *unit OWindows*.

La declaración del tipo objeto *TMDIClient* es la siguiente:

```
TMDIClient = OBJECT(TWindow)
 ClientAttr: TClientCreateStruct;
 CONSTRUCTOR Init(AParent: PMDIWindow);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
 FUNCTION GetClassName: PChar; VIRTUAL;
 FUNCTION Register: Boolean; VIRTUAL;
 PROCEDURE TileChildren; VIRTUAL;
 PROCEDURE CascadeChildren; VIRTUAL;
 PROCEDURE ArrangeIcons; VIRTUAL;
 PROCEDURE WMPaint(var Msg: TMessage);
 VIRTUAL wm_First + wm_Paint;
END;
```

### • El tipo objeto TControl

El tipo objeto *TControl* es un descendiente del tipo objeto *TWindows*, y permite la construcción de controles visuales, tales como cajas combinadas (*combo box*), botones de control, cajas de listas, cuadros de comprobación (*check boxes*) y botones de radio.

El tipo *TControl* está en la *unit ODialogs*, y su declaración es la siguiente:

```
PControl = ^TControl;
TControl = OBJECT(TWindow)
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnId: Integer;
 ATitle: PChar; X, Y, W, H: Integer);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word);
 FUNCTION Register: Boolean; virtual;
 FUNCTION GetClassName: PChar; virtual;
 PROCEDURE WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
END;
```

### • El tipo objeto TGroupBox

*TGroupBox* es un tipo objeto interfaz que representa a los elementos correspondientes de Windows denominados cajas de grupo. Las cajas de grupo no tienen un papel activo, tan sólo unifican un grupo de cajas de selección (cuadros de comprobación y botones de radio). Sin embargo pueden coordinar los estados de las distintas cajas de selección. Por ejemplo, se puede activar un cuadro de comprobación dentro de una caja de grupo, si se desactivan el resto de los cuadros de comprobación.

El tipo *TGroupBox* está en la *unit ODialogs*, y su declaración es la siguiente:

```
PGroupBox = ^TGroupBox;
TGroupBox = OBJECT(TControl)
 NotifyParent: Boolean;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnID: Integer;
 AText: PChar; X, Y, W, H: Integer);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
```

```

FUNCTION GetClassName: PChar; virtual;
PROCEDURE SelectionChanged(ControlId: Integer); virtual;
END;

```

### • El tipo objeto TButton

*TButton* es un tipo objeto interfaz que representa al elemento correspondiente de Windows botón de pulsar.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```

PButton = ^TButton;
TButton = OBJECT(TControl)
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnId: Integer;
 AText: PChar; X, Y, W, H: Integer; IsDefault: Boolean);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word);
 FUNCTION GetClassName: PChar; virtual;
END;

```

### Ejemplo 15.14

Escribir un programa que presente en pantalla dos botones y envíe un mensaje diferente según se pulse un botón o otro.

#### Solución.

```

PROGRAM EjemploDeManejoDeBotones;
USES WinTypes, WinProcs, OWindows, ODialogs;
CONST
 id_Boton1 = 101;
 id_Boton2 = 102;
TYPE
 TMiAplicacion = OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;

 PMiVentana = ^TMiVentana;
 TMiVentana = OBJECT(TWindow)
 Boton1: PButton;
 Boton2: PButton;
 CONSTRUCTOR Init(UnPadre: PWindowsObject; Untitulo: PChar);
 PROCEDURE ManejaMsgBoton1(var Msg: TMessage);
 VIRTUAL id_First + id_Boton1;
 PROCEDURE ManejaMsgBoton2(var Msg: TMessage);
 VIRTUAL id_First + id_Boton2;
 END;

CONSTRUCTOR TMiVentana.Init;
BEGIN
 INHERITED Init(UnPadre, Untitulo);
 Boton1 := New(PButton, Init(@Self, id_Boton1, 'Pulsar botón 1',
 20, 70, 150, 90, true));
 Boton2 := New(PButton, Init(@Self, id_Boton2, 'Pulsar botón 2',
 220, 70, 360, 90, true));
END;

PROCEDURE TMiVentana.ManejaMsgBoton1;
BEGIN
 MessageBeep(0);
 MessageBox(HWindow, 'Se ha pulsado el botón 1', 'Mensaje', mb_OK);
END;

```

## LA BIBLIOTECA OBJECTWINDOWS®

```
PROCEDURE TMiVentana.ManejaMsgBoton2;
BEGIN
 MessageBeep(1);
 MessageBox(HWindow, 'Se ha pulsado el botón 2', 'Mensaje', mb_OK);
END;

PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
 MainWindow := New(PMiVentana, Init(nil, 'Ejemplo de botones'));
END;
VAR
 MiAplica : TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.
```

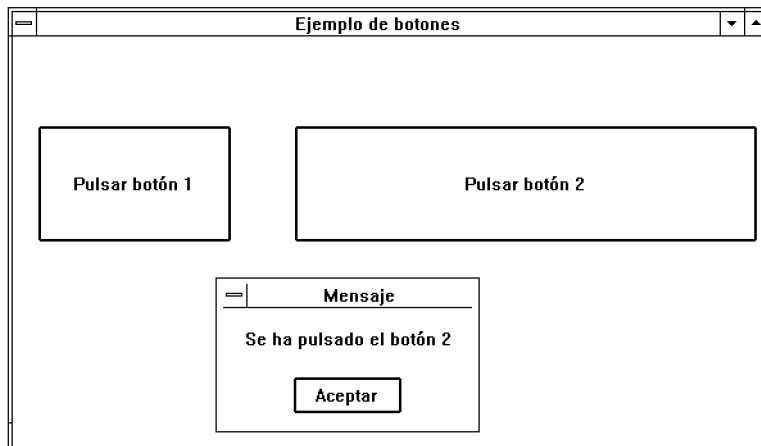


Figura 15.19 Ejecución del ejemplo 15.14

### • El tipo objeto TCheckBox

*TCheckBox* es un tipo objeto que representa a las cajas de comprobación del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```
PCheckBox = ^TCheckBox;
TCheckBox = OBJECT(TButton)
 Group: PGroupBox;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnID: Integer;
 ATitle: PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word);
 CONSTRUCTOR Load(var S: TStream);
```

```

PROCEDURE Store(var S: TStream);
PROCEDURE Check;
PROCEDURE Uncheck;
PROCEDURE Toggle;
FUNCTION GetClassName: PChar; virtual;
FUNCTION GetCheck: Word;
PROCEDURE SetCheck(CheckFlag: Word);
FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
PROCEDURE BNClicked(var Msg: TMessage);
 VIRTUAL nf_First + bn_Clicked;
END;

```

### • El tipo objeto TRadioButton

Este tipo objeto representa a los botones de radio del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```

PRadioButton = ^TRadioButton;
TRadioButton = OBJECT(TCheckBox)
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnID: Integer;
 ATitle: PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
 FUNCTION GetClassName: PChar; virtual;
END;

```

### • El tipo objeto TStatic

Este tipo objeto representa a los textos estáticos del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```

PStatic = ^TStatic;
TStatic = OBJECT(TControl)
 TextLen: Word;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnID: Integer;
 ATitle: PChar; X, Y, W, H: Integer; ATextLen: Word);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word;
 ATextLen: Word);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
 FUNCTION GetClassName: PChar; virtual;
 FUNCTION GetText(ATextString: PChar; MaxChars: Integer): Integer;
 FUNCTION GetTextLen: Integer;
 PROCEDURE SetText(ATextString: PChar);
 PROCEDURE Clear;
 FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
END;

```

El ejemplo 15.16 maneja el tipo *TStatic* para mostrar un texto estático dentro de una ventana.

### • El tipo objeto TEdit

Este tipo objeto representa a los controles de edición del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:



LA BIBLIOTECA OBJECTWINDOWS®

```

PEdit = ^TEdit;
TEdit = OBJECT(TStatic)
 Validator: PValidator;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnId: Integer; ATitle: PChar;
 X, Y, W, H: Integer; ATextLen: Word; Multiline: Boolean);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word;
 ATextLen: Word);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; virtual;
 FUNCTION GetClassName: PChar; virtual;
 PROCEDURE Undo;
 FUNCTION CanClose: Boolean; virtual;
 FUNCTION CanUndo: Boolean;
 PROCEDURE Paste;
 PROCEDURE Copy;
 PROCEDURE Cut;
 FUNCTION GetNumLines: Integer;
 FUNCTION GetLineLength(LineNumber: Integer): Integer;
 FUNCTION GetLine(ATextString: PChar;
 StrSize, LineNumber: Integer): Boolean;
 PROCEDURE GetSubText(ATextString: PChar; StartPos, ENDPos: Integer);
 FUNCTION DeleteSubText(StartPos, ENDPos: Integer): Boolean;
 FUNCTION DeleteLine(LineNumber: Integer): Boolean;
 PROCEDURE GetSelection(var StartPos, ENDPos: Integer);
 FUNCTION DeleteSelection: Boolean;
 FUNCTION IsModified: Boolean;
 PROCEDURE ClearModify;
 FUNCTION GetLineFromPos(CharPos: Integer): Integer;
 FUNCTION GetLineIndex(LineNumber: Integer): Integer;
 FUNCTION IsValid(ReportError: Boolean): Boolean;
 PROCEDURE Scroll(HorizontalUnit, VerticalUnit: Integer);
 FUNCTION SetSelection(StartPos, ENDPos: Integer): Boolean;
 PROCEDURE Insert(ATextString: PChar);
 FUNCTION Search(StartPos: Integer; AText: PChar; CaseSensitive: Boolean):
Integer;
 PROCEDURE SetupWindow; virtual;
 PROCEDURE SetValidator(AValid: PValidator);
 PROCEDURE Store(var S: TStream);
 FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
 PROCEDURE CMEditCut(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditCut;
 PROCEDURE CMEditCopy(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditCopy;
 PROCEDURE CMEditPaste(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditPaste;
 PROCEDURE CMEditDelete(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditDelete;
 PROCEDURE CMEditClear(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditClear;
 PROCEDURE CMEditUndo(var Msg: TMessage);
 VIRTUAL cm_First + cm_EditUndo;
 PROCEDURE WMChar(var Msg: TMessage);
 VIRTUAL wm_First + wm_Char;
 PROCEDURE WMKeyDown(var Msg: TMessage);
 VIRTUAL wm_First + wm_KeyDown;
 PROCEDURE WMGetDlgCode(var Msg: TMessage);
 VIRTUAL wm_First + wm_GetDlgCode;
 PROCEDURE WMKillFocus(var Msg: TMessage);
 VIRTUAL wm_First + wm_KillFocus;
END;

```

El campo *validator* es un puntero al tipo objeto *TValidator* de la jerarquía *ObjectWindows*.

### • El tipo objeto TListBox

Este tipo objeto representa a las cajas de listas del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```
PListBox = ^TListBox;
TListBox = OBJECT(TControl)
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnId: Integer;
 X, Y, W, H: Integer);
 FUNCTION GetClassName: PChar; VIRTUAL;
 FUNCTION AddString(AStrng: PChar): Integer;
 FUNCTION InsertString(AStrng: PChar; Index: Integer): Integer;
 FUNCTION DeleteString(Index: Integer): Integer;
 PROCEDURE ClearList;
 FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; VIRTUAL;
 FUNCTION GetCount: Integer;
 FUNCTION GetString(AStrng: PChar; Index: Integer): Integer;
 FUNCTION GetStringLen(Index: Integer): Integer;
 FUNCTION GetSelString(AStrng: PChar; MaxChars: Integer): Integer;
 FUNCTION SetSelString(AStrng: PChar; Index: Integer): Integer;
 FUNCTION GetSelIndex: Integer;
 FUNCTION SetSelIndex(Index: Integer): Integer;
PRIVATE
 FUNCTION GetMsgID(AMsg: TMsgName): Word; VIRTUAL;
END;
```

El tipo *TMsgName* usado por el método *GetMsgID* es el tipo enumerado que se define a continuación:

```
TMsgName = (
 mn_AddString, mn_InsertString, mn_DeleteString,
 mn_ResetContent, mn_GetCount, mn_GetText,
 mn_GetTextLen, mn_SelectString, mn_SetCurSel,
 mn_GetCurSel);
```

### Ejemplo 15.15

Construir un programa que muestre una ventana con una caja con una lista, y envíe un mensaje cuando se selecciona un elemento de la lista.

**Solución.** Se declara un nuevo tipo objeto denominado *TMiVentana* derivado de *TWindow*, con un campo de tipo *PListBox*. Se redefine el método *Init*, para que cuando se inicie la ventana se cree la caja con lista. Se redefine también el método *SetupWindow* para que aparezcan los valores de la lista; se usan los métodos *AddString* y *InsertString* de *TListBox* para introducir los valores, que quedarán ordenados. Por último se construye el método dinámico *ManejaMsgCajaLista* para enviar un mensaje cada vez que se selecciona un elemento de la lista.

El código del programa se muestra a continuación y una ejecución se puede observar en la figura 15.15.

```
PROGRAM EjemploDeCajaDeLista;
USES WinTypes, WinProcs, OWindows, ODialogs;
CONST
 id_CajaLista1 = 101;
TYPE
 TMiAplicacion = OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;
```

LA BIBLIOTECA OBJECTWINDOWS®

```

PMiVentana = ^TMiVentana;
TMiVentana = OBJECT(TWindow)
 CajaListal: PListBox;
 CONSTRUCTOR Init(UnPadre: PWindowsObject; Untitulo: PChar);
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE ManejaMsgCajaLista(var Msg: TMessage);
 VIRTUAL id_First + id_CajaListal;
END;

CONSTRUCTOR TMiVentana.Init;
BEGIN
 INHERITED Init(UnPadre, Untitulo);
 CajaListal := New(PListBox, Init(@Self, id_CajaListal, 20, 20, 340, 100));
END;

PROCEDURE TMiVentana.SetupWindow;
BEGIN
 INHERITED SetupWindow;
 CajaListal^.AddString('Elemento 1: color');
 CajaListal^.AddString('Elemento 2: sonido');
 CajaListal^.AddString('Elemento 3: imagen');
 CajaListal^.InsertString('Elemento 1.5: bis', 1);
 CajaListal^.AddString('Elemento 4: reservado');
 CajaListal^.AddString('Elemento 5: reservado');
 CajaListal^.InsertString('Elemento 5.bis: reservado',6);
 CajaListal^.AddString('Elemento 6: reservado');
END;

PROCEDURE TMiVentana.ManejaMsgCajaLista(var Msg: TMessage);
CONST
 maxCadena=40;
VAR
 indice: Integer;
 TextoElemento: ARRAY[0..maxCadena] OF Char;
BEGIN
 IF Msg.LParamHi = lbn_SelChange THEN
 BEGIN
 indice := CajaListal^.GetSelIndex;
 IF CajaListal^.GetStringLen(indice) < (maxCadena-1) THEN
 BEGIN
 CajaListal^.GetSelString(TextoElemento, maxCadena);
 MessageBox(HWindow, TextoElemento, 'Ha seleccionado', mb_OK);
 END;
 END
 ELSE DefWndProc(Msg);
END;

PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
 MainWindow := New(PMiVentana, Init(nil, 'Ejemplo de cajas de listas'));
END;

VAR
 MiAplica : TMiAplicacion;
BEGIN
 MiAplica.Init('Prueba');
 MiAplica.Run;
 MiAplica.Done;
END.

```

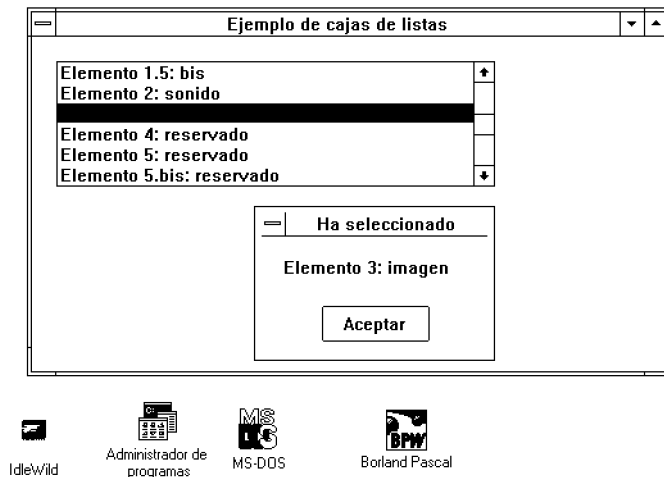


Figura 15.20 Ejecución del ejemplo 15.15

Si se deseara introducir otra caja de lista en la ventana, se declararía otro campo *CajaLista2* y otra constante *id\_cajaLista2* con otro valor para identificar a los mensajes de la 2ª caja de listas. Habría que añadir una segunda creación caja de lista en *Init*; ampliar *SetupWindow* para meter los valores de la 2ª lista; y construir un nuevo método para manejar los mensajes de la segunda lista.

### • El tipo objeto TComboBox

Este tipo objeto representa a las cajas combinadas del entorno Windows. Este tipo de controles combinan una caja de textos para entradas (control de edición) y una caja de listas asociada.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```

PComboBox = ^TComboBox;
TComboBox = OBJECT(TListBox)
 TextLen: Word;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnID: Integer;
 X, Y, W, H: Integer; AStyle: Word; ATextLen: Word);
 CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Integer;
 ATextLen: Word);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Store(var S: TStream);
 FUNCTION GetClassName: PChar; virtual;
 PROCEDURE ShowList;
 PROCEDURE HideList;
 FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
 PROCEDURE SetupWindow; virtual;
 FUNCTION GetTextLen: Integer;
 FUNCTION GetText(Str: PChar; MaxChars: Integer): Integer;
 PROCEDURE SetText(Str: PChar);

```

## LA BIBLIOTECA OBJECTWINDOWS®

```
FUNCTION SetEditSel(StartPos, ENDPos: Integer): Integer;
FUNCTION GetEditSel(var StartPos, ENDPos: Integer): Boolean;
PROCEDURE Clear;
PRIVATE
FUNCTION GetMsgID(AMsg: TMsgName): Word; virtual;
END;
```

### • El tipo objeto TScrollBar

Este tipo objeto representa a las barras de desplazamiento horizontales o verticales del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```
PScrollBar = ^TScrollBar;
TScrollBar = OBJECT(TControl)
LineMagnitude, PageMagnitude: Integer;
CONSTRUCTOR Init(AParent: PWindowsOBJECT; AnID: Integer;
X, Y, W, H: Integer; IsHScrollBar: Boolean);
CONSTRUCTOR InitResource(AParent: PWindowsOBJECT; ResourceID: Word);
CONSTRUCTOR Load(var S: TStream);
PROCEDURE Store(var S: TStream);
FUNCTION GetClassName: PChar; virtual;
PROCEDURE SetupWindow; virtual;
PROCEDURE GetRange(var LoVal, HiVal: Integer);
FUNCTION GetPosition: Integer;
PROCEDURE SetRange(LoVal, HiVal: Integer);
PROCEDURE SetPosition(ThumbPos: Integer);
FUNCTION DeltaPos(Delta: Integer): Integer;
FUNCTION Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
PROCEDURE SBLineUp(var Msg: TMessage);
VIRTUAL nf_First + sb_LineUp;
PROCEDURE SBLineDown(var Msg: TMessage);
VIRTUAL nf_First + sb_LineDown;
PROCEDURE SBPageUp(var Msg: TMessage);
VIRTUAL nf_First + sb_PageUp;
PROCEDURE SBPageDown(var Msg: TMessage);
VIRTUAL nf_First + sb_PageDown;
PROCEDURE SBThumbPosition(var Msg: TMessage);
VIRTUAL nf_First + sb_ThumbPosition;
PROCEDURE SBThumbTrack(var Msg: TMessage);
VIRTUAL nf_First + sb_ThumbTrack;
PROCEDURE SBTop(var Msg: TMessage);
VIRTUAL nf_First + sb_Top;
PROCEDURE SBBottom(var Msg: TMessage);
VIRTUAL nf_First + sb_Bottom;
END;
```

### Ejemplo 15.16

Escribir un programa que presente el control de un termostato en forma de barra de desplazamiento horizontal, además tendrá otro control en forma de texto estático que muestre en cada momento la temperatura marcada en la barra de desplazamiento.

**Solución.** Este programa es muy parecido al presentado en el ejemplo 15.15. Se construye un nuevo tipo objeto denominado *TMiVentana* derivado de *TWindow*, con dos campos de tipo

*PScrollBar* y *PStatic*. El método *Init* se encarga de iniciar estos dos campos, y el método *SetupWindow* configura el rango de la barra desplazable. El método *ManejaMsgTermoBarraD* se encarga de manejar los mensajes enviados por el movimiento de la barra desplazable, para que aparezcan como texto estático en la posición fija de la pantalla.

```
PROGRAM PruebaBarraDesplazamientoYtextoEstatico;
USES WinTypes, WinProcs, Strings, OWindows, ODialogs;
CONST
 id_barraDesplazamiento = 100;
 id_textoEstatico = 101;
TYPE
 TMiAplicacion = OBJECT(TApplication)
 PROCEDURE InitMainWindow; VIRTUAL;
 END;

 PMiVentana = ^TMiVentana;
 TMiVentana = OBJECT(TWindow)
 TermoBarraDesplazable : PScrollBar;
 textoEstatico : PStatic;
 CONSTRUCTOR Init(UnPadre: PWindowsObject; UnTitulo: PChar);
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE ManejaMsgTermoBarraD(var Msg: TMessage);
 VIRTUAL id_First + id_barraDesplazamiento;
 END;

 { Establece atributos de inicialización de la ventana y
 construye los controles hijos }
 CONSTRUCTOR TMiVentana.Init(UnPadre: PWindowsObject; UnTitulo: PChar);
 BEGIN
 INHERITED Init(UnPadre, UnTitulo);
 WITH Attr DO
 BEGIN
 X := 20;
 Y := 20;
 W := 380;
 H := 250; (* Atributos iniciales de la ventana *)
 END;
 TermoBarraDesplazable := New(PScrollBar, Init(@Self,
 id_barraDesplazamiento,
 20, 170, 340, 0, True));
 textoEstatico := New(PStatic, Init(@Self, id_textoEstatico,
 ' 10 grados', 135, 40, 160, 17, 0));
 END;

 { Establece el rango de la barra desplazable }
 PROCEDURE TMiVentana.SetupWindow;
 BEGIN
 INHERITED SetupWindow;
 TermoBarraDesplazable^.SetRange(-40, 60);
 END;

 PROCEDURE TMiVentana.ManejaMsgTermoBarraD(var Msg: TMessage);
 VAR
 cString: array[0..11] of Char;
 BEGIN
 Str(TermoBarraDesplazable^.GetPosition:3, cString);
 StrCat(cString, ' grados');
 textoEstatico^.SetText(cString);
 END;

 PROCEDURE TMiAplicacion.InitMainWindow;
 BEGIN
 MainWindow := New(PMiVentana, Init(nil, 'Termoestato'));
 END;
```

## LA BIBLIOTECA OBJECTWINDOWS®

```
VAR
 TestApp : TMiAplicacion;

BEGIN
 TestApp.Init('Prueba');
 TestApp.Run;
 TestApp.Done;
END.
```

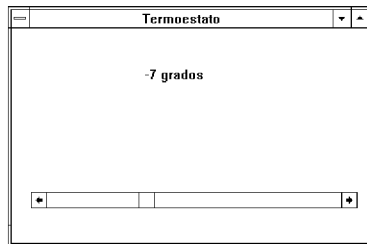


Figura 15.21 Ejecución del ejemplo 15.16

### • El tipo objeto TDialog

El tipo objeto *TDialog* es un descendiente del tipo objeto *TWindowsOBJECT*, y permite una mejora de la interfaz de usuario de su antecesor por medio del uso de cajas de diálogos. Se puede utilizar directamente, o a través de sus descendientes, que son cajas de diálogo especializadas (*TFileDialog*, *TInputDialog*, *TPrintDialog*,...).

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```
PDialog = ^TDialog;
TDialog = OBJECT(TWindowsOBJECT)
 Attr: TDialogAttr;
 IsModal: Boolean;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AName: PChar);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Store(var S: TStream);
 FUNCTION Create: Boolean; VIRTUAL;
 FUNCTION Execute: Integer; VIRTUAL;
 PROCEDURE ENDDlg(ARetVal: Integer); VIRTUAL;
 FUNCTION GetItemHandle(DlgItemID: Integer): HWnd;
 FUNCTION SENDDlgItemMsg(DlgItemID: Integer; AMsg, WParam: Word;
 LParam: LongInt): LongInt;
 PROCEDURE Ok(var Msg: TMessage); VIRTUAL id_First + id_Ok;
 PROCEDURE Cancel(var Msg: TMessage); VIRTUAL id_First + id_Cancel;
 PROCEDURE WMInitDialog(var Msg: TMessage);
 VIRTUAL wm_First + wm_InitDialog;
 PROCEDURE WMQueryENDSession(var Msg: TMessage);
 VIRTUAL wm_First + wm_QueryENDSession;
```

```

PROCEDURE WMClose(var Msg: TMessage);
 VIRTUAL wm_First + wm_Close;
PROCEDURE WMPostInvalid(var Msg: TMessage);
 VIRTUAL wm_First + wm_PostInvalid;
PROCEDURE DefWndProc(var Msg: TMessage); VIRTUAL;
END;

```

El tipo *TDialogAttr* es el registro siguiente:

```

TDialogAttr = RECORD
 Name: PChar;
 Param: LongInt;
END;

```

#### • El tipo objeto *TDlgWindow*

Este tipo objeto representa a una combinación de las cajas de diálogo y de las ventanas del entorno Windows.

Este tipo está en la *unit ODialogs*, y su declaración es la siguiente:

```

PDlgWindow = ^TDlgWindow;
TDlgWindow = OBJECT(TDialog)
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AName: PChar);
 PROCEDURE GetWindowClass(var AWndClass: TWndClass); VIRTUAL;
 FUNCTION Create: Boolean; VIRTUAL;
END;

```

#### • El tipo objeto *TPrinterSetupDlg*

El tipo objeto *TPrinterSetupDlg* es un tipo diálogo para modificar la impresora que está conectada actualmente. Este tipo muestra todas las impresoras activas en el sistema, permitiendo al usuario seleccionar la impresora deseada y configurarla. Este tipo objeto está en la *unit OPrinter*.

```

PPrinterSetupDlg = ^TPrinterSetupDlg;
TPrinterSetupDlg = OBJECT(TDialog)
 Printer: PPrinter;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; TemplateName: PChar;
 APrinter: PPrinter);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE TransferData(TransferFlag: Word); VIRTUAL;
 PROCEDURE IDSetup(var Msg: TMessage);
 VIRTUAL id_First + id_Setup;
 PROCEDURE Cancel(var Msg: TMessage);
 VIRTUAL id_First + id_Cancel;
PRIVATE
 OldDevice, OldDriver, OldPort: PChar;
 DeviceCollection: PCollection;
END;

```

#### • El tipo objeto *TPrinterAbortDlg*

Este tipo objeto representa a la caja de diálogo que se presenta cuando se ha comenzado a imprimir, y que permite la cancelación del proceso de impresión.

Este tipo objeto está en la *unit OPrinter*.



## LA BIBLIOTECA OBJECTWINDOWS®

```
PPrinterAbortDlg = ^TPrinterAbortDlg;
TPrinterAbortDlg = OBJECT(TDialog)
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; Template, Title,
 Device, Port: PChar);
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE WMCommand(var Msg: TMessage);
 VIRTUAL wm_First + wm_Command;
END;
```

### • El tipo objeto TPrintDialog

Este tipo objeto representa a la caja de diálogo que define los parámetros de un trabajo de impresión (número de páginas, número de copias, impresora a utilizar, etc...).

Este tipo objeto está en la *unit OPrinter*.

```
PPrintDialog = ^TPrintDialog;
TPrintDialog = OBJECT(TDialog)
 Printer: PPrinter;
 PData: PPrintDialogRec;
 PrinterName: PStatic;
 Pages: Integer;
 Controls: PCollection;
 AllBtn, SelectBtn, PageBtn: PRadioButton;
 FromPage, ToPage: PEdit;
 Copies: PEdit;
 Collate: PCheckBox;
 PrnDC: HDC;
 SelAllowed: Boolean;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; Template: PChar; APrnDC: HDC;
 APages: Integer; APrinter: PPrinter; ASelAllowed: Boolean;
 var Data: TPrintDialogRec);
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE TransferData(Direction: Word); VIRTUAL;
 PROCEDURE IDSetup(var Msg: TMessage);
 VIRTUAL id_First + id_Setup;
END;
```

### • El tipo objeto TInputDialog

Este tipo objeto representa a una caja de diálogo de uso general para la introducción de textos.

Este tipo está en la *unit OStdDlgs*, y su declaración es la siguiente:

```
PInputDialog = ^TInputDialog;
TInputDialog = OBJECT(TDialog)
 Caption: PChar;
 Prompt: PChar;
 Buffer: PChar;
 BufferSize: Word;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT;
 ACaption, APrompt, ABuffer: PChar; ABufferSize: Word);
 FUNCTION CanClose: Boolean; VIRTUAL;
 PROCEDURE SetupWindow; VIRTUAL;
END;
```

### • El tipo objeto TFileDialog

Este tipo objeto representa a la caja de diálogo que permite seleccionar entre distintos ficheros.

Este tipo está en la *unit OStdDlgs*, y su declaración es la siguiente:

```
PFileDialog = ^TFileDialog;
TFileDialog = OBJECT(TDialog)
 Caption: PChar;
 FilePath: PChar;
 PathName: array[0..fsPathName] of Char;
 Extension: array[0..fsExtension] of Char;
 FileSpec: array[0..fsFileSpec] of Char;
 CONSTRUCTOR Init(AParent: PWindowsOBJECT; AName, AFilePath: PChar);
 FUNCTION CanClose: Boolean; VIRTUAL;
 PROCEDURE SetupWindow; VIRTUAL;
 PROCEDURE HandleFName(var Msg: TMessage); VIRTUAL id_First + id_FName;
 PROCEDURE HandleFList(var Msg: TMessage); VIRTUAL id_First + id_FList;
 PROCEDURE HandleDList(var Msg: TMessage); VIRTUAL id_First + id_DList;
PRIVATE
 PROCEDURE SelectFileName;
 PROCEDURE UpdateFileName;
 FUNCTION UpdateListBoxes: Boolean;
END;
```

### • El tipo objeto TScroller

El tipo objeto *TScroller* da soporte al manejo de las barras de desplazamiento horizontal de las ventanas. El campo *Scroller* de *TWindow* contiene un puntero a la instancia de *TScroller*. Habitualmente las instancias a *TScroller* se realizan de forma "invisible" desde *TWindow*.

Este tipo se encuentra en la *unit OWindows*, y su declaración es la siguiente:

```
TScroller = OBJECT(TOBJECT)
 Window: PWindow;
 XPos: LongInt; (* Posición horizontal actual en unidades de desp. hor. *)
 YPos: LongInt; (* Posición vertical actual en unidades de desp. vert. *)
 XUnit: Integer; (* Unidades del dispositivo lógico por cada unidad hor. *)
 YUnit: Integer; (* Unidades de dispositivo lógico por cada unidad ver. *)
 XRange: LongInt; (* Rango de desplazamiento horizontal *)
 YRange: LongInt; (* Rango de desplazamiento vertical *)
 XLine: Integer; (* n° de unidades hor. de desplazamiento por línea *)
 YLine: Integer; (* n° de unidades vert. de desplazamiento por línea *)
 XPage: Integer; (* n° de unidades hor. de desplazamiento por página *)
 YPage: Integer; (* n° de unidades vert. de desplazamiento por página *)
 AutoMode: Boolean;
 TrackMode: Boolean;
 AutoOrg: Boolean;
 HasHScrollBar: Boolean;
 HasVScrollBar: Boolean;
 CONSTRUCTOR Init(TheWindow: PWindow; TheXUnit, TheYUnit: Integer;
 TheXRange, TheYRange: LongInt);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Store(var S: TStream);
 PROCEDURE SetUnits(TheXUnit, TheYUnit: LongInt);
 PROCEDURE SetPageSize; VIRTUAL;
 PROCEDURE SetSBarRange; VIRTUAL;
 PROCEDURE SetRange(TheXRange, TheYRange: LongInt);
 PROCEDURE BEGINView(PaintDC: HDC; var PaintInfo: TPaintStruct); VIRTUAL;
 PROCEDURE ENDView; VIRTUAL;
```

## LA BIBLIOTECA OBJECTWINDOWS®

```
PROCEDURE VScroll(ScrollRequest: Word; ThumbPos: Integer); VIRTUAL;
PROCEDURE HScroll(ScrollRequest: Word; ThumbPos: Integer); VIRTUAL;
PROCEDURE ScrollTo(X, Y: LongInt);
PROCEDURE ScrollBy(Dx, Dy: LongInt);
PROCEDURE AutoScroll; VIRTUAL;
FUNCTION IsVisibleRect(X, Y: LongInt; XExt, YExt: Integer): Boolean;
PRIVATE
FUNCTION XScrollValue(ARangeUnit: Longint): Integer;
FUNCTION YScrollValue(ARangeUnit: Longint): Integer;
FUNCTION XRangeValue(AScrollUnit: Integer): Longint;
FUNCTION YRangeValue(AScrollUnit: Integer): Longint;
END;
```

### • El tipo objeto TPrinter

El tipo objeto *TPrinter* representa al periférico donde se va a imprimir. Mientras que el tipo *TPrintOut* es el documento a imprimir. Para imprimir se envía un objeto de tipo *TPrintOut* con el método *Print* de *TPrinter*. El tipo objeto *TPrinter* se encuentra en la *unit OPrinter*, y su declaración es la siguiente:

```
PPrinter = ^TPrinter;
TPrinter = OBJECT(TOBJECT)
Device, Driver, Port: PChar; { Descripción del dispositivo de impresión }
Status: Integer; { Estatus del dispositivo, error si es <> ps_Ok }
Error: Integer; { Si ocurre un error durante la impresión, Error<0 }
DeviceModule: THandle;
DeviceMode: TDeviceMode;
ExtDeviceMode: TExtDeviceMode;
DevSettings: PDevMode;
DevSettingSize: Integer;
CONSTRUCTOR Init;
DESTRUCTOR Done; VIRTUAL;
PROCEDURE ClearDevice;
PROCEDURE Configure(Window: PWindowsOBJECT);
FUNCTION GetDC: HDC; VIRTUAL;
FUNCTION InitAbortDialog(Parent: PWindowsOBJECT;
 Title: PChar): PDialog; VIRTUAL;
FUNCTION InitPrintDialog(Parent: PWindowsOBJECT; PrnDC: HDC;
 Pages: Integer; SelAllowed: Boolean;
 VAR Data: TPrintDialogRec): PDialog; VIRTUAL;
FUNCTION InitSetupDialog(Parent: PWindowsOBJECT): PDialog; VIRTUAL;
PROCEDURE ReportError(PrintOut: PPrintOut); VIRTUAL;
PROCEDURE SetDevice(ADevice, ADriver, APort: PChar);
PROCEDURE Setup(Parent: PWindowsOBJECT);
FUNCTION Print(ParentWin: PWindowsOBJECT; PrintOut: PPrintOut): Boolean;
END;
```

El tipo *TPrintDialogRec* que aparece en el método *InitPrintDialog* se define a continuación:

```
PPrintDialogRec = ^TPrintDialogRec;
TPrintDialogRec = RECORD
 drStart: Integer; { Página de comienzo }
 drStop: Integer; { Página final }
 drCopies: Integer; { Número de copias a imprimir }
 drCollate: Boolean;
 drUseSelection: Boolean;
END;
```

### • El tipo objeto TPrintOut

El tipo objeto *TPrintOut* representa el documento físico impreso que se envía a la impresora. Este tipo está en la *unit Oprinter*.

```
PPrintOut = ^TPrintOut;
TPrintOut = OBJECT(TOBJECT)
 Title: PChar;
 Banding: Boolean;
 ForceAllBands: Boolean;
 DC: HDC;
 Size: TPoint;
 CONSTRUCTOR Init(ATitle: PChar);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE BeginDocument(StartPage, ENDPAGE: Integer;
 Flag: Word); VIRTUAL;
 PROCEDURE BeginPrinting; VIRTUAL;
 PROCEDURE EndDocument; VIRTUAL;
 PROCEDURE EndPrinting; VIRTUAL;
 FUNCTION GetDialogInfo(var Pages: Integer): Boolean; VIRTUAL;
 FUNCTION GetSelection(var Start, Stop: Integer): Boolean; VIRTUAL;
 FUNCTION HasNextPage(Page: Word): Boolean; VIRTUAL;
 PROCEDURE PrintPage(Page: Word; var Rect: TRect; Flags: Word); VIRTUAL;
 PROCEDURE SetPrintParams(ADC: HDC; ASize: TPoint); VIRTUAL;
END;
```

### • El tipo objeto TEditPrintout

Este tipo objeto está diseñado para imprimir los contenidos de un control de edición.

Este tipo está en la *unit Oprinter*.

```
PEditPrintout = ^TEditPrintout;
TEditPrintout = OBJECT(TPrintout)
 Editor: PEdit;
 NumLines: Integer;
 LinesPerPage: Integer;
 LineHeight: Integer;
 StartPos: Integer;
 StopPos: Integer;
 StartLine: Integer;
 StopLine: Integer;
 CONSTRUCTOR Init(AEditor: PEdit; ATitle: PChar);
 PROCEDURE BeginDocument(StartPage, ENDPAGE: Integer;
 Flags: Word); VIRTUAL;
 FUNCTION GetDialogInfo(var Pages: Integer): Boolean; VIRTUAL;
 FUNCTION GetSelection(var Start, Stop: Integer): Boolean; VIRTUAL;
 FUNCTION HasNextPage(Page: Word): Boolean; VIRTUAL;
 PROCEDURE PrintPage(Page: Word; var Rect: TRect; Flags: Word); VIRTUAL;
 PROCEDURE SetPrintParams(ADC: HDC; ASize: TPoint); VIRTUAL;
END;
```

### • El tipo objeto TWindowPrintout

Este tipo objeto permite imprimir los contenidos de una ventana del entorno Windows.

Este tipo está en la *unit Oprinter*.

## LA BIBLIOTECA OBJECTWINDOWS®

```
PWindowPrintout = ^TWindowPrintout;
TWindowPrintout = OBJECT(TPrintOut)
 Window: PWindow;
 Scale: Boolean;
 CONSTRUCTOR Init(ATitle: PChar; AWindow: PWindow);
 FUNCTION GetDialogInfo(var Pages: Integer): Boolean; VIRTUAL;
 PROCEDURE PrintPage(Page: Word; var Rect: TRect; Flags: Word); VIRTUAL;
END;
```

### • El tipo objeto TValidator

*TValidator* define un tipo objeto abstracto para ser usado por sus tipos objeto descendientes en la validación de datos. Dado que es un tipo objeto abstracto no se crearán instancias de *TValidator*.

Este tipo objeto está en la *unit Validate*.

```
PValidator = ^TValidator;
TValidator = object(TObject)
 Status: Word;
 Options: Word;
 CONSTRUCTOR Init;
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Error; VIRTUAL;
 FUNCTION IsValidInput(var S: string;
 SuppressFill: Boolean): Boolean; VIRTUAL;
 FUNCTION IsValid(const S: string): Boolean; VIRTUAL;
 PROCEDURE Store(var S: TStream);
 FUNCTION Transfer(var S: String; Buffer: Pointer;
 Flag: TVTransfer): Word; VIRTUAL;
 FUNCTION Valid(const S: string): Boolean;
END;
```

### • El tipo objeto TPXPictureValidator

Este tipo objeto comprueba la validez de una entrada dada por un fichero de formato gráfico.

Este tipo objeto está en la *unit Validate*.

```
PPXPictureValidator = ^TPXPictureValidator;
TPXPictureValidator = object(TValidator)
 Pic: PString;
 CONSTRUCTOR Init(const APic: string; AutoFill: Boolean);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Error; VIRTUAL;
 FUNCTION IsValidInput(var S: string;
 SuppressFill: Boolean): Boolean; VIRTUAL;
 FUNCTION IsValid(const S: string): Boolean; VIRTUAL;
 FUNCTION Picture(var Input: string;
 AutoFill: Boolean): TPicResult; VIRTUAL;
 PROCEDURE Store(var S: TStream);
END;
```

El resultado de la función *Picture* es el tipo *TPXPictureValidator* está definido por el tipo enumerado:

```
TPicResult = (prComplete, prIncomplete, prEmpty, prError, prSyntax,
 prAmbiguous, prIncompNoFill);
```

### • El tipo objeto TFilterValidator

Este tipo objeto comprueba la validez de un campo de entrada interactiva.

Este tipo objeto está en la *unit Validate*.

```
PFilterValidator = ^TFilterValidator;
TFilterValidator = object(TValidator)
 ValidChars: TCharSet;
 CONSTRUCTOR Init(AValidChars: TCharSet);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Error; VIRTUAL;
 FUNCTION IsValid(const S: string): Boolean; VIRTUAL;
 FUNCTION IsValidInput(var S: string;
 SuppressFill: Boolean): Boolean; VIRTUAL;
 PROCEDURE Store(var S: TStream);
END;
```

### • El tipo objeto TRangeValidator

Este tipo objeto comprueba la validez de un campo de entrada interactiva dentro de un rango de enteros.

Este tipo objeto está en la *unit Validate*.

```
PRangeValidator = ^TRangeValidator;
TRangeValidator = object(TFilterValidator)
 Min, Max: LongInt;
 CONSTRUCTOR Init(Amin, AMax: LongInt);
 CONSTRUCTOR Load(var S: TStream);
 PROCEDURE Error; VIRTUAL;
 FUNCTION IsValid(const S: string): Boolean; VIRTUAL;
 PROCEDURE Store(var S: TStream);
 FUNCTION Transfer(var S: String; Buffer: Pointer;
 Flag: TVTransfer): Word; VIRTUAL;
END;
```

### • El tipo objeto TLookupValidator

Este tipo objeto comprueba la validez de un campo de entrada interactiva de tipo cadena con una lista de valores aceptables.

Este tipo objeto está en la *unit Validate*.

```
PLookupValidator = ^TLookupValidator;
TLookupValidator = object(TValidator)
 FUNCTION IsValid(const S: string): Boolean; VIRTUAL;
 FUNCTION Lookup(const S: string): Boolean; VIRTUAL;
END;
```

### • El tipo objeto TStringLookupValidator

Este tipo objeto comprueba la validez de un campo de entrada interactiva de tipo cadena por medio de la búsqueda a través de una colección de valores aceptables.

Este tipo objeto está en la *unit Validate*.

## LA BIBLIOTECA OBJECTWINDOWS®

```
PStringLookupValidator = ^TStringLookupValidator;
TStringLookupValidator = object(TLookupValidator)
 Strings: PStringCollection;
 CONSTRUCTOR Init(AStrings: PStringCollection);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Error; VIRTUAL;
 FUNCTION Lookup(const S: string): Boolean; VIRTUAL;
 PROCEDURE NewStringList(AStrings: PStringCollection);
 PROCEDURE Store(var S: TStream);
END;
```

### • El tipo objeto TCollection

*TCollection* es un tipo objeto abstracto para construir colecciones de elementos, que a su vez pueden ser objetos. La colección es un concepto más general que los tipos *ARRAY*, conjunto o lista.

Este tipo objeto está en la *unit Objects*.

```
PCollection = ^TCollection;
TCollection = object(TObject)
 Items: PItemList;
 Count: Integer;
 Limit: Integer;
 Delta: Integer;
 CONSTRUCTOR Init(ALimit, ADelta: Integer);
 CONSTRUCTOR Load(var S: TStream);
 DESTRUCTOR Done; VIRTUAL;
 FUNCTION At(Index: Integer): Pointer;
 PROCEDURE AtDelete(Index: Integer);
 PROCEDURE AtFree(Index: Integer);
 PROCEDURE AtInsert(Index: Integer; Item: Pointer);
 PROCEDURE AtPut(Index: Integer; Item: Pointer);
 PROCEDURE Delete(Item: Pointer);
 PROCEDURE DeleteAll;
 PROCEDURE Error(Code, Info: Integer); VIRTUAL;
 FUNCTION FirstThat(Test: Pointer): Pointer;
 PROCEDURE ForEach(Action: Pointer);
 PROCEDURE Free(Item: Pointer);
 PROCEDURE FreeAll;
 PROCEDURE FreeItem(Item: Pointer); VIRTUAL;
 FUNCTION GetItem(var S: TStream): Pointer; VIRTUAL;
 FUNCTION IndexOf(Item: Pointer): Integer; VIRTUAL;
 PROCEDURE Insert(Item: Pointer); VIRTUAL;
 FUNCTION LastThat(Test: Pointer): Pointer;
 PROCEDURE Pack;
 PROCEDURE PutItem(var S: TStream; Item: Pointer); VIRTUAL;
 PROCEDURE SetLimit(ALimit: Integer); VIRTUAL;
 PROCEDURE Store(var S: TStream);
END;
```

Los tipos que utiliza *TCollection* se presentan a continuación:

```
PItemList = ^TItemList;
TItemList = array[0..MaxCollectionSize - 1] of Pointer;
```

### • El tipo objeto TSortedCollection

Este tipo objeto implementa colecciones ordenadas por una clave.

Este tipo objeto está en la *unit Objects*.

```
PSortedCollection = ^TSortedCollection;
TSortedCollection = object(TCollection)
 Duplicates: Boolean;
 CONSTRUCTOR Init(ALimit, ADelta: Integer);
 CONSTRUCTOR Load(var S: TStream);
 FUNCTION Compare(Key1, Key2: Pointer): Integer; VIRTUAL;
 FUNCTION IndexOf(Item: Pointer): Integer; VIRTUAL;
 PROCEDURE Insert(Item: Pointer); VIRTUAL;
 FUNCTION KeyOf(Item: Pointer): Pointer; VIRTUAL;
 FUNCTION Search(Key: Pointer; var Index: Integer): Boolean; VIRTUAL;
 PROCEDURE Store(var S: TStream);
END;
```

#### • El tipo objeto TStrCollection

Este tipo objeto implementa colecciones ordenadas de cadenas ASCII. Se puede redefinir el método *compare* para que tenga en cuenta la ñ y las vocales acentuadas.

Este tipo objeto está en la *unit Objects*.

```
PStringCollection = ^TStringCollection;
TStringCollection = object(TSortedCollection)
 FUNCTION Compare(Key1, Key2: Pointer): Integer; VIRTUAL;
 PROCEDURE FreeItem(Item: Pointer); VIRTUAL;
 FUNCTION GetItem(var S: TStream): Pointer; VIRTUAL;
 PROCEDURE PutItem(var S: TStream; Item: Pointer); VIRTUAL;
END;
```

#### • El tipo objeto TStringCollection

Este tipo objeto implementa colecciones ordenadas de cadenas ASCII. Se puede redefinir el método *compare* para que tenga en cuenta la ñ y las vocales acentuadas.

Este tipo objeto está en la *unit Objects*.

```
PStrCollection = ^TStrCollection;
TStrCollection = object(TSortedCollection)
 FUNCTION Compare(Key1, Key2: Pointer): Integer; VIRTUAL;
 PROCEDURE FreeItem(Item: Pointer); VIRTUAL;
 FUNCTION GetItem(var S: TStream): Pointer; VIRTUAL;
 PROCEDURE PutItem(var S: TStream; Item: Pointer); VIRTUAL;
END;
```

#### • El tipo objeto TStream

*TStream* es un tipo objeto abstracto que permite el manejo de la entrada/salida polimórfica hacia o desde un fichero, también se puede emplear con periféricos.

Este tipo objeto está en la *unit Objects*.

```
PStream = ^TStream;
TStream = object(TObject)
 Status: Integer;
 ErrorInfo: Integer;
 CONSTRUCTOR Init;
 PROCEDURE CopyFrom(var S: TStream; Count: Longint);
 PROCEDURE Error(Code, Info: Integer); VIRTUAL;
```



## LA BIBLIOTECA OBJECTWINDOWS®

```
PROCEDURE Flush; VIRTUAL;
FUNCTION Get: PObject;
FUNCTION GetPos: Longint; VIRTUAL;
FUNCTION GetSize: Longint; VIRTUAL;
PROCEDURE Put(P: PObject);
PROCEDURE Read(var Buf; Count: Word); VIRTUAL;
FUNCTION ReadStr: PString;
PROCEDURE Reset;
PROCEDURE Seek(Pos: Longint); VIRTUAL;
FUNCTION StrRead: PChar;
PROCEDURE StrWrite(P: PChar);
PROCEDURE Truncate; VIRTUAL;
PROCEDURE Write(var Buf; Count: Word); VIRTUAL;
PROCEDURE WriteStr(P: PString);
END;
```

El tipo registro que maneja el tipo objeto *TStream* es el siguiente:

```
PStreamRec = ^TStreamRec;
TStreamRec = record
 ObjType: Word;
 VmtLink: Word;
 Load: Pointer;
 Store: Pointer;
 Next: Word;
END;
```

### • El tipo objeto TMemoryStream

Es un tipo objeto que implementa una especialización de *TStream* sobre memoria.

Este tipo objeto está en la *unit Objects*.

```
PMemoryStream = ^TMemoryStream;
TMemoryStream = object(TStream)
 SegCount: Integer;
 SegList: PWordArray;
 CurSeg: Integer;
 BlockSize: Integer;
 Size: Longint;
 Position: Longint;
 CONSTRUCTOR Init(ALimit: Longint; ABlockSize: Word);
 DESTRUCTOR Done; VIRTUAL;
 FUNCTION GetPos: Longint; VIRTUAL;
 FUNCTION GetSize: Longint; VIRTUAL;
 PROCEDURE Read(var Buf; Count: Word); VIRTUAL;
 PROCEDURE Seek(Pos: Longint); VIRTUAL;
 PROCEDURE Truncate; VIRTUAL;
 PROCEDURE Write(var Buf; Count: Word); VIRTUAL;
PRIVATE
 FUNCTION ChangeListSize(ALimit: Word): Boolean;
END;
```

### • El tipo objeto TDosStream

Es un tipo objeto que implementa una especialización de *TStream* sobre ficheros DOS sin buffer.

Este tipo objeto está en la *unit Objects*.

```

PDosStream = ^TDosStream;
TDosStream = object(TStream)
 Handle: Word;
 CONSTRUCTOR Init(FileName: FNameStr; Mode: Word);
 DESTRUCTOR Done; VIRTUAL;
 FUNCTION GetPos: Longint; VIRTUAL;
 FUNCTION GetSize: Longint; VIRTUAL;
 PROCEDURE Read(var Buf; Count: Word); VIRTUAL;
 PROCEDURE Seek(Pos: Longint); VIRTUAL;
 PROCEDURE Truncate; VIRTUAL;
 PROCEDURE Write(var Buf; Count: Word); VIRTUAL;
END;

```

#### • El tipo objeto TBufStream

Es un tipo objeto que implementa una especialización de *TDosStream* con la incorporación de buffer.

Este tipo objeto está en la *unit Objects*.

```

PBufStream = ^TBufStream;
TBufStream = object(TDosStream)
 Buffer: Pointer;
 BufSize: Word;
 BufPtr: Word;
 BufEnd: Word;
 CONSTRUCTOR Init(FileName: FNameStr; Mode, Size: Word);
 DESTRUCTOR Done; VIRTUAL;
 PROCEDURE Flush; VIRTUAL;
 FUNCTION GetPos: Longint; VIRTUAL;
 FUNCTION GetSize: Longint; VIRTUAL;
 PROCEDURE Read(var Buf; Count: Word); VIRTUAL;
 PROCEDURE Seek(Pos: Longint); VIRTUAL;
 PROCEDURE Truncate; VIRTUAL;
 PROCEDURE Write(var Buf; Count: Word); VIRTUAL;
END;

```

#### • El tipo objeto TEmsStream

Es un tipo objeto que implementa una especialización de *TStream* sobre memoria EMS.

Este tipo objeto está en la *unit Objects*.

```

PEmsStream = ^TEmsStream;
TEmsStream = object(TStream)
 Handle: Word;
 PageCount: Word;
 Size: Longint;
 Position: Longint;
 CONSTRUCTOR Init(MinSize, MaxSize: Longint);
 DESTRUCTOR Done; VIRTUAL;
 FUNCTION GetPos: Longint; VIRTUAL;
 FUNCTION GetSize: Longint; VIRTUAL;
 PROCEDURE Read(var Buf; Count: Word); VIRTUAL;
 PROCEDURE Seek(Pos: Longint); VIRTUAL;
 PROCEDURE Truncate; VIRTUAL;
 PROCEDURE Write(var Buf; Count: Word); VIRTUAL;
END;

```

## LOS RECURSOS

### 15.7 LOS RECURSOS

Los recursos son las partes de un programa Windows que interactúan con los usuarios, es decir son los datos que definen las partes visibles de un programa Windows. La mayor parte de los elementos que se visualizan en las pantallas de un programa Windows son recursos.

El taller de recursos (*resource workshop*) es una herramienta que incorpora *Borland Pascal* para diseñar los recursos, que habitualmente se engloban en un fichero con extensión *.RES*, y que se incluye en el código fuente del programa con la directiva *\$R*. Una vez incluida en el código fuente esta directiva los recursos se manejarán con instrucciones como: *LoadMenu*, *LoadAccelerators*, *LoadCursor*, *LoadIcon*, *AddFontResource*, etc...

Los recursos después de realizarse la compilación del programa fuente están incluidos en el fichero ejecutable (*.EXE*).

### EL TALLER DE RECURSOS

El taller de recursos (*resource workshop*) es una herramienta que incorpora *Borland Pascal* para crear y modificar recursos de Windows. El taller de recursos incluye:

- Un editor gráfico que permite diseñar, modificar y visualizar recursos.
- Un editor de textos para la manipulación de los textos que describen los recursos.
- Un compilador de recursos, que permite compilarlos de forma compatible con el *Microsoft Resource Compiler*.
- Un decompilador de ficheros, que permite modificar los recursos de programas sin acceder al código fuente.

El taller de recursos puede manejar los siguientes tipos de ficheros:

- *Ficheros de recursos compilados (.RES)*.
- *Ficheros de recursos en modo texto (.RC)*.
- *Ficheros ejecutables y DLL (.EXE y .DLL)*.
- *Ficheros de diálogo (.DLG)*.
- *Ficheros de mapas de bits (.BMP)*
- *Ficheros de iconos (.ICO)*
- *Ficheros de cursores (.CUR)*
- *Ficheros binarios de fuentes (.FNT)*
- *Bibliotecas de fuentes (.FON)*
- *Ficheros de dispositivos Windows (.DRV)*

## TIPOS DE RECURSOS

Los tipos de recursos predefinidos son: las teclas aceleradoras, los mapas de bits, los cursores, los cuadros de diálogo, los iconos, los menús, las tablas de cadenas, y las fuentes. También se incluyen en Windows 3.1 los recursos definidos por el usuario, y los recursos de información de versión.

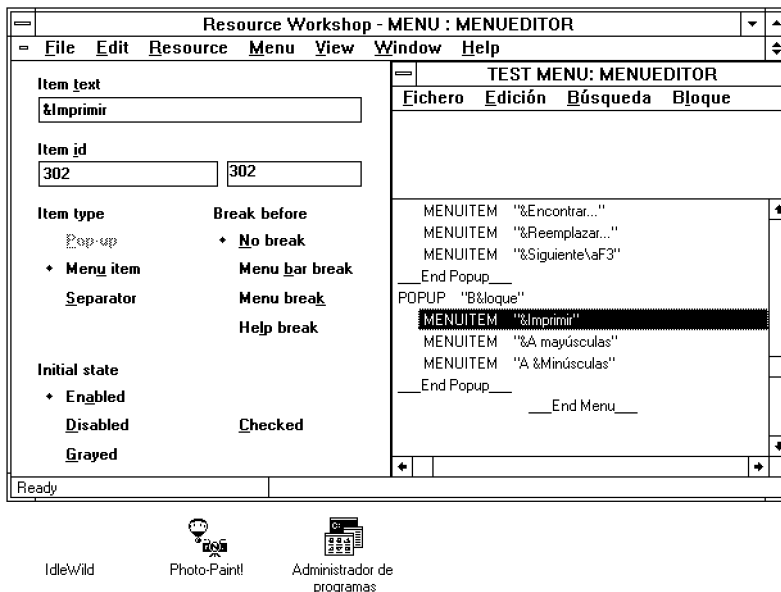


Figura 15.22 Diseño de un menú de editor con el taller de recursos

### • Menús

Los programas windows tienen una barra de menús que contiene la lista de los menús desplegables. Cada menú desplegable contiene uno o más elementos, que a su vez pueden desplegar otros menús, y así sucesivamente. Estos menús se pueden diseñar, editar y modificar con el taller de recursos (figura 15.22).

En la figura 15.22 aparece una de las ventanas del taller de recursos, en el momento de crear el elemento *Imprimir* del menú *Bloque* en el recurso *MENUEDITOR*, que será utilizado en el ejemplo 15.18. Los ficheros del taller de recursos, se guardan compilados con el formato *\*.RES*, pero también se pueden almacenar en modo texto guardándose entonces en la extensión *\*.RC*. A continuación se muestra el fichero completo (*\*.RC*) del recurso *MENUEDITOR* del ejemplo 15.18.

## LOS RECURSOS

```
MENUEEDITOR MENU
BEGIN
 POPUP "&Fichero"
 BEGIN
 MENUITEM "&Nuevo", 24329
 MENUITEM "&Abrir...", 24330
 MENUITEM "&Guardar", 24333
 MENUITEM "Guardar &como...", 24334
 MENUITEM SEPARATOR
 MENUITEM "&Imprimir", 301
 MENUITEM SEPARATOR
 MENUITEM "&Salir", 24340, CHECKED
 END
 POPUP "&Edición"
 BEGIN
 MENUITEM "&Deshacer\aAlt+BkSp", 24325
 MENUITEM SEPARATOR
 MENUITEM "&Cortar\aShift+Del", 24320
 MENUITEM "C&opiar\aCtrl+Ins", 24321
 MENUITEM "&Pegar\aShift+Ins", 24322
 MENUITEM "&Eliminar\aDel", 24323
 MENUITEM "&Borrar Todo\aCtrl+Del", 24324
 END
 POPUP "&Búsqueda"
 BEGIN
 MENUITEM "&Encontrar...", 24326
 MENUITEM "&Reemplazar...", 24327
 MENUITEM "&Siguiente\aF3", 24328
 END
 POPUP "B&loque"
 BEGIN
 MENUITEM "&Imprimir", 302
 MENUITEM "&A mayúsculas", 303
 MENUITEM "A &Minúsculas", 304
 END
END
```

### • Teclas aceleradoras

Las teclas aceleradoras son combinaciones de teclas que realizan una determinada tarea en una aplicación Windows. Por ejemplo **Alt+F4** cierra la ventana de una aplicación Windows.

Las teclas aceleradoras aparecen a la derecha de los menús desplegables. Generalmente los programas Windows utilizan las teclas aceleradoras para un acceso más rápido a los distintos menús. Sin embargo también se pueden crear teclas aceleradoras de opciones no disponibles en los menús, aunque no es aconsejable dado que va en contra de los principios de los GUI.

El taller de recursos permite de forma interactiva la creación, edición y modificación de las teclas aceleradoras (figura 15.23).

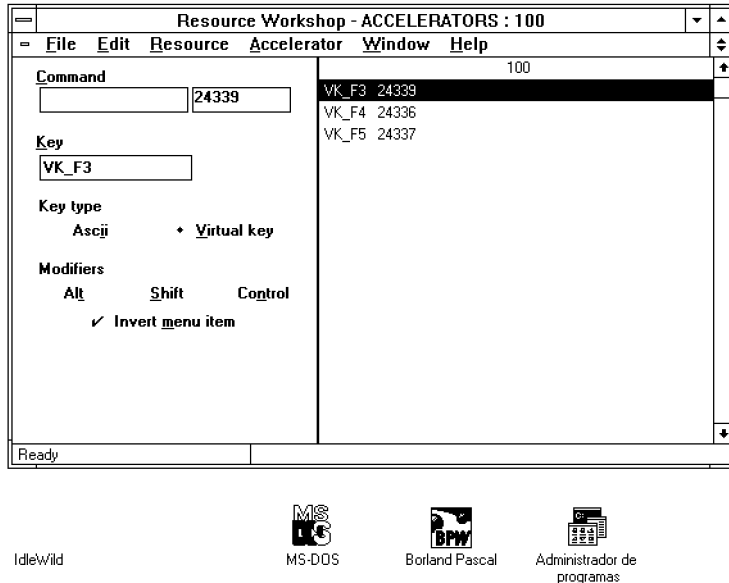


Figura 15.23 Eligiendo teclas aceleradoras con el taller de recursos

En la figura 15.23 aparece una de las ventanas del taller de recursos, en el momento de crear la tecla aceleradora **F3**, que es una de las teclas aceleradoras utilizadas en el ejemplo 15.13. A continuación se muestra la parte de teclas aceleradoras del fichero de recursos (*c15e13.rc*) empleado en el ejemplo 15.13.

```
100 ACCELERATORS LOADONCALL MOVEABLE DISCARDABLE
BEGIN
 VK_F3, 24339, VIRTKEY
 VK_F4, 24336, VIRTKEY, SHIFT
 VK_F5, 24337, VIRTKEY, SHIFT
END
```

### • Mapas de bits

Un mapa de bits es una representación binaria de una imagen gráfica en un programa. Cada bit, o grupos de bits, del mapa de bits representan un pixel de la imagen. El entorno Windows usa gran cantidad de mapas de bits para mostrar los distintos símbolos del entorno: barras de desplazamiento, botones de maximización y minimización, el símbolo del menú de control, etc... Los mapas de bits se almacenan en ficheros con extensión *\*.BMP*.

Para crear mapas de bits se puede utilizar el editor *Paint* del taller de recursos (figura 15.24), aunque también existen gran cantidad de programas en el mercado que crean mapas de bits, por ejemplo el *PaintBrush* que incorpora Windows.

## LOS RECURSOS

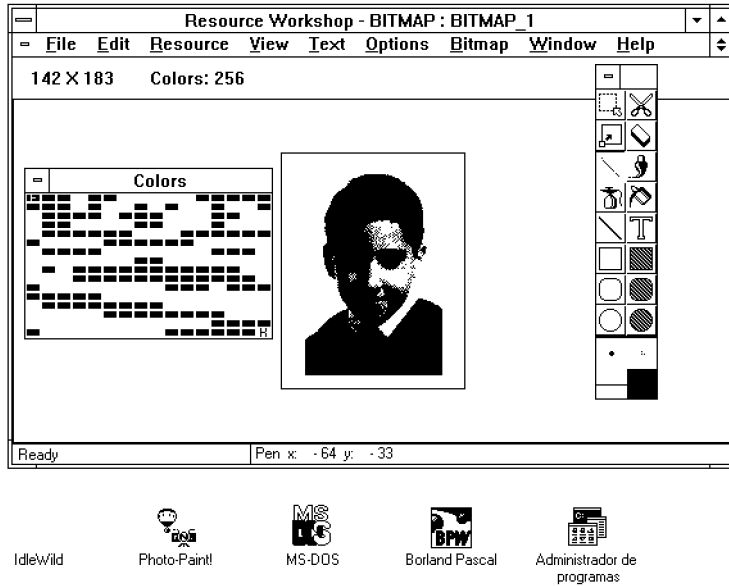


Figura 15.24 Taller de recursos: mapas de bits

El mapa de bits de la figura 15.24 se puede almacenar en un fichero de texto con la extensión .RC, tal y como se presenta a continuación:

```

BITMAP_1 BITMAP
BEGIN
'42 4D 26 6B 00 00 00 00 00 00 36 04 00 00 28 00'
'00 00 8E 00 00 00 B7 00 00 00 01 00 08 00 00 00'
'00 00 F0 66 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 BF 00 00 BF'
'00 00 00 BF BF 00 BF 00 00 00 BF 00 BF 00 BF BF'
'00 00 C0 C0 C0 00 C0 DC C0 00 F0 CA A6 00 80 00'
'00 00 80 80 00 00 00 80 00 00 00 80 80 00 00 00'
'B4 B8 62 B5 62 B8 62 B8 62 B5 62 B8 0E B8 62 62'
...
'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF B8 33'
END

```

### • Cursores

Un cursor es un pequeño mapa de bits que representa la posición del ratón en la pantalla. El tamaño de un cursor es  $32 \times 32$  píxeles. Para crear cursores se utiliza el editor *Paint* del taller de recursos (figura 15.25). El tipo más común de cursor es la flecha con punta.

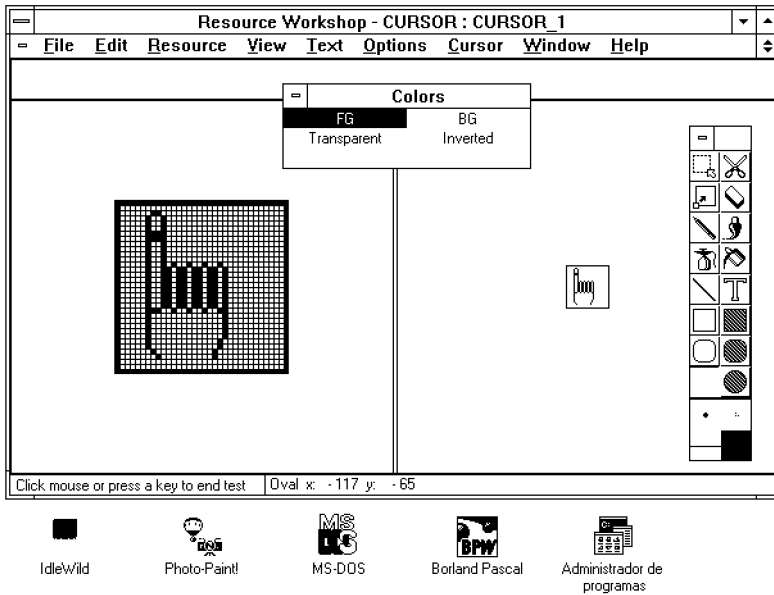


Figura 15.25 Taller de recursos: cursor

El cursor de la figura 15.25 se puede almacenar en un fichero de texto con la extensión *.RC*, tal y como se presenta a continuación:

```
CURSOR_1 CURSOR
BEGIN
'00 00 02 00 01 00 20 20 02 00 10 00 0E 00 34 01'
'00 00 16 00 00 00 28 00 00 00 20 00 00 00 40 00'
'00 00 01 00 01 00 00 00 00 00 00 00 02 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 FF FF FF 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 FF FF FF FF FF FF FF FE FF'
'DF FF FD FF DF FF FB FF EF FF FB FF EF FF FB FF'
'F7 FF FB FF F7 FF FB FF F7 FF FB FF F7 FF FB FF'
'F7 FF F8 DB 67 FF FB 24 97 FF FB 24 97 FF FB 24'
'97 FF FB 24 97 FF FB 24 97 FF FB 24 97 FF FB 24'
'97 FF FB 24 97 FF FB 5B 6F FF FB 7F FF FF FB 7F'
'FF FF FB 7F FF FF FB 7F FF FF F8 7F FF FF F8 7F'
'FF FF FB 7F FF FF FB 7F FF FF FB 7F FF FF FC FF'
'FF FF FF FF FF FF 00 00 00 00'
END
```

END



## LOS RECURSOS

### • Iconos

Los iconos son pequeños mapas de bits que representan a ventanas minimizadas. Se pueden crear con el editor *Paint* del taller de recursos (figura 15.26).

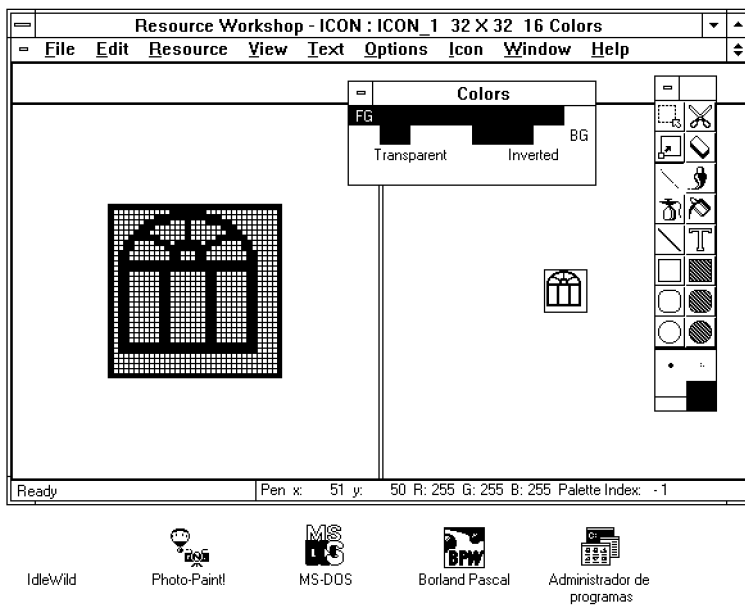


Figura 15.26 Taller de recursos: icono

El icono de la figura 15.26 se puede almacenar en un fichero de texto con la extensión *.RC*, tal y como se presenta a continuación:

```

ICON_1 ICON
BEGIN
'00 00 01 00 01 00 20 20 10 00 00 00 00 00 E8 02'
'00 00 16 00 00 00 28 00 00 00 20 00 00 00 40 00'
'00 00 01 00 04 00 00 00 00 00 80 02 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 80 00 00 80 00 00 00 80 80 00 80 00'
'00 00 80 00 80 00 80 80 00 00 80 80 80 00 C0 C0'
'00 00 00 00 FF 00 00 FF 00 00 00 FF FF 00 FF 00'
'00 00 FF 00 FF 00 FF FF 00 00 FF FF FF 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07'
'77 77 77 77 77 77 77 77 77 77 77 77 00 00 07'
'77 77 77 77 77 77 77 77 77 77 77 77 00 0C 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 77 00 06 C6'
'66 66 66 C6 66 66 66 6C 66 66 66 60 77 00 06 0F'
'FF FF F6 0F FF FF FF 60 FF FF FF 60 77 00 06 0F'
'FF FF F6 0F FF FF FF 60 FF F8 FF 60 77 00 06 0F'
'FF F8 F6 0F FF FF 8F 60 FF 8F FF 60 77 00 06 0F'
'FF 8F F6 0F FF F8 FF 60 FF FF FF 60 77 00 06 0F'
'F8 FF F6 0F FF 8F FF 60 FF FF FF 60 77 00 06 0F'
'8F FF F6 0F F8 FF FF 60 FF FF FF 60 77 00 06 0F'
'FF FF F6 0F 8F FF FF 60 FF FF FF 60 77 00 06 0F'
'FF FF F6 08 FF FF FF 60 FF FF F8 60 77 00 06 0F'

```

PROGRAMACION EN ENTORNO WINDOWS®

```
'FF FF F6 0F FF FF 8F 60 FF FF 8F 60 77 00 06 0F'
'FF FF F6 0F FF F8 FF 60 FF F8 FF 60 77 00 06 0F'
'FF FF 86 0F FF 8F FF 60 FF 8F FF 60 77 00 06 0F'
'FF F8 F6 0F FF FF FF 60 F8 FF FF 60 77 00 06 0F'
'FF 8F F6 0F FF FF FF 60 FF FF FF 60 77 00 06 0F'
'FF FF F6 0F FF FF FF 60 FF FF FF 60 77 00 06 0F'
'00 00 0C 00 00 00 00 C0 00 00 00 C0 77 00 06 C6'
'66 66 66 66 66 66 66 66 66 66 C0 77 00 06 0F'
'FF FF FF F6 0F FF 60 FF FF FF FF 60 70 00 06 0F'
'FF FF FF F6 0F FF 60 FF FF 8F FF 60 70 00 00 60'
'FF 8F FF 00 C0 00 66 0F F8 FF F6 07 00 00 00 60'
'F8 FF 06 6F 66 C6 6F 66 0F FF F6 00 00 00 00 06'
'00 06 6F FF FF C8 FF FF 66 00 60 00 00 00 00 00'
'60 6F FF FF CF FF FF FF 66 00 00 00 00 00 00 00'
'06 00 FF FF F8 CF FF FF F0 0C 00 00 00 00 00 00'
'00 66 00 0F FF CF FF 00 06 60 00 00 00 00 00 00'
'00 00 66 60 00 00 00 66 60 00 00 00 00 00 00'
'00 00 00 06 66 66 66 00 00 00 00 00 00 00 FF FF'
'FF FF FF FF FF FF E0 00 00 03 E0 00 00 03 80 00'
'00 03 80 00 00 03 80 00 00 03 80 00 00 03 80 00'
'00 03 80 00 00 03 80 00 00 03 80 00 00 03 80 00'
'00 03 80 00 00 03 80 00 00 03 80 00 00 03 80 00'
'00 03 80 00 00 03 80 00 00 07 80 00 00 07 C0 00'
'00 0F C0 00 00 1F E0 00 00 3F F0 00 00 7F F8 00'
'00 FF FC 00 01 FF FF 00 07 FF FF E0 3F FF'
```

END

• Las fuentes

Las fuentes (*fonts*) o tipos de letra son un conjunto de caracteres de un tamaño y un estilo dado. Windows maneja tres tipos de fuentes: fuentes *raster*, fuentes vectoriales o escalables, fuentes *TrueType*. El taller de recursos (*Resource Workshop*) tan sólo crea y edita las fuentes de tipo *raster*. Un recurso de fuentes son una serie de datos usados por el ordenador para representar las imágenes en mapas de bits (de caracteres o símbolos) en un periférico de salida, tal como una pantalla o una impresora. Se pueden crear con el editor *Paint* del taller de recursos, tal y como se muestra en la figura 15.27.

Las fuentes de la figura 15.27 también se pueden almacenar en un fichero de texto con la extensión *.RC*, tal y como se presenta a continuación:

```
1 FONT
BEGIN
'00 02 8C 02 00 00 28 63 29 20 43 6F 70 79 72 69'
'67 68 74 20 42 6F 72 6C 61 6E 64 20 49 6E 74 65'
'72 6E 61 74 69 6F 6E 61 6C 2E 20 41 6C 6C 20 72'
'69 67 68 74 73 20 72 65 73 65 72 76 65 64 2E 00'
'00 00 00 00 08 00 64 00 64 00 08 00 00 00 00 00'
'00 00 00 90 01 00 20 00 20 00 00 20 00 20 00 61'
'63 61 00 03 00 00 00 00 00 00 86 02 00 00 00 00'
'00 86 00 00 00 00 20 00 86 00 20 00 06 01 20 00'
'86 01 20 00 06 02 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 FF FF FF FF FF 7F 7F 7F 3F 3F 1F'
'0F 07 03 01 00 00 00 00 01 01 01 00 3E 23 20 20'
'30 1F 00 00 00 FF FF FF FF FF FF FF FF FF FF'
'FF FF FF FF 7F 0F 78 FC FE FE FE FC 78 30 F7 3C'
'78 FE 33 30 30 30 30 30 30 30 FF FF FF FF FF FF'
'FF FF FF FF FE F0 00 00 00 00 00 00 00 F0 98 08'
'08 18 F0 00 00 00 00 00 00 00 00 FE FE FE FC FC F8'
'F0 E0 C0 80 00 00 00 00 01 02 04 08 10 20 20 40'
```

LOS RECURSOS

```
'40 40 80 80 80 80 80 80 80 80 40 40 40 20 21 12'
'0C 04 02 01 00 00 0F 71 81 01 01 01 01 01 01'
'01 01 01 01 01 01 01 03 05 09 11 21 41 81 01 01'
'01 01 01 81 71 0F E0 1C 03 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 80 40 20 10 08 04 02 01 00'
'00 00 00 03 1C E0 00 00 00 80 40 20 10 08 08 04'
'04 04 02 02 02 02 02 02 02 04 04 04 08 08 90'
'60 60 80 00 00 00 00 00 00 00 00 00 00 00 00'
'01 01 01 03 03 03 07 07 07 0F 0F 0F 1F 1F 3F 3F'
'3F 7D 7C 7C F8 F8 1F 1F 3F 3F 3F 79 78 7D F0 F5'
'F0 F5 E0 F5 E0 EC C1 EA 8B D8 88 47 3F FF FF FE'
'F0 80 00 00 00 00 C0 E0 E0 F0 F0 F8 FC FC 7E 7E'
'3F 1F 5F 0F 4F 07 6F 03 23 1F FF FF FF F8 C0 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 80 80 C0 C0 E0 F0 F0 F8 F8 FC FE 7E 3F 3F 1F'
'1F 0F 07 07 03 03 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
'00 00 00 00 00 00 66 69 78 65 64 00'
```

END

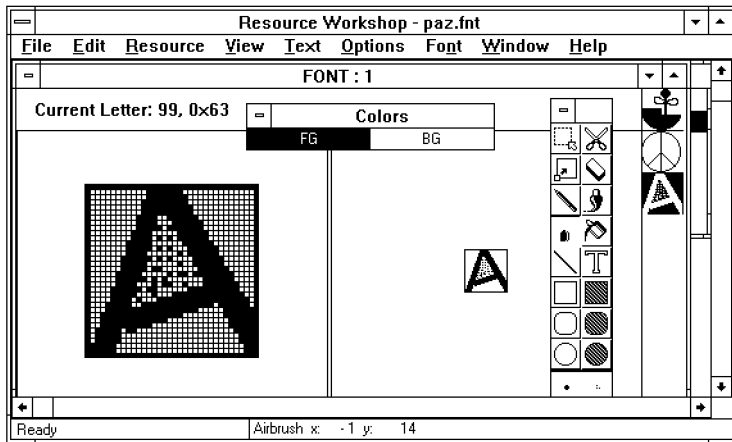


Figura 15.27 Taller de recursos: fuentes

• Cuadros de diálogo

Un cuadro o caja de diálogo (*dialog box*) es una ventana que comunica información al usuario y le permite seleccionar opciones tales como abrir ficheros, mostrar colores, búsqueda de un texto, etc... Un cuadro de diálogo habitualmente incluye controles como: botones de radio, cajas de comprobación, botones de pulsar, etc...

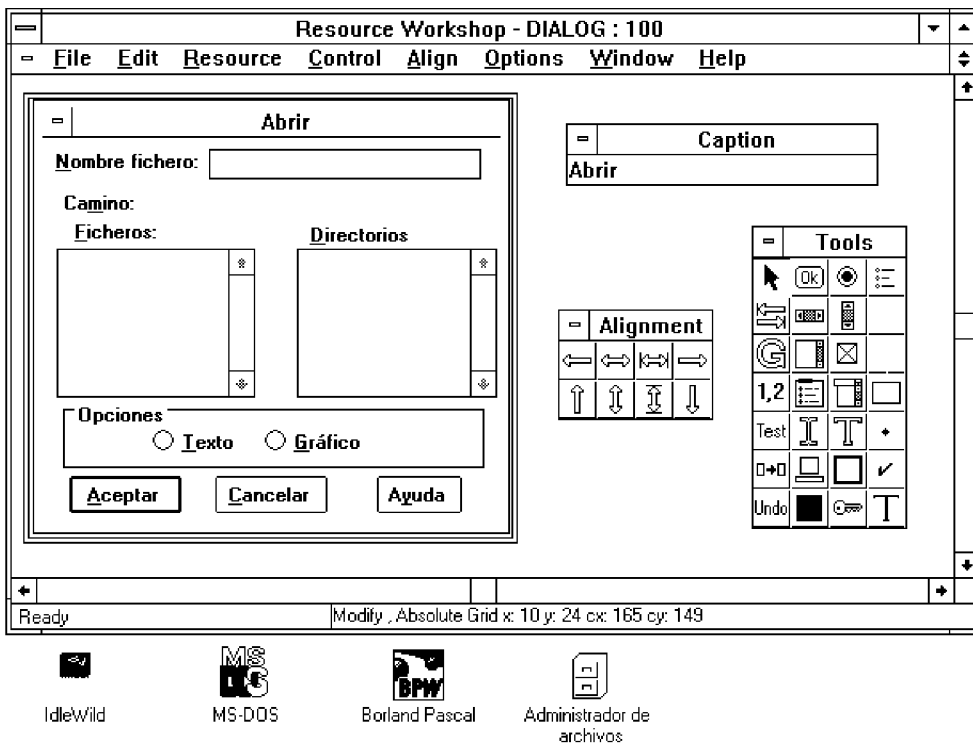


Figura 15.28 Taller de recursos: cuadros de diálogo

Los cuadros de diálogo se suelen utilizar para proporcionar más información de la que se puede manejar fácilmente con menús. Las directivas marcadas por la especificación IBM SAA/CUA, indican que una opción de menú que conduce a un cuadro de diálogo, irá seguida de puntos suspensivos. Así por ejemplo la opción de menú *Abrir fichero...* conduce a un cuadro de diálogo.

Las cajas de diálogo se pueden clasificar por su comportamiento en *cuadros de diálogo modales* y *cuadros de diálogo amodales*.

## LOS RECURSOS

Los cuadros de diálogo *modales* no permiten que el usuario pase a otra parte de la aplicación dejando el cuadro de diálogo desplegado y sin finalizar sus opciones. Para salir de estos cuadros de diálogo es necesario acabar las opciones o elegir **Cancelar**. Estas son el tipo de cajas de diálogo más usadas en Windows. Esto no impide que se pueda que se pueda desplazar la ventana de la caja de diálogo, o que se pueda cambiar de tarea con **Alt+Esc**. Aunque existe un tipo de caja de diálogo modal, denominado *System Modal*, que no permite cambiar de tarea.

Los cuadros de diálogo *amodales* permiten que el usuario pase a otra parte de la aplicación dejando el cuadro de diálogo desplegado y sin finalizar sus opciones, es decir no toma el control exclusivo del teclado y del ratón.

En la figura 15.28 se muestra el diseño con el taller de recursos de un típico cuadro de diálogo para abrir ficheros.

El cuadro de diálogo de la figura 15.28 también se puede almacenar como fichero de texto con la extensión *.RC*, tal y como se muestra a continuación:

```
100 DIALOG 10, 24, 165, 149
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Abrir"
FONT 8, "Helv"
BEGIN
 CONTROL "&Nombre fichero:", -1, "STATIC", SS_LEFT | WS_CHILD | WS_VISIBLE |
WS_GROUP, 5, 5, 60, 12
 CONTROL "", 100, "EDIT", ES_LEFT | ES_AUTOHSCROLL | ES_OEMCONVERT |
WS_CHILD | WS_VISIBLE | WS_BORDER | WS_GROUP | WS_TABSTOP, 60, 4, 97, 12
 CONTROL "Ca&mino:", -1, "STATIC", SS_LEFT | WS_CHILD | WS_VISIBLE |
WS_GROUP, 8, 21, 30, 8
 LTEXT "", 101, 40, 21, 116, 9
 CONTROL "&Ficheros:", -1, "STATIC", SS_LEFT | WS_CHILD | WS_VISIBLE |
WS_GROUP, 12, 32, 40, 8
 CONTROL "", 102, "LISTBOX", LBS_STANDARD | WS_CHILD | WS_VISIBLE | WS_TABS-
TOP, 6, 43, 70, 59
 CONTROL "&Directorios", -1, "STATIC", SS_LEFT | WS_CHILD | WS_VISIBLE |
WS_GROUP, 96, 33, 51, 9
 CONTROL "", 103, "LISTBOX", LBS_STANDARD | WS_CHILD | WS_VISIBLE | WS_TABS-
TOP, 92, 43, 70, 59
 CONTROL "Opciones", -1, "BUTTON", BS_GROUPBOX | WS_CHILD | WS_VISIBLE |
WS_GROUP, 8, 100, 150, 27
 CONTROL "&Texto", 205, "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE
| WS_GROUP | WS_TABSTOP, 40, 110, 37, 12
 CONTROL "&Gr&fico", 207, "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD | WS_VISI-
BLE, 80, 110, 35, 12
 CONTROL "&Aceptar", 1, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE |
WS_GROUP | WS_TABSTOP, 10, 130, 40, 14
 CONTROL "&Cancelar", 2, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE |
WS_TABSTOP, 62, 130, 40, 14
 CONTROL "A&yuda", 998, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE |
WS_TABSTOP, 120, 130, 30, 14
END
```

### • Tablas de cadenas

Las tablas de cadenas (*string tables*) contienen textos (como descripciones, avisos, y mensajes de error) que se muestran en un programa windows. Se pueden definir varias tablas de cadenas en el fichero de recursos (\*.RES). Habitualmente se define una tabla de cadenas para cada agrupación de elementos del programa.

La definición de cadenas de texto como recursos separados del programa fuente, facilita la traducción de las aplicaciones Windows a distintos idiomas sin tener que tocar el código fuente. Aún así es necesario traducir otros recursos como son los cuadros de diálogo.

Las tablas de cadenas se pueden crear con el taller de recursos tal y como se muestra en la figura 15.29.

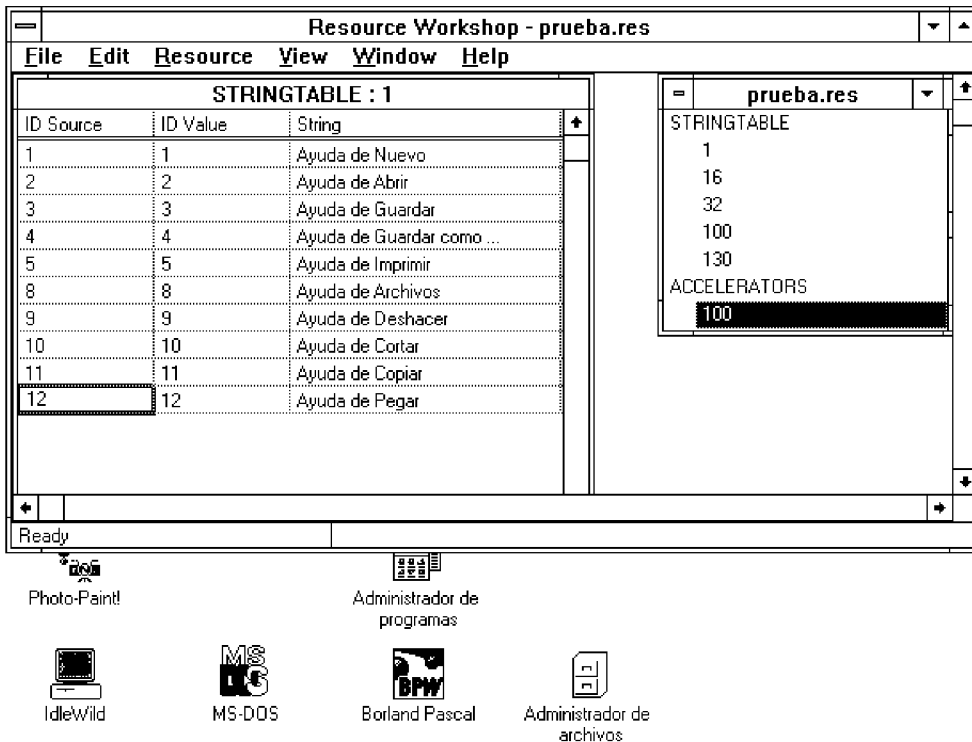


Figura 15.29 Taller de recursos: tablas de cadenas

La representación interna de la tabla de cadenas de la figura 15.29 es el fichero RC siguiente:

## LOS RECURSOS

```
STRINGTABLE LOADONCALL MOVEABLE DISCARDABLE
BEGIN
 1, "Ayuda de Nuevo"
 2, "Ayuda de Abrir"
 3, "Ayuda de Guardar"
 4, "Ayuda de Guardar como ..."
 5, "Ayuda de Imprimir"
 8, "Ayuda de Archivos"
 9, "Ayuda de Deshacer"
 10, "Ayuda de Cortar"
 11, "Ayuda de Copiar"
 12, "Ayuda de Pegar"
END
```

### • Los recursos de información de versión

Los recursos de información de versión fueron añadidos en Windows 3.1, y contienen: número de versión de fichero, número de versión de producto, sistema operativo, tipo de fichero, y función del fichero. Este recurso se han diseñado para ser utilizado con las funciones de instalación de ficheros. Los recursos de información de versión se escriben en modo texto. Así a continuación se presenta un ejemplo de recurso de información de versión. El código hexadecimal *040A0000* representa al idioma (Español Castellano) según la tabla manejada por Windows.

```
VERSIONINFO_1 VERSIONINFO
FILEVERSION 1,0
FILEOS VOS__WINDOWS16
FILETYPE VFT_APP
BEGIN
 BLOCK "StringFileInfo"
 BEGIN
 BLOCK "040A0000"
 BEGIN
 VALUE "Empresa", "Universidad de Oviedo\000"
 VALUE "Aplicación", "Ejemplos de Windows en Pascal\000"
 VALUE "Copyright", "J.M. Cueva et al.\000"
 END
 END
END
```

### • Los recursos definidos por el usuario

Los recursos definidos por el usuario son cualquier conjunto de datos que el usuario desea añadir a su fichero ejecutable. Por ejemplo pueden añadirse: grandes bloques de inicialización y datos de sólo lectura, como los ficheros de texto.

## PROGRAMANDO CON RECURSOS

Los recursos se crean con el taller de recursos, aunque también es posible crearlos con otros programas comerciales. En este apartado se explicará el manejo de los creados con el taller de recursos que incorpora *Borland Pascal*.

Para manejar los recursos en un programa fuente es necesario la inclusión de la directiva *\$R* seguida de un fichero de recursos en formato binario *.RES*. Un programa en Pascal sólo permite la inclusión de un sólo fichero de recursos. Este fichero puede a su vez incluir otros ficheros de recursos.

Un fragmento de código con la forma de incluir un fichero de recursos se muestra a continuación:

```
PROGRAM Prueba;
 {$R prueba.res}
```

Una vez que el recurso ha sido incluido en el fichero fuente con la instrucción *\$R*, debe ser explícitamente cargado en la aplicación antes de ser manejado. La forma de cargar un recurso depende del tipo de recurso.

#### • Cargando menús

La barra de menús de una ventana está dentro del campo *Attr* del tipo objeto *TWindow*. Este campo es un registro variante de tipo *TWindowAttr* que almacena los atributos de las instancias de *TWindows*, y que se muestra a continuación:

```
TWindowAttr = RECORD
 Title: PChar;
 Style: LongInt;
 ExStyle: LongInt;
 X,Y,W,H: Integer;
 Param: Pointer;
 CASE Integer OF
 0: (Menu:HMenu);
 1: (Id: Integer);
 END;
```

Para cargar una barra de menús se llama a la función *Windows LoadMenu*, antes de que se llame a la función de crear ventana. Habitualmente la función *LoadMenu* se introduce en un método constructor *Init*, tal y como se muestra en el fragmento de código siguiente:

```
PROGRAM Prueba;
 {$R prueba.res}
 ...
TYPE
 PMiVentanaMenu=^TmiVentanaMenu;
 TmiVentanaMenu=OBJECT(TWindow)
 CONSTRUCTOR Init(UnPadre:PWindowsObject;UnTitulo:PChar);
 ...
END;

CONSTRUCTOR TmiVentanaMenu.Init;
BEGIN
 INHERITED Init (UnPadre,UnTitulo);
 Attr.Menu := LoadMenu(HInstance, MakeIntResource(100));
 ...
END;
...
```



## LOS RECURSOS

*LoadMenu* carga el recurso menú con el campo *Id* con valor 100, en el *handle HInstance*, y devuelve un *handle* de tipo *HMenu*. *MakeIntResource(100)* convierte el entero 100 en un tipo *PChar*, también se podría haber usado *PChar(100)* para forzar el tipo a *PChar*.

Un recurso puede tener un nombre simbólico, dado por una cadena de caracteres, como '*EjemploDeMenu*'. Entonces otra forma de usar *LoadMenu* se muestra en el siguiente fragmento de programa:

```
...
CONSTRUCTOR TmiVentanaMenu.Init;
BEGIN
 INHERITED Init (UnPadre,UnTitulo);
 Attr.Menu := LoadMenu(HInstance, 'EjemploDeMenu');
...
END;
...
```

En el ejemplo 15.13 se muestra un programa completo que carga un recurso de menú.

### • Cargando teclas aceleradoras

Los recursos de teclas aceleradoras se almacenan en la tabla de aceleradores, con la función Windows *LoadAccelerators*, que devuelve un *handle* de dicha tabla. A diferencia de los menús que están asociados a una determinada ventana, las teclas aceleradoras están asociadas a la aplicación completa. Cada aplicación sólo puede tener una tabla de teclas aceleradoras, contenida en el campo *HAccTable* del tipo objeto *TApplication*. Habitualmente la carga de teclas aceleradoras se realiza en la redefinición del método *InitInstance* del tipo objeto *TApplication*, por ejemplo:

```
PROGRAM Prueba;
{$R prueba.res}
...
TYPE
 PMiAplicacion=^TmiAplicacion;
 TmiAplicacion=OBJECT(TApplication)
 ...
 PROCEDURE InitInstance; VIRTUAL;
 ...
END;

PROCEDURE TmiAplicacion.InitInstance;
BEGIN
 INHERITED InitInstance;
 HAccTable := LoadAccelerators(HInstance, MakeIntResource(100));
...
END;
...
```

Un recurso puede tener un nombre simbólico, dado por una cadena de caracteres, como '*EjemploDeAceleradoras*'. Entonces otra forma de usar *LoadAccelerators* se muestra en el siguiente fragmento de programa:

```

...
PROCEDURE TmiAplicacion.InitInstance;
BEGIN
 INHERITED InitInstance;
 HAccTable := LoadAccelerators(HInstance, 'EjemploDeAceleradoras');
 ...
END;
...

```

En el ejemplo 15.13 se muestra un programa completo que carga un recurso de teclas aceleradoras.

#### • Cargando cuadros de diálogo

Los cuadros de diálogo son los únicos recursos que tienen una correspondencia directa con los tipos objeto de *ObjectWindows*. El tipo objeto *TDialog* y sus descendientes, incluyendo el *TDlgWindow*, definen objetos que utilizan cuadros de diálogo como interfaz con el usuario. Cada objeto cuadro de diálogo se asocia habitualmente con un recurso cuadro de diálogo, que especifica su tamaño, localización, y los controles asociados, tales como botones y cajas de listas (*list boxes*).

Los recursos de un objeto cuadro de diálogo se cargan cuando se construye el objeto cuadro de diálogo. Al igual que los recursos de menú y de las teclas aceleradoras, los recursos de cuadros de diálogo se pueden describir por medio de un nombre simbólico o un entero ID. Por ejemplo:

```
UnDlg:=New(PEjemploDialogo, Init(@Self,'CuadroInformativo'));
```

o también

```
UnDlg:=New(PEjemploDialogo, Init(@Self,MakeIntResource(120)));
```

#### • Cargando cursores e iconos

Para cargar un cursor o un icono es necesario redefinir el método *GetWindowClass* del tipo objeto *TWindow*. Este método se encarga de manejar los denominados atributos de registro de la ventana (*registration attributes*), denominados así dado que se establecen cuando se registra una ventana en Windows. Estos atributos están en el tipo registro *TWndClass*, que está definido y mantenido por el entorno Windows.

Hay una diferencia entre los cursores y los iconos. Los cursores se especifican para cada ventana, mientras los iconos representan la aplicación completa. Así, sólo se establece un icono para la ventana principal (*main*) de la aplicación. Una excepción a la regla de un icono por aplicación son las aplicaciones MDI (*Multiple Document Interface*), en las cuales cada ventana MDI hija tiene su propio icono. Un ejemplo de la redefinición del método *GetWindowClass* y la carga de iconos y cursores se muestra en el siguiente fragmento de código:

```

PROGRAM Prueba;
{$R prueba.res}
...
TYPE
 PEjemploVentana=^TEjemploVentana;
 TEjemploVentana=OBJECT(TWindow)

```

## LOS RECURSOS

```
PROCEDURE GetWindowClass(VAR UnaClaseVentana:TWndClass);
...
END;

PROCEDURE TEjemploVentana.GetWindowClass;
BEGIN
 INHERITED GetWindowClass(UnaClaseVentana);
 UnaClaseVentana.hCursor := LoadCursor(HInstance, 'dedo');
 UnaClaseVentana.hIcon := LoadIcon (HInstance, 'UnIcono');
END;
...
```

### • Cargando tablas de cadenas

Una de las razones principales para definir recursos de tablas de cadenas es facilitar la adaptación de las aplicaciones a casos particulares, tales como traducciones de la aplicación a distintos idiomas. Si las cadenas están definidas en el código fuente es necesario buscarlas y cambiarlas. Sin embargo si están definidas como recursos, sólo es necesario modificarlas con el editor de cadenas del taller de recursos, sin tocar para nada el código fuente. Cada fichero ejecutable tiene sólo un recurso de tabla de cadenas.

Otra de las razones de utilizar recursos de tablas de cadenas, es que suelen ahorrar espacio de memoria, dado que el compilador añade los recursos de tablas de cadena en distintos segmentos de datos, dependiendo del número ID de identificación de la cadena.

Los recursos de tablas de cadenas se pueden utilizar en cualquier sitio de la aplicación, tan sólo hay que llamar a la función *LoadString*, que devolverá en uno de sus parámetros un puntero a la cadena. La sintaxis de *LoadString* es la siguiente:

- ```
LoadString(HInstance, IDcadena, @cadena, SizeOf(cadena));
```
- *IDcadena*. Es un entero que identifica la cadena en la tabla de cadenas.
 - *@cadena*. Es un puntero a la cadena, de tipo *PChar*. Es el parámetro que recibe la cadena
 - *SizeOf(cadena)*. Es el tamaño máximo de la cadena. Este parámetro no puede ser superior a 255 caracteres.

LoadString devuelve el número de caracteres copiados o cero si el recurso no existe.

• Cargando mapas de bits

La función *LoadBitmap* carga los recursos de mapas de bits en memoria, y devuelve su *handle*. Un ejemplo de su manejo es el siguiente:

```
HMiMapaDeBits:=LoadBitmap(HInstance, MakeIntResource(501));
```

donde se carga el recurso identificado por 501 y se almacena el *handle* resultante en la variable *HMiMapaDeBits*. Una vez que un mapa de bits está cargado en memoria, permanece en memoria hasta que sea eliminado explícitamente. A diferencia de otros recursos, permanece incluso después de finalizar la aplicación.

Para eliminar un mapa de bits de memoria se utiliza la función *DeleteObject*, que devuelve un tipo *Bool*. Por ejemplo para eliminar el mapa de bits anterior sería:

```
IF DeleteObject(HMiMapaDeBits) THEN (* eliminado... *);
```

En el ejercicio resuelto 15.2 se escribe una aplicación que maneja mapas de bits.

15.8 LAS FUNCIONES API DE WINDOWS

Windows 3.0 incorpora casi 600 funciones API (*Applications Programming Interface*). Windows 3.1 añade 400, con lo que se tienen del orden de 1000 funciones, que se pueden clasificar en tres grupos:

- *Funciones de gestión del interfaz Windows*. Estas funciones gestionan el procesamiento de mensajes; la creación, dimensionado y movimiento de ventanas; y el manejo de salidas. Aunque *ObjectWindows* maneja la mayor parte de estas funciones encapsuladas dentro de tipos objeto, en algunos casos es necesario utilizar algunas de ellas (especialmente las funciones de manejo de mensajes). Véase el apartado *Enviar Mensajes* dentro del epígrafe 15.9 de este capítulo.
- *Funciones de interfaz gráfico de dispositivos (GDI, Graphics Device Interface)*. Estas funciones llevan a cabo operaciones gráficas independientes del dispositivo en las aplicaciones Windows. Dichas operaciones incluyen escribir texto, dibujar distintos tipos de líneas e imágenes en forma de mapas de bits sobre diferentes dispositivos de salida. Estas operaciones no están implementadas en la jerarquía de *ObjectWindows*.
- *Funciones del interfaz de servicios del sistema*. Estas funciones acceden a los datos y código que están en distintos módulos, asignan memoria local y global, gestionan tareas, cargan los recursos, manipulan cadenas, cambian los archivos de inicialización de Windows, ofrecen ayudas para la depuración (*debuggin*), realizan operaciones de entrada y salida en ficheros y puertas de comunicaciones, así como el manejo de sonidos. La mayor parte de estas funciones no están en la jerarquía de *ObjectWindows*.

Para manejar las funciones API de Windows 3.0 es necesario incluir la *unit WinProcs*. Esta *unit* define la cabecera de cada procedimiento o función API de Windows, como cualquier subprograma Pascal. Para ver las funciones API puede utilizarse la ayuda en línea del Borland Pascal de la *unit WinProcs*.

Las funciones API de Windows 3.0 manejan constantes y tipos de datos definidos en la *unit WinTypes*. Habitualmente siempre que se manejan directamente funciones API deben incluirse las *units WinProcs* y *WinTypes*.

Las funciones API de Windows 3.1 están en las *units* de la tabla 15.6.

LAS FUNCIONES API DE WINDOWS

La forma más cómoda para conocer las distintas funciones API de Windows es escribir la función en el editor del entorno integrado de *Borland Pascal* y pedir la ayuda en línea sobre dicha función pulsando la combinación de teclas **Ctrl** + **F1**. Otra forma es pedir ayuda sobre una *unit* determinada.

LA NOTACION HUNGARA

Los nombres de los tipos que manejan las funciones API de Windows tienen unos nombres de apariencia en un principio extraña, es debido a lo que se denomina notación húngara, en honor al origen de su creador *Charles Simonyi* de *Microsoft*.

En la notación húngara se escriben los nombres de los tipos anteponiéndoles un prefijo con el tipo de datos. Así por ejemplo el prefijo *h* de *hIcon* indica que es un *handle*. Algunos prefijos pueden ser combinación de varios. Así el prefijo *lpsz* en *lpszWindowClass* indica que es un puntero largo a una cadena terminada en carácter nulo (o puntero largo a cadena ASCII). La tabla 15.7 muestra los prefijos más comunes de la notación húngara.

Prefijo	Descripción
b	Tipo boolean o Bool
by	Tipo byte
c	Tipo carácter
cb	Tipo contador de bytes
cx, cy	Entero contador de coordenadas x o y
fn	Tipo función
h	Tipo <i>Handle</i>
i	Tipo entero
l	Tipo entero largo
lp	Tipo puntero largo
n	Número, generalmente entero corto
sz	Cadena acabada en carácter nulo
w	Tipo word
x, y	Tipo entero para coordenadas x o y

Tabla 15.7 Prefijos de la notación húngara

Generalmente los prefijos están en minúscula, y los nombres que les siguen combinan mayúsculas y minúsculas, aunque el compilador no es sensible a la diferencia entre mayúsculas y minúsculas. Tan sólo se utiliza para facilitar la legibilidad de los programas.

La notación húngara es una convención procedente del lenguaje C, para que el programador sea consciente en cada momento de los tipos de las variables que maneja, dado que el lenguaje C no realiza comprobaciones de tipo. En cambio en el lenguaje Pascal hace una estricta comprobación

de tipos, por lo que la notación húngara no es tan importante como en C. Sin embargo se mantiene la notación húngara en Pascal por compatibilidad con la documentación de Windows, que viene preparada siempre para C, y en algunos casos para otros lenguajes.

MANEJO DE FUNCIONES API

La jerarquía de tipos objeto *ObjectWindows* permite el manejo del entorno Windows de una forma cómoda, sin embargo no se puede hacer todo con dicha jerarquía, es necesario utilizar funciones API de Windows. Las funciones API de Windows 3.0 se pueden utilizar en cualquier parte del programa, con la única condición de incluir las *units WinProcs* y *WinTypes*. Así en los ejemplos elementales 15.7 y 15.8, ya se utilizaron funciones API para dibujar un círculo o escribir un texto en una ventana. Para el manejo de las funciones API de Windows 3.1 es necesario incluir alguna de las *units* de la tabla 15.6.

EL INTERFAZ GRAFICO DE DISPOSITIVOS (GDI)

Windows tiene un conjunto de funciones API que se denominan GDI (*Graphics Device Interface*), que permiten desarrollar aplicaciones con capacidades de dibujo y escritura independientes del dispositivo gráfico utilizado para hacer la representación. Por ejemplo se utilizan las mismas funciones para dibujar en una pantalla VGA, SuperVGA, o una impresora *PostScript*®.

EL CONTEXTO DE REPRESENTACION (DC)

Cuando se quiere escribir o dibujar en un dispositivo de salida gráfico (como puede ser una pantalla o una impresora), lo primero que se tiene que obtener es el *handle* de un contexto de representación (o DC, *Display Context*) de tipo *HDC*. Una vez que el entorno Windows suministra el *handle*, ya se puede utilizar dicho contexto para escribir o pintar en él.

Las funciones GDI del API de Windows utilizadas para escribir o dibujar siempre se les tiene que pasar como parámetro el *handle* del contexto de representación. Por ejemplo las funciones GDI (*TextOut*, *Lineto*, *Ellipse*,...) se encargan de escribir un texto o pintar una elipse u otra figura en unas determinadas coordenadas de una ventana.

A continuación se muestran algunas de las funciones más comunes para escribir o pintar en los contextos de representación.

```
TextOut(UnHDC, x, y, s, StrLen(s));
LineTo(UnHDC, Msg.LParamLo, Msg.LParamHi);
Ellipse(UnHDC, xi, yi, xd, yd);
MoveTo(UnHDC,x,y);
Polyline(UnHDC,@Puntos,10);
Arc(UnHDC,xi,yi,xd,yd,x1,y1,x2,y2);
Rectangle(UnHDC,xi, yi, xd, yd);
RoundRect(UnHDC,xi, yi, xd, yd);
Pie(UnHDC,xi,yi,xd,yd,x1,y1,x2,y2);
Chord(UnHDC,xi,yi,xd,yd,x1,y1,x2,y2);
Polygon(UnHDC,@Puntos,10);
```

LAS FUNCIONES API DE WINDOWS

Un contexto de representación contiene muchos *atributos* como son tipo de letra, color del texto, color de fondo detrás del texto, espaciado de los caracteres, etc... Inicialmente tienen unos valores por defecto, que se pueden cambiar por medio de funciones GDI de modificación de atributos del contexto de representación.

Las funciones GDI (*TextOut*, *LineTo*, *Ellipse*,...) trabajan con los atributos del contexto de representación activos en el momento de su ejecución. Si se llama a una función que cambia el atributo del contexto de representación, las llamadas subsiguientes a *TextOut*, *LineTo*, *Ellipse*,... utilizarán los nuevos atributos.

Los requerimientos de memoria de los contextos de representación son muy altos, de tal forma que el sistema Windows tan sólo soporta cinco contextos de representación accesibles concurrentemente en una sesión Windows. Esto significa que cada ventana no puede mantener su propio contexto de representación, es decir debe obtener sólo uno cuando lo necesita, y liberarlo tan pronto como sea posible.

• Forma 1 de obtener y liberar el *handle* del contexto de representación

La forma habitual de escribir o pintar en una ventana es redefiniendo el método *Paint* de *TWindow*. Si se realiza de esta forma la jerarquía *ObjectWindows* ya se encarga de obtener y liberar el *handle* automáticamente. El siguiente fragmento de programa muestra la forma de hacerlo.

```
...
TYPE
  PVentanaCirculo=^TVentanaCirculo;
  TVentanaCirculo= OBJECT (TWindow)
    PROCEDURE Paint (PaintDC:HDC; VAR PaintInfo:TPaintStruct); VIRTUAL;
  END;

  TMiAplicacion=OBJECT(TApplication)
    PROCEDURE InitMainWindow; VIRTUAL;
  END;

  PROCEDURE TVentanaCirculo.Paint;
  BEGIN
    Ellipse(PaintDC, 10, 10, 100,100);
  END;
...
```

Un programa completo muy simple que muestra esta forma de trabajo es el ejemplo 15.7.

Con el *handle* del contexto de representación obtenido de esta forma sólo se puede pintar o escribir dentro del área de trabajo.

• Forma 2 de obtener y liberar el *handle* del contexto de representación

Otros métodos distintos de *Paint* también pueden obtener un *handle* del contexto de representación de forma directa utilizando las funciones GDI de Windows, pero en este caso se tienen que encargar también de liberar el contexto. En el siguiente fragmento de programa se utilizan las funciones *GetDC* y *ReleaseDC* para obtener y liberar el contexto de representación.

```

TYPE
  PEjemploWindow = ^TEjemploWindow;
  TEjemploWindow = object(TWindow)
    UnDC: HDC;
    PROCEDURE WMLButtonDown(VAR Msg: TMessage);
    VIRTUAL wm_First + wm_LButtonDown;
  END;

PROCEDURE TEjemploWindow.WMLButtonDown(VAR Msg: TMessage);
...
BEGIN
  UnDC := GetDC(HWindow); (* Obtiene el handle de contexto de representación *)
  ...
  TextOut(UnDC, Msg.LParamLo, Msg.LParamHi, S, StrLen(S)); (* Escribe *)
  ...
  ReleaseDC(HWindow, UnDC); (* Libera el handle *)
END;

```

Un programa completo muy simple que muestra esta forma de trabajo es el ejemplo 15.17.

Con el *handle* del contexto de representación obtenido de esta forma sólo se puede pintar o escribir dentro del área de trabajo. Existen otras funciones GDI para obtener y liberar directamente el contexto de representación:

GetWindowDC Obtiene el *handle* de un contexto de representación que sea aplicable a toda la ventana, y no sólo al área de trabajo.

CreateDC Obtiene el *handle* de un contexto de representación que sea aplicable a toda la pantalla, y no sólo a una ventana. El *handle* debe ser liberado usando la función *DeleteDC*.

FUNCIONES CALLBACK

Algunas funciones API de Windows requieren como parámetro un puntero a unas funciones denominadas *callback* o "de retorno". Estas funciones *callback* son funciones que se colocan dentro de una aplicación Windows, y son llamadas por el entorno Windows.

Las funciones *callback* se construyen como funciones normales, con la indicación *FAR* y *EXPORT*. No pueden utilizarse como funciones *callback* los métodos función de tipos objeto. A continuación se presenta un ejemplo de declaración de una función que va a ser utilizada como *callback*:

```
FUNCTION ActuaSobreVentana (UnHandle:HWnd; UnValor:Longint):Integer;FAR;EXPORT;
```

Un puntero a esta función *callback* se pasa como primer parámetro a la función API. Este puntero debe convertirse al tipo *TFarProc* obligatoriamente. A continuación se presenta un ejemplo de uso de esta función *callback* por la función API de Windows *EnumWindows*.

```
ValorRetorno:=EnumWindows(TFarProc(ActuaSobreVentana), UnLongInt);
```

La función API *EnumWindows* enumera todas las ventanas padre que están sobre la pantalla, pasando el *handle* de cada ventana a la función *callback*, en este caso *ActuaSobreVentana*, que

LOS MENSAJES

realizará algún tipo de operación sobre las ventanas que recibe. *EnumWindows* también devuelve un entero *UnLongInt* que se pasa al parámetro *UnValor* de la función *callback*. *EnumWindows* continua hasta que la última ventana padre es enumerada o la función *callback* devuelve cero.

La función *callback* debe retornar el mismo tipo de valor que la función API que la usa como parámetro.

Algunas funciones API de Windows que requieren el uso de funciones *callback* son las siguientes: *EnumChildWindows*, *EnumClipboardFormats*, *EnumFonts*, *EnumMetaFile*, *EnumObjects*, *EnumProps*, *EnumTaskWindows*, y *EnumWindows*. Estas funciones se utilizan para enumerar ciertos elementos del sistema Windows (ventanas hijas, formatos del portapapeles, fuentes, etc...).

La directiva del compilador `{ $\$$ κ +}` maneja las funciones *callback* automáticamente. Si no se utiliza la directiva `{ $\$$ κ +}`, las funciones *callback* deben pasarse a través de la función API de Windows *MakeProcInstance*, que devuelve una dirección capaz de ser utilizada por el entorno Windows.

15.9 LOS MENSAJES

Los mensajes son el medio que utiliza Windows para el manejo de eventos. El entorno Windows y sus aplicaciones interactúan entre sí y con el mundo exterior usando mensajes.

Los mensajes se generan desde las siguientes fuentes:

- Eventos generados por el usuario. Por ejemplo son los que resultan de pulsar el teclado o hacer *click* o mover el ratón.
- Mensajes devueltos por funciones API llamadas por el programa.
- Mensajes enviados explícitamente por una aplicación Windows
- Mensajes enviados por el entorno Windows a una determinada aplicación.
- Mensajes asociados con un intercambio de datos enlazados dinámicamente entre dos aplicaciones Windows (*DDE*, *Dynamic Data Exchange*) y (*OLE*, *Object Linking and Embedding*).

ESTRUCTURA DE LOS MENSAJES

Los diversos mensajes Windows se envían en forma de paquetes de información, con una estructura de datos, que contiene el identificador del mensaje transmitido y otra información adicional necesaria. La jerarquía de tipos objetos de *ObjectWindows* utiliza la estructura de tipo registro *TMessage* para transmitir los mensajes. La declaración del tipo *TMessage* es la siguiente:

```

TMessage = RECORD
  Receiver: HWnd; (* Handle de la ventana que recibe el mensaje *)
  Message: Word; (* Constante entera predefinida con el mensaje *)
  CASE integer OF
    0: (WParam:Word;
        LParam:Longint;
        Result:Longint);
    1: (WParamLo:Byte;
        WParamHi:Byte;
        LParamLo:Word;
        LParamHi:Word;
        ResultLo:Word;
        ResultHi:Word);
  END;

```

La estructura *TMessage* contiene cinco campos:

- El campo *Receiver* es un *handle* que identifica la ventana receptora del mensaje.
- El campo *Message* es una constante entera de tipo *word*, que identifica el mensaje que se envía. Estas constantes son las que aparecen en la ayuda Windows como mensajes. Habitualmente se escriben con letras mayúsculas, por ejemplo con el formato `wm_xxxxxxxx` para los mensajes de ventanas, así la constante `wm_Close` se envía cuando se cierra una ventana.
- El campo *WParam* es un valor entero de tipo *word* (2 bytes) que contiene información adicional al mensaje. La estructura de registro variante de *TMessage* permite acceder fácilmente a los bytes alto (*WParamHi*) y bajo (*WParamLo*) de *WParam*. Por ejemplo con el mensaje `wm_MButtonDblClk` el campo *WParam* contiene información sobre qué botón del ratón se ha pulsado (izquierdo, central o derecho) y si está pulsada simultáneamente la tecla **Ctrl** o la tecla **Shift**.
- El campo *LParam* es un valor entero de tipo *Longint* (2 word) que contiene información adicional al mensaje, así por ejemplo en el caso del mensaje `wm_MButtonDblClk` contiene las coordenadas *x* e *y* de donde se produjo el *click* del ratón. La estructura de registro variante de *TMessage* permite acceder fácilmente a las palabras alta (*LParamHi*) y baja (*LParamLo*) de *LParam*. En el ejemplo 15.17 se muestra como se obtienen las coordenadas de la posición del ratón.
- El campo *Result* se utiliza internamente por la jerarquía de tipos *ObjectWindows* para conocer el valor de salida del mensaje procesado.

El entorno Windows también suministra una estructura de datos de tipo registro denominada *TMsg*, para el manejo de mensajes directamente con funciones API de Windows.

TIPOS DE MENSAJES

La ayuda en línea de *Borland Pascal para Windows* ofrece la lista completa de todos los mensajes y las constantes que los definen. Los nemotécnicos de los mensajes son de la forma `wm_xxxxxxxx` para los referentes a ventanas, `cb_xxxxxxxx` para los referentes a las cajas combinadas (*combo box*), `bm_xxxxxxxx` para los referentes a los botones de radio y las cajas de comprobación,

LOS MENSAJES

`dm_XXXXXXXXXX` para los referentes al botón por defecto de las cajas de diálogo, `em_XXXXXXXXXX` para los referentes a las ventanas de edición, y `lb_XXXXXXXXXX` para los referentes a las cajas de listas (*list box*). A continuación se hace una clasificación de los distintos tipos de mensajes del entorno Windows.

- *Mensajes de gestión de ventanas.* Windows envía estos mensajes a una aplicación cuando el estado de una ventana se ve alterado. Las constantes que los definen son de la forma `wm_XXXXXXXXXX`. Ejemplos de estas constantes son: `wm_Activate` (se envía cuando se activa o desactiva una ventana), `wm_Close` (se envía cuando se cierra una ventana), `wm_Move` (se envía cuando se mueve la ventana), etc...
- *Mensajes de inicialización.* Windows envía este tipo de mensajes cuando una aplicación construye un menú o una caja de diálogo. Ejemplos de constantes que definen estos mensajes son: `wm_InitDialog` (se envía inmediatamente antes de que se muestre en pantalla una caja de diálogo), `wm_InitMenu` (se envía cuando se solicita la inicialización de un menú), etc...
- *Mensajes de entrada.* Windows envía estos mensajes como respuesta a una entrada a través del ratón, el teclado, barras de desplazamiento, o el temporizador programable del sistema. Ejemplos de constantes que definen estos mensajes son: `wm_Command` (se envía cuando se selecciona una opción de menú), `wm_LButtonDblClk` (se envía cuando se hace doble *click* en el botón izquierdo del ratón), `wm_LButtonDown` (se envía cuando se pulsa el botón izquierdo del ratón), etc...
- *Mensajes del sistema.* Windows envía estos mensajes a una aplicación cuando se accede al menú de control, a las barras de desplazamiento, o a los botones de dimensionamiento. Por ejemplo `wm_SysCommand` (se envía cuando se selecciona una opción del menú de control).
- *Mensajes del portapapeles.* Windows envía estos mensajes a una aplicación cuando otras aplicaciones intentan acceder al portapapeles de una ventana. Ejemplos de constantes que definen estos mensajes son: `wm_Cut` (copia lo seleccionado y lo envía al portapapeles), `wm_Copy` (copia lo seleccionado y lo envía al portapapeles), etc...
- *Mensajes de información del sistema.* Windows envía estos mensajes cuando se efectúa un cambio a nivel de sistema que afecta a otras aplicaciones Windows. Entre estos cambios están los que afectan al contenido del archivo *WIN.INI*, fecha y hora del sistema, paleta de colores, fuentes, etc... Por ejemplo `wm_TimeChange` (se envía cuando se realiza un cambio en el reloj del sistema).
- *Mensajes de manipulación de controles.* Las aplicaciones Windows envían estos mensajes a un objeto de tipo control como, por ejemplo, botones pulsables, cajas de listas, cajas combinadas o controles de edición. Los mensajes de controles llevarán

a cabo tareas específicas y devolverán un valor indicando el estado a la salida. Ejemplos de mensajes son: `bm_GetCheck` (devuelve el estado de un botón), `bm_SetCheck` (establece el estado de un botón), etc...

- *Mensajes de notificación de los controles.* Estos mensajes notifican a la ventana padre de un control qué acciones han ocurrido dentro del control.
- *Mensajes de notificación de barras de desplazamiento.* Las barras de desplazamiento envían estos mensajes cuando se hace *click* sobre ellas. Ejemplos de constantes que definen estos mensajes son: `WM_VSCROLL` (se envía cuando se hace un *click* de ratón en la barra de desplazamiento vertical), `WM_HSCROLL` (se envía cuando se hace un *click* de ratón en la barra de desplazamiento horizontal), etc...
- *Mensajes del área que no es de trabajo (non client area).* Windows envía estos mensajes para crear o actualizar el área que no es de trabajo de una ventana. Normalmente no son procesados por las aplicaciones, sino internamente por Windows. Son de la forma `WM_NCxxxxxxx`.
- *Mensajes del MDI (Multiple Document Interface).* Las ventanas marco de una ventana MDI envían estos a las ventanas hijas. Estos mensajes pueden ser activaciones, desactivaciones, creaciones, eliminaciones, reordenamientos, etc... Son de la forma `WM_MDIxxxxxxx`.
- *Mensajes DDE (Dynamic Data Exchange).* Son relativos al protocolo DDE de intercambio de datos entre aplicaciones. Son de la forma `WM_DDE_xxxxxxx`.
- *Mensajes OLE (Object Linking and Embedding).* Son relativos al protocolo OLE de intercambio de datos entre aplicaciones. Comienzan por `OLE_`.

RESPUESTA A LOS MENSAJES

La biblioteca *ObjectWindows* facilita la tarea de procesar mensajes, dado que realiza una gran cantidad de trabajo interno, incluyendo la toma y el control de reexpediciones de mensajes y realizando el procesamiento por defecto de los mensajes recibidos. En este apartado se explicará como se responde a los mensajes utilizando la biblioteca *ObjectWindows*, aunque también se puede realizar de otras formas manejando directamente los tratamientos de mensajes de las funciones API de Windows.

Para construir un método que responda a un mensaje, se declara un tipo objeto derivado, y se redefine el método que ofrece la respuesta deseada a cada tipo de mensaje. El método utilizado para responder al mensaje es un método dinámico cuyo índice se calcula en base a la constante del mensaje. Por ejemplo para responder a un mensaje que es una pulsación del botón izquierdo del ratón (`WM_LBUTTONDOWN`), se debe redefinir el método *WMLButtonDown* del tipo objeto *TWindow*, tal y como se presenta en el siguiente fragmento de código:

LOS MENSAJES

```

TYPE
  TMiVentana = OBJECT (TWindow)
  ...
  PROCEDURE WMLButtonDown(VAR Msg:TMessage);
    VIRTUAL wm_First+wm_LButtonDown;
  ...
END;
...
PROCEDURE TMiVentana.WMLButtonDown;
BEGIN

  (* Ejemplo de respuesta *)

  MessageBox(HWindow, '¡Hola a todos!', 'Saludo', mb_OK);
END;

```


Constante	Valor	Subrango	Tipo de mensajes
wm_First	\$0000	\$0000-\$03FF	Mensajes de ventanas
wm_User	\$0400	\$0400-\$6EFF	Mensajes de ventanas definidos por el programador
wm_Internal	\$7F00	\$7F00-\$7FFF	Reservado para uso interno por <i>ObjectWindows</i>
id_First	\$8000	\$8000-\$8EFF	Mensajes de identificadores de ventanas o controles definidos por el programador
id_Internal	\$8F00	\$8F00-\$8FFF	Reservado para notificaciones de control de uso interno por <i>ObjectWindows</i>
nf_First	\$9000	\$9000-\$9EFF	Mensajes de notificación definidos por el programador
nf_Internal	\$9F00	\$9F00-\$9FFF	Reservado para notificaciones a padres de uso interno por <i>ObjectWindows</i>
cm_First	\$A000	\$A000-\$FEFF	Mensajes de comandos definidos por el programador
cm_Internal	\$FF00	\$FF00-\$FFFF	Reservado para comandos de uso interno por <i>ObjectWindows</i>

Tabla 15.8 Tipos y rangos de mensajes

Obsérvese que el método de respuesta *WMLButtonDown* tiene un parámetro de tipo *TMessage*, que se pasa por dirección (*VAR*), y que permite una doble comunicación entre quien despacha el mensaje y el método que lo responde.

El índice del método dinámico se calcula sumando la constante que representa el mensaje (*wm_LButtonDown*) a otra constante (*wm_First*), donde *wm_First* representa un valor base para los mensajes de ventanas y *wm_LButtonDown* representa el desplazamiento.

ObjectWindows utiliza como índice de los métodos dinámicos un entero sin signo de tipo *word* (2 bytes). Es decir con esta forma de tratamiento de los mensajes con métodos dinámicos se pueden manejar hasta 65.536 mensajes diferentes. Los índices pueden tomar valores en un rango entre 0 (\$0000 en base 16) y 65.535 (\$FFFF en base 16). *ObjectWindows* establece una organización de este rango, usando unos valores base para definir distintos subrangos para cada tipo de mensajes. La tabla 15.8 muestra las constantes que define *ObjectWindows* como base a cada tipo de mensajes, así como el subrango de dicho tipo de mensajes.

En el fragmento de programa anterior se responde al mensaje saludando en una caja de mensajes. Se utiliza la función API de Windows *MessageBox*, cuyo primer parámetro es *HWindow*, que es un *handle* de la ventana que ha recibido el *click* de ratón. La primera cadena es el mensaje que saldrá en el centro del cuadro, mientras que la segunda es el título del cuadro. La constante *mb_OK* especifica que el cuadro debe incluir sólo un botón OK, habitualmente representado en castellano por el botón . Un programa completo que usa el fragmento de código anterior se presenta en el ejemplo 15.9.

Puede ocurrir que *ObjectWindows* envíe un mensaje a un objeto que no tiene definido un método específico de respuesta, entonces *ObjectWindows* pasa el registro *TMessage* a un método denominado *DefWndProc*, que maneja las acciones que se realizan por defecto con todos los mensajes a ventanas. El método *DefWndProc* está definido en el tipo objeto *TWindowsObject*, y en sus descendientes *TWindow*, *TMDIWindow*, y *TDialog*.

Ejemplo 15.17

Escribir un programa que recoja las coordenadas enviadas por el ratón en el campo *LParam* de la estructura *TMessage*.

Solución. Se construye un programa que al pulsar el botón izquierdo del ratón, se escriben las coordenadas de la posición del ratón. Se redefine el método *WMLButtonDown* del tipo objeto *TWindow* para escribir en el dispositivo de contexto las coordenadas de la posición del ratón, que se reciben en el campo *LParam* de *TMessage*. Se utiliza la función *wvsprintf* para escribir las coordenadas con un formato dentro de la cadena *S*; esta función es una versión en Pascal de la función *sprintf* del lenguaje C. El código del programa se presenta a continuación y una ejecución se muestra en la figura 15.30.

```
PROGRAM MensajesConCoordenadas;
USES Strings, WinTypes, WinProcs, OWindows;

TYPE
  PEjemploWindow = ^TEjemploWindow;
  TEjemploWindow = object(TWindow)
    UnDC: HDC;
    PROCEDURE WMLButtonDown(VAR Msg: TMessage);
      VIRTUAL wm_First + wm_LButtonDown;
  END;
```

LOS MENSAJES

```

TmiAplicacion = object(TApplication)
  PROCEDURE InitMainWindow; VIRTUAL;
END;

PROCEDURE TEjemploWindow.WMLButtonDown(VAR Msg: TMessage);
VAR
  S: ARRAY[0..9] OF Char;
BEGIN
  UnDC := GetDC(HWindow);
  wvsprintf(S, '%d,%d', Msg.LParam);
  TextOut(UnDC, Msg.LParamLo, Msg.LParamHi, S, StrLen(S));
  ReleaseDC(HWindow, UnDC);
END;

PROCEDURE TmiAplicacion.InitMainWindow;
BEGIN
  MainWindow := New(PEjemploWindow, Init(nil, 'Coordenadas del ratón'));
END;

VAR
  MiAp: TmiAplicacion;

BEGIN
  MiAp.Init('Prueba');
  MiAp.Run;
  MiAp.Done;
END.

```

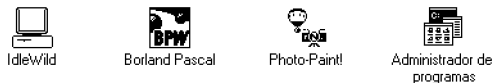
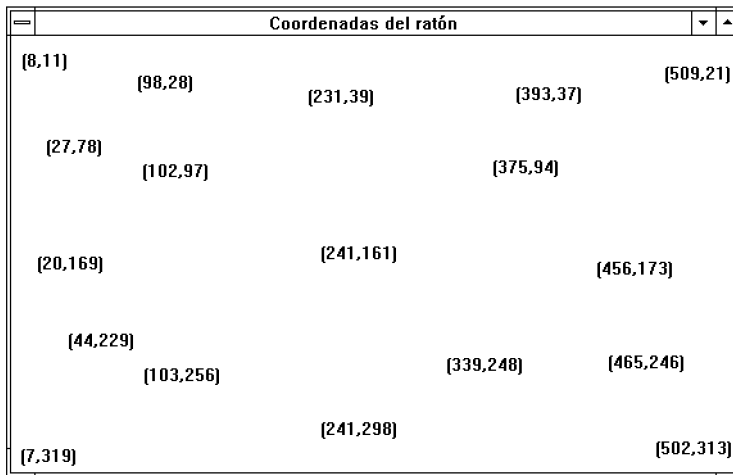


Figura 15.30 Ejecución del ejemplo 15.17

MENSAJES DE COMANDOS

ObjectWindows puede tratar los comandos de los menús y de las teclas aceleradoras como si fuesen mensajes.

Para procesar los mensajes de comandos *ObjectWindows* utiliza el método *WMCommand* del tipo objeto *TWindowsObject*, que habitualmente se hereda al definir un tipo descendiente. Pero en lugar de manejar los comandos por sí mismos, *WMCommand* maneja los mensajes de comandos basados en el identificador ID del menú o de la tecla aceleradora que genera el comando. Por ejemplo, si se define una opción en un menú con un identificador ID tal como *cm_HacerAlgo*, se puede escribir el siguiente método de respuesta a dicha opción de menú:

```

...
CONST
  cm_HacerAlgo=102;

TYPE
  TUnaVentana=OBJECT(TWindow)
  ...
  PROCEDURE CMHacerAlgo(VAR Msg:TMessage);
    VIRTUAL cm_First+cm_HacerAlgo;
  ...
END;

PROCEDURE TUnaVentana.CMHacerAlgo;
BEGIN
  (* Responder al comando HacerAlgo *)
END;
...

```

La constante *cm_First* es una constante de base para definir el rango de los mensajes de comandos (véase tabla 15.8). Las constantes de comandos pueden estar en el rango 0..24.319.

El método que procesa todos los comandos por defecto es *DefCommandProc*. Es equivalente al método *DefWndProc* para el proceso de mensajes de ventanas.

MENSAJES DE NOTIFICACIONES

Los comandos no proceden siempre de menús o teclas aceleradoras. Los controles de Windows envían comandos a sus ventanas padre, cuando se hace *click* con el ratón en ellos. Estos mensajes se denominan *mensajes de notificación*, y *ObjectWindows* los maneja de dos formas diferentes según el mensaje de notificación se envíe al objeto de control o a la ventana padre.

§ *Notificaciones de control*. Si el control tiene asociado un objeto de la jerarquía *ObjectWindows*, entonces *ObjectWindows* da al objeto una posibilidad de responder al primer comando.

§ *Notificaciones a padres*. Si el control no tiene asociado un objeto, o si el objeto de control no tiene definido un comando de respuesta, entonces la ventana padre tiene la posibilidad de responder.

LOS MENSAJES

• Notificaciones de control

Normalmente los controles no necesitan hacer nada especial para responder a las acciones de los usuarios, por defecto realizan el comportamiento esperado. Los mensajes de notificación permiten al programador hacer que el control haga algo extra o diferente de lo habitual.

Por ejemplo, supóngase que se desea que un botón pite cada vez que se le pincha. Debe definirse un método que responda a la notificación de que se ha pinchado el botón, la codificación sería de la siguiente forma:

```
...
CONST
  bn_pinchado = 301;

TYPE
  TPitaBoton = OBJECT (TButton)
    PROCEDURE BNPinchado (VAR Msg:TMessage);
      VIRTUAL nf_First+bn_pinchado;
    END;

PROCEDURE TPitaBoton.BNPinchado;
BEGIN
  MessageBeep(0);
END;
...
```

La constante *nf_First* es una constante de base para definir el rango de los mensajes de notificación de controles (véase tabla 15.8).

La función API *MessageBeep* toca un sonido (con formato *.wav) correspondiente a un nivel dado de alerta del sistema, definido por un entero de tipo *word*. Los sonidos de cada nivel de alerta están definidos en la sección [sounds] del fichero de inicialización de Windows.

• Notificaciones a padres

Los mensajes de notificación se envían al objeto ventana padre si un control no tiene asociado a él un objeto como interfaz, o si el objeto control no define una respuesta a un comando particular.

El mensaje de notificación a una ventana padre se basa en el ID del control, dado que las ventanas padre necesitan conocer cual de sus controles les envía la notificación. La forma de enviar un mensaje de notificación a un objeto ventana padre, puede verse en el siguiente fragmento de programa:

```
...
CONST
  id_MiControl = 401;

TYPE
  TMiVentana = OBJECT (TWindow)
    ...
    PROCEDURE IDMiControl (VAR Msg:TMessage);
      VIRTUAL id_First+id_MiControl;
    END;
```

```

PROCEDURE TmiVentana.IDMiControl;
BEGIN
  (* Responder a la notificación *)
  END;
...

```

La constante *id_First* es una constante de base para definir el rango de los mensajes de notificación de controles (véase tabla 15.8).

Es muy raro el caso en que Windows tiene un comportamiento por defecto como respuesta a controles particulares. Sin embargo, si se desea asegurar un comportamiento por defecto se puede utilizar el método *DefChildProc*, que trabaja como *DefWndProc*, pero maneja mensajes de notificación a ventanas padres.

• Notificaciones a controles y a padres

Es posible que en algunos casos sea necesario responder a las notificaciones tanto por los controles como por las ventanas padre. *ObjectWindows* permite hacerlo definiendo una notificación de control a la que se añade una llamada al método *DefNotificationProc*, que se encarga de que la ventana padre reciba el mensaje, como si el control no lo hubiera hecho anteriormente. Un ejemplo se puede ver en el siguiente fragmento de código, que es el mismo que se ha empleado para explicar la notificación de control, y al que se ha añadido la llamada al procedimiento *DefNotificationProc*.

```

...
CONST
  bn_pinchado = 301;

TYPE
  TPitaBoton = OBJECT (TButton)
    PROCEDURE BNPinchado (VAR Msg:TMessage);
      VIRTUAL nf_First+bn_pinchado;
    END;

PROCEDURE TPitaBoton.BNPinchado;
BEGIN
  MessageBeep(0);
  DefNotificationProc(Msg);
END;
...

```

MENSAJES DEFINIDOS POR EL PROGRAMADOR

ObjectWindows permite a los programadores definir sus propios mensajes. La constante *wm_User* está asociada con el número del primer mensaje (véase tabla 15.5). El resto de los mensajes se definen como desplazamientos de ese valor, por ejemplo se pueden definir:

```

CONST
  wm_MiPrimerMensaje = wm_User;
  wm_MiSegundoMensaje = wm_User+1;
  wm_MiTercerMensaje = wm_User+2;
...

```

LOS MENSAJES

Para responder a estos mensajes se hace como para cualquier otro, véase el siguiente fragmento de código:

```
TYPE
  TMiVentana =OBJECT(TWindow)
  ...
  PROCEDURE WMMiPrimerMensaje(VAR Msg:TMessage);
  VIRTUAL wm_First+wm_MiPrimerMensaje;
  PROCEDURE WMMiSegundoMensaje(VAR Msg:TMessage);
  VIRTUAL wm_First+wm_MiSegundoMensaje;
  PROCEDURE WMMiTercerMensaje(VAR Msg:TMessage);
  VIRTUAL wm_First+wm_MiTercerMensaje;
END;
```

ENVIAR MENSAJES

Windows permite que las aplicaciones se envíen mensajes a sí mismas, a otras aplicaciones o al propio entorno Windows. Las funciones API de Windows *SendMessage*, *PostMessage*, y *SendDlgItemMessage* se encargan de enviar los mensajes.

• La función *SendMessage*

La función *SendMessage* envía un mensaje a una ventana, y requiere que se le pase el *handle* de la ventana receptora, y el mensaje. Es necesario que la ventana receptora procese el mensaje. La función *SendMessage* está declarada en *WinProcs* de la siguiente forma:

```
FUNCTION SendMessage(Wnd:HWND; Msg, wParam: Word; lParam:LongInt):LongInt;
```

El parámetro *Wnd* es el *handle* de la ventana que recibe el mensaje. El parámetro *Msg* especifica el mensaje a enviar. Los parámetros *wParam* y *lParam* contienen información opcional que acompaña al mensaje.

La función *SendMessage* permite establecer comunicación con otras ventanas y controles (descendientes de *TControl*).

• La función *PostMessage*

La función *PostMessage* es similar a *SendMessage*, excepto que no tiene el sentido de urgencia de *SendMessage*. Con la función *PostMessage* el mensaje es colocado en la lista de espera de la cola de mensajes de la ventana receptora. Esa ventana receptora procesará el mensaje posteriormente, cuando le sea conveniente. La función booleana *PostMessage* está declarada en *WinProcs* de la siguiente forma:

```
FUNCTION PostMessage(Wnd:HWND; Msg, wParam: Word; lParam:LongInt):Bool;
```

El parámetro *Wnd* es el *handle* de la ventana que recibe el mensaje. El parámetro *Msg* especifica el mensaje a enviar. Los parámetros *wParam* y *lParam* contienen información opcional que acompaña al mensaje.

• La función *SendDlgItemMessage*

La función *SendDlgItemMessage* envía un mensaje a un componente particular de una caja de diálogo. La función *SendDlgItemMessage* está declarada en *WinProcs* de la siguiente forma:

```
FUNCTION SendDlgItemMessage(Dlg:HWND; IDDlgItem:Integer;
    Msg, wParam: Word;
    lParam:LongInt): LongInt;
```

El parámetro *Dlg* es el *handle* de la caja de diálogo que contiene el control que recibe el mensaje. El parámetro *IDDlgItem* es el número entero identificador ID del control de la caja de diálogo que recibirá el mensaje. El parámetro *Msg* especifica el mensaje a enviar. Los parámetros *wParam* y *lParam* contienen información opcional que acompaña al mensaje.

Ejemplo 15.18: construcción de un editor

Desarrollar un editor, que permita imprimir el fichero de texto que se está editando, así como los bloques de texto marcados. También deberá poder pasar bloques de texto de mayúsculas a minúsculas.

Solución. Se construye el programa por medio de un tipo objeto derivado de *TFileWindow*. Los recursos están en el fichero *editor.res*, ya mostrado en el epígrafe *menús* del apartado 15.7 de este capítulo. Un instante de la ejecución del programa se muestra en la figura 15.31.

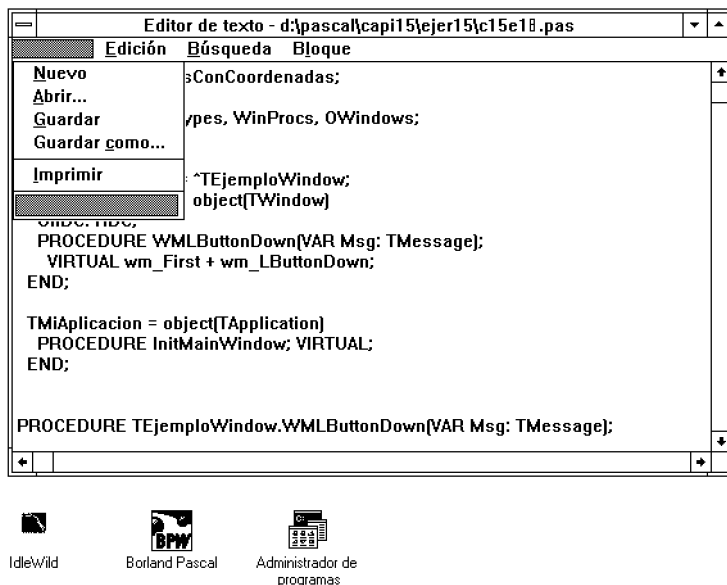


Figura 15.31 Ejecución del ejemplo 15.18

El listado del programa se presenta a continuación:

LOS MENSAJES

```
PROGRAM EditorDeTexto;

Uses OWindows, ODIALOGS, WinTypes, WinProcs,
    Strings, OStdWnds;

{$R editor.RES}

CONST
    nombreFichero_Menu = 'MenuEditor';
    cm_Imprimir = 301;
    cm_ImprimirBloque = 302;
    cm_BloqueMayusculas = 303;
    cm_BloqueMinusculas = 304;

TYPE
    TMiAplicacion = OBJECT(TApplication)
        PROCEDURE InitMainWindow; VIRTUAL;
        PROCEDURE InitInstance; VIRTUAL;
    END;

    PMiVentana = ^TMiVentana;
    TMiVentana = OBJECT(TFileWindow)
        UnDC_impresora : HDC;
        CONSTRUCTOR Init(UnPadre : PWindowsObject;
            UnTitulo : PChar;
            UnNombreFichero : PChar);
        PROCEDURE CMImprimir(VAR Msg : TMessage);
            VIRTUAL cm_First + cm_Imprimir;
        PROCEDURE CMImprimirBloque(VAR Msg : TMessage);
            VIRTUAL cm_First + cm_ImprimirBloque;
        PROCEDURE CMBloqueMayusculas(VAR Msg : TMessage);
            VIRTUAL cm_First + cm_BloqueMayusculas;
        PROCEDURE CMBloqueMinusculas(VAR Msg : TMessage);
            VIRTUAL cm_First + cm_BloqueMinusculas;
        PROCEDURE ImprimirPagina(hDCImpresora : HDC;
            texto : PChar);
        FUNCTION CogerDCImpresora : HDC;
            { Devuelve un puntero al siguiente token }
        FUNCTION StrTok(UnaCadena : PChar;
            unCaracter: CHAR) : PChar;
    END;

CONSTRUCTOR TMiVentana.Init(UnPadre : PWindowsObject;
    UnTitulo : PChar;
    UnNombreFichero : PChar);

BEGIN
    INHERITED Init(UnPadre, UnTitulo, UnNombreFichero);
    Attr.Menu := LoadMenu(hInstance, nombreFichero_Menu);
END;

PROCEDURE TMiVentana.CMImprimir(VAR Msg : TMessage);
VAR
    i : integer;
    MensajeImpresion : ARRAY[0..30] OF CHAR;
    hDCImpresora : HDC;
    textoLength : integer;
    textoStr : PChar;
BEGIN
    StrCopy(MensajeImpresion, 'Imprimiendo...');
    { Se inicializa el contador de texto }
    textoLength := 0;
    { Bucle para obtener el número total de líneas }
    FOR i := 0 TO Editor^.GetNumLines - 1 DO
        Inc(textoLength, Editor^.GetLineLength(i) + 2);
    GetMem(textoStr, textoLength * SizeOf(CHAR));
    Editor^.Gettext(textoStr, textoLength);
    { coge el handle de la impresora }
```

PROGRAMACION EN ENTORNO WINDOWS®

```

hDCImpresora := CogerDCImpresora;
{ Comienza la impresión }
IF hDCImpresora <> 0 THEN
BEGIN
  IF Escape(hDCImpresora, STARTDOC,
            SizeOf(MensajeImpresion)-1,
            @MensajeImpresion, NIL) > 0
  THEN
  BEGIN
    { Imprimir las páginas }
    ImprimirPagina(hDCImpresora, textoStr);
    IF Escape(hDCImpresora, NEWFRAME, 0, NIL, NIL) > 0
    THEN
      Escape(hDCImpresora, ENDDOC, 0, NIL, NIL);
  END;
END;
{ Elimina el handle de la impresora y la
  cadena dinámica }
DeleteDC(hDCImpresora);
FreeMem(textoStr, textoLength * SizeOf(CHAR));
END;

PROCEDURE TmiVentana.CMImprimirBloque(VAR Msg : TMessage);
VAR
  PosicionInicio,
  PosicionFinal,
  textoLength : INTEGER;
  MensajeImpresion : ARRAY[0..30] OF CHAR;
  hDCImpresora : HDC;
  textoStr : PChar;
BEGIN
  StrCopy(MensajeImpresion, 'Imprimiendo bloque...');
  { Toma la posición inicial y final del texto seleccionado }
  Editor^.GetSelection(PosicionInicio, PosicionFinal);
  { ¿ Hay texto seleccionado ? }
  IF PosicionInicio = PosicionFinal THEN Exit;
  { Calcula la longitud del texto seleccionado }
  textoLength := PosicionFinal - PosicionInicio + 1;
  { Crea una cadena dinámica para almacenar el texto seleccionado }
  GetMem(textoStr, (textoLength + 1) * SizeOf(CHAR));
  { Obtiene el texto seleccionado y lo almacena en textoStr }
  Editor^.GetSubtext(textoStr, PosicionInicio, PosicionFinal);
  { coge el handle de la impresora }
  hDCImpresora := CogerDCImpresora;
  { Comienza el proceso de impresión }
  IF hDCImpresora <> 0
  THEN
  BEGIN
    IF Escape(hDCImpresora, STARTDOC,
              SizeOf(MensajeImpresion)-1,
              @MensajeImpresion, NIL) > 0
    THEN
    BEGIN
      BEGIN
        ImprimirPagina(hDCImpresora, textoStr);
        IF Escape(hDCImpresora, NEWFRAME, 0, NIL, NIL) > 0
        THEN
          Escape(hDCImpresora, ENDDOC, 0, NIL, NIL);
      END;
    END;
  DeleteDC(hDCImpresora);
  FreeMem(textoStr, textoLength * SizeOf(CHAR));
END;

PROCEDURE TmiVentana.CMBloqueMayusculas(VAR Msg : TMessage);

```

LOS MENSAJES

```
VAR
    PosicionInicio,
    PosicionFinal,
    textoLength : integer;
    textoStr: PChar;
BEGIN
    Editor^.GetSelection(PosicionInicio, PosicionFinal);
    IF PosicionInicio = PosicionFinal THEN Exit;
    textoLength := PosicionFinal - PosicionInicio + 1;
    GetMem(textoStr, (textoLength + 1) * SizeOf(CHAR));
    Editor^.GetSubtext(textoStr, PosicionInicio, PosicionFinal);
    { Convierte los caracteres a mayúsculas }
    StrUpper(textoStr);
    { Inserta el nuevo y borra el viejo }
    Editor^.Insert(textoStr);
    FreeMem(textoStr, (textoLength + 1) * SizeOf(CHAR));
END;

PROCEDURE TmiVentana.CMBloqueMinusculas(VAR Msg : TMessage);
VAR
    PosicionInicio,
    PosicionFinal,
    textoLength : INTEGER;
    textoStr : PChar;
BEGIN
    Editor^.GetSelection(PosicionInicio, PosicionFinal);
    IF PosicionInicio = PosicionFinal THEN Exit;
    textoLength := PosicionFinal - PosicionInicio + 1;
    GetMem(textoStr, (textoLength + 1) * SizeOf(CHAR));
    Editor^.GetSubText(textoStr, PosicionInicio, PosicionFinal);
    StrLower(textoStr);
    Editor^.Insert(textoStr);
    FreeMem(textoStr, (textoLength + 1) * SizeOf(CHAR));
END;

FUNCTION TmiVentana.CogerDCImpresora : HDC;
CONST
    LongitudDescripcionImpresora = 80;
VAR
    DatosImpresora : ARRAY[0..LongitudDescripcionImpresora] OF CHAR;
    DevicePtr,
    DriverPtr,
    OutputPtr : PChar;
BEGIN
    { Coge la información de impresora del fichero WIN.INI
      en la sección [windows] y con la palabra 'device' }

    GetProfileString('windows', 'device', ',,,',
        DatosImpresora,
        LongitudDescripcionImpresora);
    DevicePtr := strtok(DatosImpresora, ',');
    DriverPtr := strtok(NIL, ',');
    OutputPtr := strtok(NIL, ',');
    IF (DevicePtr <> NIL) AND
        (DriverPtr <> NIL) AND
        (OutputPtr <> NIL)
    THEN
        CogerDCImpresora := CreateDC(DriverPtr, DevicePtr, OutputPtr, NIL)
    ELSE
        CogerDCImpresora := 0;
END;

PROCEDURE TmiVentana.ImprimirPagina(hDCImpresora : HDC;
    texto : PChar);
CONST
    altura_linea = 90;
    lineas_pag = 60;
```

PROGRAMACION EN ENTORNO WINDOWS®

```

VAR
  p1, p2 : PChar;
  NumeroLinea : integer;
BEGIN
  p1 := texto;
  p2 := texto;
  NumeroLinea := 0;
  WHILE p2^ <> #0 DO
    BEGIN
      IF p2^ <> #13 THEN Inc(p2)
      ELSE
        BEGIN
          textOut(hDCImpresora, 0, NumeroLinea * altura_linea, p1, p2 - p1);
          Inc(NumeroLinea);
          { ¿ página completa ? }
          IF NumeroLinea >= lineas_pag THEN
            BEGIN
              { Nueva página }
              Escape(hDCImpresora, NEWFRAME, 0, NIL, NIL);
              NumeroLinea := 0; { inicializar contador de líneas por pág. }
            END;
            { inicializar punteros }
            Inc(p2, 2);
            p1 := p2;
          END;
        END;
      { ¿Queda texto? }
      IF p1 < p2 THEN BEGIN
        IF NumeroLinea >= lineas_pag THEN BEGIN
          { nueva página }
          Escape(hDCImpresora, NEWFRAME, 0, NIL, NIL);
          NumeroLinea := 0;
        END;
        { imprimir la última línea }
        textOut(hDCImpresora, 0, NumeroLinea * altura_linea,
          p1, p2 - p1);
      END;
    END;
  END;

FUNCTION TMiVentana.StrTok(UnaCadena : PChar;
  unCaracter: char) : PChar;
(* Versión en Pascal de la función strtok de lenguaje C *)
CONST Result : PChar = NIL;
BEGIN
  IF UnaCadena = NIL THEN UnaCadena := Result;
  Result := StrScan(UnaCadena, unCaracter);
  IF Result <> NIL
  THEN
    BEGIN
      Result^ := #0;
      Result := @Result[1];
    END;
  StrTok := UnaCadena;
END;

PROCEDURE TMiAplicacion.InitMainWindow;
BEGIN
  MainWindow := New(PMiVentana, Init(NIL, Name, ''));
END;

PROCEDURE TMiAplicacion.InitInstance;
BEGIN
  INHERITED InitInstance;
  IF Status = 0 THEN
    HAccTable := LoadAccelerators(hInstance, 'FileCommands');
  END;
END;

```


EL PORTAPAPELES

```
VAR MiAplica : TMiAplicacion;  
  
BEGIN  
  MiAplica.Init('Editor de texto');  
  MiAplica.Run;  
  MiAplica.Done;  
END.
```

15.10 EL PORTAPAPELES

El portapapeles (*clipboard*) es un área de memoria del ordenador, que Windows utiliza para almacenar textos o imágenes procedentes de una aplicación Windows. Una vez almacenados los datos en el portapapeles, se pueden pasar a otra aplicación que trabaje bajo Windows.

Muchos programas manejan documentos o gráficos incluyen un menú *Edición* con las opciones *Cortar*, *Copiar*, y *Pegar*. Cuando el usuario selecciona *Cortar* o *Copiar*, el programa transfiere datos del programa al portapapeles. Cuando el usuario selecciona *Pegar*, el programa determina si el portapeles contiene datos en un formato utilizable por la aplicación, y si es así se transfieren los datos al programa.

El entorno Windows tiene un programa (CLIPBRD.EXE) que es un visor del portapapeles, y no las funciones del portapapeles. Estas están en el módulo USER de Windows.

FORMATOS DE DATOS DEL PORTAPAPELES

El portapapeles soporta varios formatos de datos. Los programas deben especificar los datos que se envían o reciben del portapapeles por medio de unas constantes predefinidas que comienzan con *cf_* (abreviatura de *clipboard format*):

- *cf_Text*. Cadena de caracteres ANSI terminada en #0, que contiene un retorno de carro y un salto de línea al final de cada línea.
- *cf_DsText*. Una representación de texto en un formato privado.
- *cf_Bitmap*. Una imagen con un mapa de bits compatible con Windows 2.0
- *cf_DsBitmap*. Una imagen con un mapa de bits en un formato privado.
- *cf_MetafilePict*. Una imagen con un *metafile*, e información adicional (altura y anchura de la imagen, etc...).
- *cf_DsMetafilePict*. Una imagen con un *metafile* en un formato privado.
- *cf_Silk*. Un bloque de memoria global que contiene datos en el formato *enlace simbólico* de Microsoft. Este formato es usado por ejemplo por la hoja de cálculo EXCEL de Microsoft.
- *cf_Dif*. Un bloque de memoria global que contiene datos en el formato de intercambio de datos DIF (*Data Interchange Format*).

- *cf_TIFF*. Un bloque de memoria global que contiene datos en el formato TIFF (*Tag Image File Format*).
- *cf_OEMText*. Un bloque de memoria global que contiene datos de texto que hacen uso del juego de caracteres OEM⁸⁰.
- *cf_DIB*. Una imagen de mapa de bits independiente del dispositivo. Esta es una imagen compatible con Windows 3.0.
- *cf_Palette*. Almacena una paleta de colores usada habitualmente en combinación con *cf_DIB*.
- *cf_Wave*. Almacena una onda de sonido.
- *cf_OwnerDisplay*. Almacena un formato privado de representación.
- *cf_RIFF*. Almacena en formato RIFF (*Resource Interchange File Format*)
- *cf_PenData*. Almacena en formato *Pen* cuando se usan las extensiones Windows adecuadas.

TRANSFERENCIA DE TEXTO AL PORTAPAPELES

Supongamos que se quiere transferir una cadena de n caracteres acabada en carácter nulo al portapapeles y que hay un puntero denominado *Pcadena*, que apunta a dicha cadena.

Los pasos a seguir son los siguientes:

- *Asignar un bloque de memoria global movable de tamaño igual al de la cadena*. Se realiza con la función API *GlobalAlloc*, especificando que es movable con la constante *GMEM_MOVEABLE*, y que la cadena tiene $n+1$ caracteres (el más uno es para contar el carácter #0). La función API *GlobalAlloc* devuelve un *handle* si se ha realizado la operación con éxito, y *NIL* si no ha podido asignar la memoria. La operación se realiza de la siguiente manera:

```
hMemoriaGlobal:=GlobalAlloc(GMEM_MOVEABLE, n+1);
```

- *Se bloquea la memoria global obtenida anteriormente para que no pueda ser reasignada*. Se hace utilizando la función API *GlobalLock*, que tiene como parámetro el *handle* obtenido en el paso anterior, y devuelve un puntero a dicho bloque de memoria. La operación se realiza de la siguiente manera:

```
PMemoriaGlobal:=GlobalLock(hMemoriaGlobal);
```

- *Se copia la cadena en el bloque de memoria global*. Se emplea el procedimiento *StrCopy* de la *unit String*.

⁸⁰ *OEM* son siglas de *Original Equipment Manufacturer*. En este caso se refiere al conjunto de caracteres *OEM* de DOS (ver anexo I).

EL PORTAPAPELES

```
StrCopy(PMemoriaGlobal, Pcadena);
```

- *Desbloquear la memoria global*, por si el sistema Windows desea moverla (recordar que se ha definido *movible*). Se emplea la función API *GlobalUnLock* cuyo parámetro es el *handle* obtenido en el primer paso.

```
GlobalUnLock(hMemoriaGlobal);
```

- *Abrir el portapapeles y vaciarlo*. Se utilizan las funciones API *OpenClipboard* y *EmptyClipboard*.

```
OpenClipboard(hWindow);  
EmptyClipboard;
```

- *Introducir la información en el portapapeles y cerrarlo*. Se utiliza la función API *SetClipboardData* cuyos parámetros son: una constante que indica el tipo de información a introducir (en este caso *cf_Text*); y el *handle* obtenido en el primer paso. Para cerrar el portapapeles se usa la función API *CloseClipboard*.

```
SetClipboardData(cf_Text, hMemoriaGlobal);  
CloseClipboard;
```

Además de seguir estos pasos hay que tener en cuenta las siguientes *reglas*:

- ✘ Sólo se puede llamar a *OpenClipboard* y *CloseClipboard* mientras se procesa el mismo mensaje, es decir la llamada a estas dos funciones API debe estar siempre dentro del mismo subprograma de ventana. No se puede dejar el control a otra ventana mientras está abierto el portapapeles.
- ✘ Siempre se debe asignar al portapapeles un bloque de memoria global fijo.
- ✘ Después de llamar a *SetClipboardData* no se puede utilizar el bloque de memoria global.

OBTENER TEXTO DEL PORTAPAPELES

Para obtener texto del portapapeles deben seguirse los pasos siguientes:

- *Comprobar si el portapapeles tiene texto*. Se utiliza la función API *IsClipboardAvailable* que devuelve *TRUE* si el portapapeles tiene datos en el formato especificado por la constante que se le pasó como parámetro. Esta función se puede utilizar sin abrir el portapapeles.

```
bDisponible:=IsClipboardAvailable(cf_Text);
```

- *Abrir el portapapeles*.

```
OpenClipboard(hWindow);
```

- *Obtener el bloque de memoria global.* Se utiliza la función API *GetClipboardData* que devuelve *ceros* (*falso*) si el portapapeles no contiene datos del tipo especificado por la constante pasada como parámetro. El *handle* devuelto por esta función pertenece al portapapeles, y no se puede modificar. Tan sólo es válido entre las llamadas a las funciones *GetClipboard* y *CloseClipboard*.

```
hMemPorta:=GetClipboardData(cf_Text);
```

- *Obtener un bloque de memoria global propio y movable.*

```
hMiMemoriaGlobal:=GlobalAlloc(gmem_Moveable, GlobalSize(hMemPorta));
```

- *Bloquear las dos zonas de memoria obtenidas, y obtener un puntero a ambas.*

```
lpMemPorta:=GlobalLock(hMemPorta);
lpMiMemoriaGlobal:=GlobalLock(hMiMemoriaGlobal);
```

- *Copiar desde el portapapeles a la zona de memoria propia.*

```
StrCopy(lpMiMemoriaGlobal,lpMemPorta);
```

- *Desbloquear las dos zonas de memoria.*

```
GlobalUnlock(hMemPorta);
GlobalUnlock(hMiMemoriaGlobal);
```

- *Cerrar el portapapeles.*

```
CloseClipboard;
```

15.11 LAS BIBLIOTECAS DE ENLACE DINAMICO (DLL)

Las bibliotecas o librerías de enlace dinámico (*Dynamic Link Libraries, DLL*⁸¹) son una vía de compartición de código y recursos entre varias aplicaciones. En un entorno multitarea como Windows, cualquier ahorro de espacio en el código resulta de interés; por eso se proporciona esta forma de ejecución de funciones.

Una DLL (*Dynamic Link Library*) es un módulo ejecutable, que contiene funciones que las aplicaciones Windows pueden utilizar. Son un concepto de gran importancia en Windows. En un primer acercamiento, son muy similares a las *units* de Turbo Pascal o las librerías de funciones, tales como las ya conocidas del lenguaje C o cualquier otro lenguaje. Pero existe una diferencia fundamental entre las *units* y las *DLLs*: en el caso de las *units*, la conexión se realiza en tiempo de enlazado (*link*), refiriéndonos con estos términos a una etapa posterior a la compilación en la que se unen diversos módulos para constituir un módulo ejecutable. El código de las funciones

81 Las DLLs pueden utilizarse tanto en Windows como en DOS en modo protegido (*protected mode*) con el compilador Borland Pascal, si se dispone como mínimo de un 80286 y 2 Mbytes de RAM. Con el modo protegido los microprocesadores 80286 pueden acceder a 16 Mbytes, y a 4Gbytes en los microprocesadores 80386 y superiores.

LAS BIBLIOTECAS DE ENLACE DINAMICO (DLL)

correspondientes se encuentra ahora embebido en el de la aplicación generada (ya que el enlazador o *linker* lo ha copiado, simplemente), y esto tiene las ventajas de que permite reutilizar código que proporcione servicios útiles a muchos programas. Este mecanismo se conoce como *enlace estático*.

En las DLLs esto no ocurre así. El programa ejecutable que utiliza llamadas a funciones de biblioteca contenidas en una DLL se genera sin repetir el código de esas funciones, sino que incluye únicamente las llamadas. Será en tiempo de ejecución cuando se cargue en memoria la DLL correspondiente y se resuelva la llamada. Esto se conoce como *enlace dinámico*.

Las bibliotecas o librerías de enlace estático pueden resultar ineficientes en un entorno multitarea. Debe pensarse que a pesar de las ventajas para el programador, que no necesita incluir los códigos fuente de las funciones reutilizadas, el código objeto de éstas estará repetido en todas las aplicaciones en ejecución que hagan uso de él; esto constituye un desperdicio de memoria. Si varias aplicaciones compartiesen realmente una sola copia de esos fragmentos de código, se ahorraría espacio; sin embargo, las librerías de enlace estático no permiten esto.

Windows aporta las DLL para remediar este problema. De hecho, una gran mayoría de las funciones del API se encuentran en algunas DLLs fundamentales: *KERNEL.EXE*, *USER.EXE* y *GDI.EXE*. Todas las aplicaciones Windows que utilicen sus funciones estarán compartiendo una sola copia de las mismas.

Además de la compartición de código, las DLL permiten la compartición de *recursos*, datos y componentes hardware. Generalmente, los drivers de dispositivo bajo Windows se encuentran implementados como DLLs.

Hay que decir que las funciones de una DLL deben ir acompañadas obligatoriamente de ciertas funciones genéricas de inicialización y de salida, con nombres y prototipos perfectamente definidos. Esta obligatoriedad depende del compilador concreto, ya que algunos de ellos generan el código objeto incluyendo versiones por defecto de estas funciones si el usuario no lo hace; en cualquier caso, los detalles sobre estas funciones obligatorias pueden consultarse en la bibliografía sobre programación en Windows. Para usar una DLL desde *Borland Pascal* no es obligatorio que la DLL esté escrita en *Borland Pascal*, así las DLLs son un mecanismo para integrar código en los proyectos realizados en distintos lenguajes.

Otra diferencia entre *units* y *DLLs* es que las *units* pueden exportar: tipos, constantes, datos y tipos objeto; mientras que las *DLLs* sólo pueden exportar procedimientos y funciones.

CONSTRUCCION DE UNA DLL

La estructura de una DLL en Pascal es similar a la de un programa en Pascal, pero comienza con la palabra reservada *LIBRARY* en vez de *PROGRAM*. LA estructura se muestra en el siguiente esquema:

PROGRAMACION EN ENTORNO WINDOWS®

```
LIBRARY nombreDLL;

USES ...

VAR ...

FUNCTION nombreF1(parámetros): tipoRetorno; EXPORT;
BEGIN
  {Cuerpo}
END;

PROCEDURE nombreP2(parámetros);EXPORT;
BEGIN
  {Cuerpo}
END;

FUNCTION nombreF3(parámetros): tipoRetorno; EXPORT;
BEGIN
  {Cuerpo}
END;

PROCEDURE nombreP4(parámetros);EXPORT;
BEGIN
  {Cuerpo}
END;

EXPORTS
  nombreF1 INDEX 1,
  nombreP2 INDEX 2,
  nombreF3 INDEX 3 NAME 'FUNCI3' RESIDENT,
  nombreP4 INDEX 4 NAME 'PROCEDI4' RESIDENT;

BEGIN
  {Sección de inicialización}
END.
```

La directiva *EXPORT* indica que los subprogramas que la llevan van a ser utilizados fuera de la DLL.

La sección *EXPORTS* enumera todos los subprogramas a exportar, siendo obligatorio asignarles a todos ellos un índice entero de rango entre 1 y 32767. Si se especifica la cláusula *INDEX* el subprograma se exporta con el número indicado, en caso contrario se le asigna el ordinal automáticamente. También pueden tener la cláusula *NAME*, que especifica una constante de cadena de caracteres que se va a utilizar como identificador del subprograma exportado. Si se añade la palabra reservada *RESIDENT*, se indica que la información queda en memoria cuando la DLL se carga, lo que permite reducir el tiempo de búsqueda del subprograma por los programas clientes de la DLL.

En la sección *EXPORTS* también pueden aparecer subprogramas incluidos con *USES* de una *unit*.

La compilación de una DLL genera un fichero con extensión *.DLL* en vez de *.EXE*.

LAS BIBLIOTECAS DE ENLACE DINAMICO (DLL)

USO DE DLLs

Tanto los programas como las *units* pueden utilizar DLLs. A continuación se muestra un esquema de código para utilizar una DLL desde un programa:

```
{SF+} (* Obliga a llamadas lejanas FAR *)  
  
PROGRAM PruebaDLL;  
  
USES ...  
  
VAR ...  
  
FUNCTION nombreF1(parámetros):tipoRetorno; FAR;  
EXTERNAL 'nombreDLL' INDEX 1;  
  
PROCEDURE nombreP2(parámetros); FAR;  
EXTERNAL 'nombreDLL' INDEX 2;  
  
FUNCTION nombreF3(parámetros):tipoRetorno; FAR;  
EXTERNAL 'nombreDLL' NAME 'FUNCI3';  
  
PROCEDURE nombreP4(parámetros); FAR;  
EXTERNAL 'nombreDLL' NAME 'PROCEDI4';  
  
BEGIN  
  (* Cuerpo del programa *)  
  { Se pueden usar los subprogramas de la DLL  
    como el resto de los subprogramas }  
END.
```

VENTAJAS DE USO DE LAS DLLs

Dadas las ventajas de las librerías de enlace dinámico, pueden resumirse algunas conclusiones útiles acerca de su utilización:

- Las DLLs resultan útiles en casos en los que no se desea realizar el enlace entre una aplicación y las funciones que utiliza hasta el momento de la ejecución, por razones de arquitectura y algorítmica.
- En Windows deberían utilizarse DLLs siempre que dispongamos de código susceptible de ser utilizado por varias aplicaciones que se ejecuten simultáneamente, con objeto de ahorrar el máximo espacio de memoria posible.
- De esta forma, es una ventaja que las DLLs permiten compartir no sólo código, sino también recursos entre distintas aplicaciones.
- Permiten adaptar con facilidad una aplicación a diferentes mercados o circunstancias (por ejemplo, todas las cuestiones susceptibles de ser influenciadas por distintos idiomas, tales como mensajes, pueden arrinconarse en DLLs fáciles de sustituir).
- Permiten al editor de diálogos utilizar cierta clase de controles definidos por el usuario, llamados *custom-controls*, que constituyen un uso bastante habitual de las DLLs.

- Permiten crear controladores de dispositivo o *drivers*. Son cierta clase de DLLs que no permanecen totalmente pasivas en espera de llamadas, sino que pueden ser activadas por interrupciones u otras vías. La mayoría de los controladores de dispositivo de Windows (*COMM.DRV*, *DISPLAY.DRV*, etc.) están implementados como DLLs.
- Constituyen una forma ideal de desarrollar aplicaciones complejas con modularidad; facilitan la división del trabajo, y cada grupo puede tener asignada una DLL distinta. Como el código de una DLL puede llamar sin problemas al código de otra, no existen esfuerzos de integración adicionales por el hecho de utilizar DLLs.
- El sistema se divide en subsistemas claramente delimitados, y además los interfaces entre ambos deberán estar rigurosamente definidos.
- Como cada DLL tiene un segmento de datos propio, los problemas de efectos laterales o contaminación de datos entre módulos están minimizados; esta encapsulación resulta muy aconsejable en sistemas grandes.

Ejemplo 15.19

Se escribe una DLL con la función seno hiperbólico.

```
LIBRARY FunHiper;
{$F+}
{$S-}
FUNCTION SenoH(x:real):real;EXPORT;
BEGIN
  senoH:=(exp(x)+exp(-x))/2;
END;

EXPORTS
  SenoH INDEX 1;
BEGIN
END.
```

Se utiliza la DLL anterior en un programa que muestra una tabla de la función seno hiperbólico en una ventana.

```
PROGRAM PruebaDLL;
{$F+}
USES WinCrt, Strings;

VAR valor:real;

FUNCTION SenH(x:real):real; FAR;
EXTERNAL 'FUNHIPER' INDEX 1;

BEGIN
  StrCopy(WindowTitle,'Prueba de DLL');
  WindowOrg.x:=100;
  WindowOrg.y:=100;
  WindowSize.x:=350;
  WindowSize.y:=250;
  InitWinCrt;
  valor:=0;
  Writeln('Valor    Seno Hiperbólico');
  REPEAT
    Writeln(valor:5:1, SenH(valor):15:10);
```


INTERCAMBIO DINAMICO DE DATOS (DDE)

```
valor:=valor+0.1;
UNTIL valor>1.1;
Write('Pulse una tecla para cerrar la ventana');
Readkey;
DoneWinCrt;
END.
```

15.12 INTERCAMBIO DINAMICO DE DATOS (DDE)

El intercambio dinámico de datos (*DDE, Dynamic Data Exchange*) es un mecanismo de comunicación entre procesos a los que da soporte Windows. El DDE está basado en el mismo sistema de mensajes en el que se basa Windows. Dos programas que están trabajando en Windows pueden mantener una *conversación* enviando mensajes el uno hacia el otro. Estos dos programas se denominan *servidor* y *cliente*. Un servidor DDE es un programa que puede ofrecer sus datos a otros programas. Un cliente DDE es un programa que obtiene datos del servidor.

Una conversación la inicia el programa cliente enviando el mensaje *wm_DDE_Initialize* a todos los programas que están funcionando en Windows. El servidor DDE que contiene estos datos puede responder a este mensaje.

Los mensajes utilizados en el DDE están en la *unit WinTypes*. Para realizar las comunicaciones se utilizan las funciones API *SendMessage* y *PostMessage*.

15.13 OBJETOS DE ENLACE E INCLUSION (OLE)

Los objetos de enlace e inclusión (*Object Linking and Embedding, OLE*) es una de las aportaciones principales de Windows 3.1 sobre Windows 3.0. Para su manejo se incorporan 64 funciones API.

OLE es un mecanismo que permite compartir datos desde aplicaciones Windows. Para explicar el funcionamiento de OLE, pongamos un ejemplo. Supongamos que se está manejando el procesador de textos *Write* incorporado por Windows, y se desea pegar una imagen de *Paintbrush*. Se dibuja la imagen con *Paintbrush* y se copia en el portapapeles. Posteriormente se pasa a *Write* y se pega en el documento desde el portapapeles. Ya tenemos un objeto OLE insertado. Se sigue escribiendo texto en *Write*, y en un momento dado deseamos volver a modificar la imagen, se hace doble *click* sobre la imagen y automáticamente pasamos a *Paintbrush* con la imagen lista para ser modificada. Cuando se cierra *Paintbrush* se vuelve a *Write*.

Al pegar un objeto OLE en otra aplicación se reciben los datos originales del objeto y una información sobre el programa que lo creó.

Windows 3.1 tiene una aplicación denominada *empaquetador de objetos (packager.exe)* que encapsula objetos OLE y no OLE en su propia clase de objetos OLE. El empaquetador es tanto un programa cliente OLE como servidor OLE. Es decir es un intermediario que ayuda a representar

objetos en los servidores OLE y de aplicaciones que no son servidores OLE. El empaquetador inserta las cosas bajo su propia clase de objeto OLE. Se puede utilizar el empaquetador para insertar sonidos o imágenes video dentro de un documento de *Write*.

OLE utiliza el DDE como mecanismo de transporte subyacente para mantener la conversación entre el servidor y el cliente. Las conversaciones DDE del OLE obedecen a un protocolo público, estándar y documentado en el SDK (*Kit de Desarrollo de Software*) de Windows 3.1. La escritura de programas con DDE y OLE es una de las partes más complejas de la programación Windows y supera los límites de la presente obra.

15.14 EJERCICIOS RESUELTOS

- 15.1** En la *unit Crt* del DOS existe el procedimiento *Delay*, que produce un retardo de un número de milisegundos pasado como parámetro. Sin embargo en la *unit WinCrt* no existe dicho procedimiento. Se desea crear dicho procedimiento para que funcione en el entorno Windows, teniendo en cuenta los problemas que conlleva su ejecución en un entorno multitarea sin privilegios como es el caso de Windows.

Solución

- Se construye en primer lugar un procedimiento *Delay* simple. Para implementar este procedimiento *Delay* se utiliza la función *GetTickCount* de la *unit WinProcs*, que devuelve el número de *tics* de reloj que se han producido desde que se comenzó la sesión de Windows. Por medio de un bucle vacío se construye el retardo.

```
PROCEDURE Delay(t:longInt);
(* T es el tiempo en milisegundos *)
VAR inicio:longInt;
BEGIN
  inicio:=GetTickCount; (* Función de WinProcs *)
  WHILE GetTickCount-inicio<=t DO; (* Bucle vacío *)
  END;
```

- Sin embargo el procedimiento anterior tiene un problema: *mientras se ejecuta este Delay nada de Windows trabaja*. Debemos modificar dicho procedimiento para que durante la ejecución del bucle vacío se permita la ejecución de otras tareas en Windows. Se escribe con tal fin un procedimiento denominado *DejaPasarOtros*, que tiene como misión dejar que se ejecuten en Windows otras tareas. Para realizar este procedimiento se usa la función *PeekMessage* para comprobar el estado de la cola de mensajes. Si hay mensajes se colocan en *mensaje*, para ser tratados posteriormente.

```
PROCEDURE DejaPasarOtros;
VAR mensaje:TMsg;
BEGIN
  WHILE PeekMessage(mensaje,0,0,0, pm_Remove) DO
  IF mensaje.Message =wm_Quit
  THEN Halt
  ELSE
  BEGIN
    TranslateMessage(mensaje);
```

EJERCICIOS RESUELTOS

```
        DispatchMessage(mensaje);
    END;
END;
```

• El siguiente paso es introducir este procedimiento dentro del bucle vacío de *Delay*, se construye así el nuevo procedimiento denominado *DelayCorrecto*. Un programa completo para comprobar el funcionamiento de los dos procedimientos *Delay* se presenta a continuación:

```
PROGRAM EjemploDelay(Output);
USES WinCrt, WinTypes, WinProcs;

PROCEDURE Delay(t:longInt);
(* T es el tiempo en milisegundos *)
VAR inicio:longInt;
BEGIN
    inicio:=GetTickCount; (* Función de WinProcs *)
    WHILE GetTickCount-inicio<=t DO; (* Bucle vacío *)
    END;

PROCEDURE DejaPasarOtros;
VAR mensaje:TMsg;
BEGIN
    WHILE PeekMessage(mensaje,0,0,0, pm_Remove) DO
        IF mensaje.Message =wm_Quit
            THEN Halt
            ELSE
                BEGIN
                    TranslateMessage(mensaje);
                    DispatchMessage(mensaje);
                END;
    END;

PROCEDURE DelayCorrecto(t:longInt);
(* T es el tiempo en milisegundos *)
VAR inicio:longInt;
BEGIN
    inicio:=GetTickCount; (* Función de WinProcs *)
    WHILE GetTickCount-inicio<=t DO DejaPasarOtros;
    END;

BEGIN
    Writeln('Cuando presione INTRO:');
    Writeln('Ocurrirá un retardo de 20 segundos');
    Writeln('No podrá mientras tanto cambiar a otra ventana o');
    Writeln('mover la actual');
    Readln;
    Writeln('Comienza el retardo...');
    Delay(20000);
    Writeln;
    Writeln('Ahora al pulsar INTRO, ocurrirá otro retardo pero');
    Writeln('si podrá cambiar de ventana o mover esta ventana');
    Readln;
    Writeln('Comienza un retardo de 30 segundos...');
    DelayCorrecto(30000);
    Writeln('Fin de la prueba de los retardos');
END.
```

- 15.2** Construir un programa que visualice ficheros de mapas de bits en una ventana, de tal forma que se pueda redimensionar y mover la ventana y los mapas de bits se adapten al tamaño de la ventana.

Solución

En primer lugar se crea un recurso denominado *GUILLEL.RES* en el taller de recursos, y se carga una imagen *.BMP (en este caso la foto de la figura 15.24). Se denomina a este mapa de bits *GUILLEL*. El recurso *GUILLE* en formato *.RC tiene la forma siguiente:

```
GUILLE1 BITMAP
BEGIN
    '42 4D 26 6B 00 00 00 00 00 00 36 04 00 00 28 00'
    '00 00 8E 00 00 00 B7 00 00 00 01 00 08 00 00 00'
    '00 00 F0 66 00 00 00 00 00 00 00 00 00 00 00 00'
    '00 00 00 00 00 00 00 00 00 00 00 00 00 BF 00 00 BF'
    '00 00 00 BF BF 00 BF 00 00 00 BF 00 BF 00 BF BF'
    '00 00 C0 C0 C0 00 C0 DC C0 00 F0 CA A6 00 80 00'
    '00 00 80 80 00 00 00 80 00 00 00 80 80 00 00 00'
    ...
    'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
    'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
    'FF FF FF FF B8 33'
END
```

El programa que presenta en pantalla el mapa de bits anterior es el siguiente:

```
PROGRAM MapaBits;

{$R GUILLEL}

USES Objects, WinTypes, WinProcs, OWindows;

TYPE
    TmiMapaBits = OBJECT(TApplication)
        PROCEDURE InitMainWindow; VIRTUAL;
    END;

    PmiVentana = ^TmiVentana;
    TmiVentana = OBJECT(TWindow)
        PROCEDURE Paint(PaintDC: HDC;
            VAR PaintInfo: TPaintStruct); VIRTUAL;
    END;

{*****}

PROCEDURE TmiMapaBits.InitMainWindow;
BEGIN
    MainWindow:= New(PmiVentana,
        Init(nil, 'Visualiza el mapa de bits: Guillel'));
END;

{*****}

PROCEDURE TmiVentana.Paint;
VAR
    rectangulo    : TRect;
    hdcMem        : HDC;
    hObjetoAntiguo, hMapaBits : THandle;
    desplazamiento, x_mb, y_mb, ancho_mb, alto_mb : integer;
```

EJERCICIOS PROPUESTOS

```
BEGIN
  GetClientRect(hWindow, rectangulo);
  desplazamiento := GetSystemMetrics(sm_CyMenu) +1;
  x_mb := rectangulo.left;
  y_mb := rectangulo.top;
  ancho_mb := rectangulo.Right - rectangulo.left;
  hdcMem := CreateCompatibleDC(PaintDC);
  hMapaBits := LoadBitmap(hInstance, 'GUILLE1');
  hObjetoAntiguo := SelectObject(hdcMem, hMapaBits);
  alto_mb := rectangulo.Bottom - rectangulo.Top;
  StretchBlt(PaintDC, x_mb, y_mb, ancho_mb, alto_mb, hdcMem,
            0,0,142,183, SRCCOPY);
  SelectObject(hdcMem, hObjetoAntiguo);
  DeleteObject(hdcMem);
  DeleteDC(hdcMem);
END;

{*****}

VAR
  Window: TmiMapaBits;

BEGIN
  Window.Init('MapaDeBits');
  Window.Run;
  Window.Done;
END.
```

15.15 EJERCICIOS PROPUESTOS

- 15.3 Modificar el ejemplo 15.18 para que los métodos de paso a mayúsculas y minúsculas pasen también las vocales acentuadas y la ñ.
- 15.4 Modificar el ejemplo 15.18 para que los textos de los cuadros de diálogo salgan en castellano. Puede hacerse de dos formas: a) diseñando nuevos cuadros de diálogo y cargándolos. b) Usando la función API de Windows 3.1 *GetOpenFileName* de la *unit commDlg*, que permite utilizar directamente los cuadros de diálogo que tiene el sistema Windows instalado (en el caso de que sea Windows en castellano saldrán en castellano).
- 15.5 Diseñar un programa que maneje fichas de alumnos almacenada como colecciones. Utilizar el tipo objeto *TCollection* y sus descendientes.
- 15.6 Modificar el ejercicio anterior comprobando todas las entradas introducidas por teclado usando el tipo *TValidator*.
- 15.7 Modificar el ejercicio anterior para que uno de los campos de datos de alumnos sea su fotografía introducida como un fichero de mapa de bits (*.BMP).
- 15.8 Utilizar la clase *TStream* para crear objetos persistentes, es decir que se pueden almacenar y recuperar del disco duro. Aplicarlo al ejercicio anterior.
- 15.9 Extender la biblioteca *ObjectWindows* creando un tipo objeto que maneje el portapapeles, denominado por ejemplo *TClipboard*, y que sea hijo de *TWindowsObject*.

- 15.10** Construir un programa de diseño gráfico utilizando como punto de partida el programa *graffiti.pas* que incorpora como curso de aprendizaje el manual de *ObjectWindows* dentro del producto *Borland Pascal*.

15.16 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Las especificaciones generalmente utilizadas en las GUI son las IBM SAA/CUA, cuya referencia es el documento publicado por IBM en 1989 y titulado *Common User Access: Advanced Interface Design Guide*. La adaptación de las especificaciones a Windows, está descrita en el tomo titulado *The Windows Interface: An Application Design Guide* del SDK (*Software Development Kit*) de Windows, publicado por *Microsoft*, y que es la guía de referencia de toda la programación Windows.

Existen relativamente pocos libros sobre programación Windows en Pascal, en castellano está publicada la obra de *Paul Perry* titulada *La biblia del Turbo Pascal para Windows* (*Anaya Multimedia*, 1993). Trata sobre una introducción general a la programación Windows, basándose más en las funciones API que en la biblioteca *ObjectWindows*. En lengua inglesa la obra *Borland Pascal 7 insider* de *P. Cilva* (Ed. Wiley, 1993) está dedicada por completo a la programación Windows con Pascal. También incorporan capítulos dedicados a Windows los libros: *Turbo Pascal 7, manual de referencia* (*S.K. O'Brien* y *S. Nameroff*, Ed McGraw-Hill, 1993), y *Borland Pascal Developer's Guide* (*E. Mitchell*, Ed. QUE, 1993). Sin embargo, por ahora son los manuales de *Borland Pascal*, su documentación en disco y su ayuda en línea la mejor fuente de información.

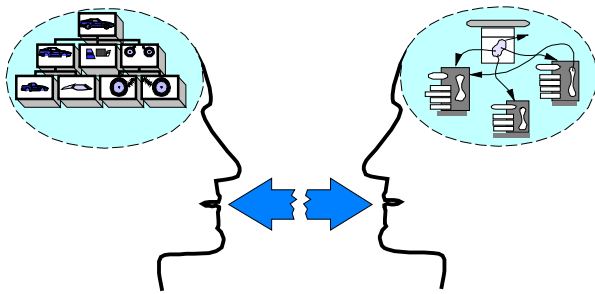
La mayor parte de la bibliografía sobre programación Windows está escrita para los lenguajes C y C++, sin embargo se puede utilizar para la programación en Pascal, dado que si dichas obras utilizan las funciones API de Windows, el *Borland Pascal* respeta el nombre de dichas funciones y las utiliza de forma similar. El libro de referencia es el *Charles Petzold* titulado *Programming Windows 3.1: Microsoft guide to writing applications for Windows 3.1* de la editorial *Microsoft Press* (1993). En castellano está publicado por la Editorial *Anaya* el mismo libro, pero para la versión de Windows 3.0. Este libro puede utilizarse como referencia de las principales funciones API de Windows.

La revistas *Microsoft System Journal* (*MSJ*), *Windows Tech Journal* (*WTJ*), y *Windows/Dos Developer's Journal* publican habitualmente artículos técnicos sobre la programación Windows. *WTJ* tiene una sección fija de Pascal. En castellano, la editorial *Anaya Multimedia* publica la revista *RMP* (*Revista Microsoft para Programadores*), con artículos traducidos de la *MSJ* estadounidense, y otros escritos en España sobre el entorno Windows.

Microsoft también suministra gran cantidad de información técnica sobre el entorno Windows y todos sus productos en el CD ROM titulado *Developer Network CD*, que actualiza cada cuatro meses.

Sobre *Windows NT* puede consultarse el libro titulado *El libro de Windows NT* de *H. Custer* (Ed. Anaya, 1993).

INTRODUCCION



CAPITULO 16

PROGRAMACION ESTRUCTURADA *VERSUS* PROGRAMACION ORIENTADA A OBJETOS

CONTENIDOS

- 16.1 Introducción
- 16.2 Programación estructurada (PE)
- 16.3 Programación orientada a objetos (POO)
- 16.4 Comparación
- 16.5 Conclusiones
- 16.6 Ampliaciones y notas bibliográficas

16.1 INTRODUCCION

Como epílogo al presente libro se presenta este capítulo en el que se compara la programación orientada a objetos (POO) y la programación estructurada (PE) con el fin de resumir los conceptos expuestos a lo largo de todos los capítulos del libro, y ayudar al lector a sacar sus propias conclusiones.

La PE es una metodología de programación que sigue el modelo de diseño descendente o de refinamientos sucesivos explicado en el capítulo 2.

El *Análisis Orientado a Objetos* (AOO) es una metodología de análisis que describe el problema a resolver y todo su entorno (sistema) en términos de objetos y clases⁸². El *Diseño Orientado a Objetos* (DOO) también describe el problema y las soluciones (sistema) en términos de objetos y clases, pero de una forma más detallada que el AOO, indicando ya las líneas precisas para llevar a cabo la implementación, que se realizará por medio de las técnicas de la POO.

En el capítulo 2 de este libro se dieron unas nociones generales de diseño de programas. Se pudo ver que la PE se basa en el diseño descendente, mientras que la POO utilizaba las técnicas de análisis y diseño orientado a objetos que son ascendentes. Sin embargo, aunque parezca paradójico, en ciertos pasajes de este libro se presentan las técnicas de POO como un refuerzo de la PE.

Se han escrito distintos artículos sobre el tema de comparar la PE y la POO. Así *Bertrand Meyer* opina que el diseño orientado a objetos y la POO cumplen los requerimientos de la PE y además van más allá (especialmente con el lenguaje *Eiffel*). A continuación se van a resumir las características de la PE y de la POO, para pasar a compararlas y extraer unas conclusiones generales.

16.2 PROGRAMACION ESTRUCTURADA (PE)

La PE es una metodología para diseñar y construir aplicaciones informáticas. Esta metodología o paradigma se puede *resumir* en los siguientes 7 puntos:

- PE1** *El problema es descompuesto en subproblemas. Esta descomposición continúa hasta que los subproblemas son atómicos y pueden ser resueltos de forma independiente. El acto de fragmentar un problema en componentes individuales reduce su complejidad⁸³.*
- PE2** *Los subproblemas están conectados por interfaces explícitas, y así pueden ser resueltos independientemente. Es decir no se utilizan para la comunicación variables globales, que podrían dar lugar a efectos laterales, y no dejan claro el interfaz.*
- PE3** *La estructura de los (sub)problemas y de los (sub)programas es casi idéntica.*
- PE4** *Hay una fuerte correspondencia entre la descripción del algoritmo, el programa fuente y su conducta dinámica (en tiempo de ejecución).*

82 Las clases son tipos objeto en Turbo Pascal.

83 La **complejidad** del software es según Booch una propiedad innata del software, debido a los siguientes aspectos: la complejidad del dominio del problema a resolver, la dificultad de coordinar el proceso de desarrollo, la flexibilidad que se puede alcanzar con el software y los problemas que se plantean cuando se quiere caracterizar el comportamiento de sistemas concretos.

PROGRAMACION ORIENTADA A OBJETOS (POO)

- PE5** *Como consecuencia de las reglas anteriores, el diseño descendente facilita la verificación de los programas.*
- PE6** *Las únicas estructuras de control de flujo permitidas son: la estructura secuencial, las estructuras alternativas y las estructuras repetitivas. No se permiten las bifurcaciones incondicionales.*
- PE7** *Se definen módulos, es decir partes del programa que pueden compilarse separadamente, pero que tienen conexiones con otros módulos. Además cada módulo distingue entre interfaz e implementación. También debe existir alguna definición de bloque, ámbito o alcance de los identificadores. En el caso particular del Turbo Pascal los módulos se construyen con units o DLL's, que a su vez están compuestas de subprogramas.*

Tal como se indicó anteriormente la PE es una metodología de programación, que no depende del lenguaje de programación utilizado, sino de la forma de diseñar y construir los programas. Sin embargo existen lenguajes de programación que por su estructura y características favorecen o refuerzan las técnicas de la PE. Así los lenguajes que permiten la construcción de módulos, y que dichos módulos definan claramente su interfaz con el exterior, facilitan algunos de los principios de la PE enumerados anteriormente. El lenguaje clásico para dar soporte a la PE, es el lenguaje *MODULA-2* (también creado por *N. Wirth*⁸⁴) y descendiente directo del lenguaje *Pascal*, que es en algunos aspectos muy similar, y que incluye el concepto de módulo en su definición estándar. En este libro los *módulos* se han construido con las *units* de Turbo Pascal. Además se ha utilizado Turbo Pascal por su gran calidad como compilador y entorno de desarrollo, además de su amplio uso comercial, y por que el uso de otros lenguajes implicaría dificultades para los lectores para encontrar compiladores y amplia documentación.

16.3 PROGRAMACION ORIENTADA A OBJETOS (POO)

La POO no es un nuevo paradigma de programación, el lenguaje *Simula 67* a finales de los sesenta y el lenguaje *Smalltalk* en los setenta comenzaron el desarrollo de este paradigma de programación, pero no tuvieron éxito (en términos comerciales), pero si desde el punto de vista de innovación tecnológica, de forma que en los ochenta y noventa todos los fabricantes promocionan sus productos indicando que soportan o están contruídos con técnicas de POO.

⁸⁴ Puede consultarse la obra de *N. Wirth* titulada ***Programming in Modula-2*** (Springer-Verlag, 4ª edición, 1988).

PROGRAMACION ESTRUCTURADA VERSUS PROGRAMACION ORIENTADA A OBJETOS

La realidad es que la *crisis del software*⁸⁵, la *complejidad* creciente de las aplicaciones informáticas y los *interfaces gráficos de usuario* han obligado a un uso intensivo de las técnicas de la POO.

Booch⁸⁶ define la **POO** como *un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase*⁸⁷, y cuyas clases son, todas ellas, miembros de una *jerarquía de clases unidas mediante relaciones de herencia*.

En la definición anterior hay tres partes importantes:

- Utiliza *objetos* como fundamento de todas sus construcciones lógicas.
- Cada *objeto* es una *instancia* de alguna *clase*.
- Las clases están relacionadas con otras clases por medio de relaciones de *herencia* (*jerarquía de clases*).

Un programa puede parecer que está construido utilizando las técnicas de la POO, pero si faltan algunos de los tres puntos anteriores, se puede decir que no es un programa orientado a objetos. Así en los capítulos que van del 8 al 12 se usó frecuentemente el concepto de *tipos abstractos de datos* (TAD), implementados como *units* de Turbo Pascal. A este tipo de programación se le denomina *programación con tipos abstractos de datos* (PTAD), y no es POO dado que no utiliza la herencia ni el polimorfismo, aunque el uso de *units* puede verse como una forma de encapsulación.

Booch también define *Análisis Orientado a Objetos (AOO)* como *un método de análisis que examina los requerimientos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema*.

Booch define además *Diseño Orientado a Objetos (DOO)* como *un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico así como los modelos estático y dinámico del sistema que se diseña*. Los pasos fundamentales del DOO ya se estudiaron en el capítulo 2 en el apartado 2.4 *Desarrollo de la solución*.

⁸⁵ Se entiende por **crisis del software** a los sucesivos fracasos de las distintas metodologías para dominar la complejidad del software, lo que implica el retraso de los proyectos de software, las desviaciones por exceso de los presupuestos fijados, y la existencia de deficiencias respecto a los requerimientos del cliente.

⁸⁶ G. Booch. *Object-Oriented Analysis and Design with Applications* (Ed. Benjamin Cummings, 2ª edición 1994).

⁸⁷ Tipo objeto en Turbo Pascal.

PROGRAMACION ORIENTADA A OBJETOS (POO)

La POO no es sólo un paradigma de programación, sino que junto con el AOO y el DOO se puede ver también como un conjunto de conceptos de *Ingeniería del Software*⁸⁸ para producir mejores aplicaciones. Al igual que *Booch, Cox, Meyer* y otros autores argumentamos que los conceptos de orientación a objetos producirán aplicaciones que serán superiores a los programas convencionales (que usan PE⁸⁹) en términos de corrección, robustez, ampliación, reusabilidad, extensibilidad y compatibilidad.

Como características básicas de la POO para comparar con la PE utilizaremos: *encapsulación* (incluye ocultación de la información), *herencia*, y *polimorfismo* (definido también como paso de mensajes).

- **Encapsulación**

En la POO los datos van unidos a los subprogramas que manipulan estos datos. No se puede acceder a los datos si no es a través de los subprogramas. Los procedimientos agrupados a estos datos se llaman *métodos*, todos los métodos unidos se llaman *interface*. La definición de los métodos y de los datos unidos son las *clases* o *tipos objeto* en Turbo Pascal. Las instancias de de cada clase se denominan *objetos*.

- **Herencia**

La herencia es una técnica que permite la reutilización de un proceder ya definido en *clases* en la definición de nuevas clases. La herencia puede expresar relaciones entre procederes tales como clasificación, especialización, generalización, aproximación y evolución.

Junto a estas ventajas conceptuales, la herencia ayuda a evitar la duplicación de código y permitir codificar muy rápido nuevas características. Por lo tanto también es una herramienta para mejorar la Ingeniería del Software.

- **Polimorfismo (o paso de mensajes)**

Cuando un mensaje es enviado a un objeto implica una demanda para realizar alguna acción. Es responsabilidad del receptor la forma de reaccionar. De este modo, es posible que diferentes objetos reaccionen de forma distinta después de recibir el mismo mensaje. En Turbo Pascal pasar un mensaje es aplicar un método sobre un objeto.

⁸⁸ *Ingeniería del Software* es el término acuñado para describir la actividad de la construcción de grandes sistemas de software. Actualmente es una de las disciplinas de la Informática.

⁸⁹ J. Stein afirma que la PE parece derrumbarse cuando las aplicaciones superan las 100.000 líneas de código (ver el artículo: *Object-Oriented Programming and Database Design*, *Dr. Dobb's Journal*, Marzo 1988, nº 137).

PROGRAMACION ESTRUCTURADA VERSUS PROGRAMACION ORIENTADA A OBJETOS

Como éste es un concepto de alto nivel que influye en el diseño de la jerarquía de objetos, en el bajo nivel de implementación del lenguaje hay una diferencia muy importante entre las llamadas de los subprogramas convencionales (*estáticos*) y las llamadas a los métodos *virtuales*. Los métodos estáticos deciden en tiempo de compilación el tipo del objeto al que se aplican. Sin embargo los métodos virtuales deciden en tiempo de ejecución el objeto de la jerarquía sobre el que se aplican.

Algunos de los lenguajes de programación tradicionales han sido ampliados para soportar las características de la POO, dando lugar a los denominados *lenguajes orientados a objetos híbridos* (C da lugar a C++, Pascal a Object Pascal, LISP a CLOS, etc...). Es preciso indicar que el soporte del lenguaje C++ a la POO es mucho más rico y potente que el dado por *Object Pascal* o *Turbo Pascal*. Pero el inconveniente del lenguaje C++ es su complejidad para un primer curso de programación y en algunos casos su oscuridad sintáctica. Además las libertades que permiten los lenguajes C y C++ al programador pueden ser manejadas incorrectamente por un programador *inexperto*, pudiendo crear gran confusión en un curso de introducción a la programación, en el que se pretende inculcar al alumno entre otras las técnicas de la PE. Sin embargo el lector que ha asimilado los conceptos presentados en este libro, puede enfrentarse con grandes posibilidades de éxito con la programación de cualquier lenguaje de programación *imperativo*⁹⁰, u orientado a objetos *híbrido* tal como C++.

Además de los lenguajes orientados a objetos híbridos están los *lenguajes orientados a objetos puros*, en los cuales toda la estructura del lenguaje se basa en objetos (*Smalltalk, Eiffel,...*). Estos lenguajes son muy apropiados para dar un curso completo de POO, con el añadido de que al ser puros obligan al alumno a realizar todo con objetos, consiguiendo un dominio amplio de las técnicas de AOO, DOO y POO, que serían susceptibles de llevar a entornos híbridos. Sin embargo la solución adoptada de unificar la PE y la POO en un sólo curso, puede que sea más pragmática.

Bertrand Meyer, autor del lenguaje *Eiffel*, define en su libro *Object-oriented software construction* (Prentice-Hall, 1988) las siete características de un lenguaje orientado a objetos puro:

- *Estructura modular basada en objetos.*
- *Abstracción de datos.* Los objetos deben describirse como implementaciones de tipos abstractos de datos.

⁹⁰ Vease el epígrafe *Tipos de lenguajes de programación* del capítulo 1.

COMPARACION

- *Gestión de memoria automática.* Los objetos dinámicos no utilizados deberán ser eliminados automáticamente por algún mecanismo oculto⁹¹, sin la intervención del programador.
- *Clases.* Las clases son una combinación de módulos y declaración de tipos. Todos los tipos de datos son clases, a excepción de los tipos simples.
- *Herencia.* Una clase puede definirse como una extensión o una restricción de otra.
- *Polimorfismo y enlace dinámico*⁹². Los objetos pueden declararse de más de una clase, siempre dentro de una jerarquía, y su comportamiento puede ser diferente en cada clase. El comportamiento específico se decide en tiempo de ejecución.
- *Herencia múltiple y repetida.* Una clase puede heredar de más de una clase y ser padre de la misma clase más de una vez.

16.4 COMPARACION

La comparación se realizará examinando las características de la POO, y ver su compatibilidad con la PE. Las tres principales características del paradigma orientado a objetos y las siete características de los PE forman una tabla con 21 campos, que se expondrá como conclusión. En los próximos tres subapartados trataremos de rellenar las columnas que tratan la encapsulación, la herencia y el polimorfismo.

ENCAPSULACION

La encapsulación no es una invención nueva de la POO dentro de la metodología de programación. PE1 es una cuestión de diseño. La encapsulación es una técnica de implementación que refuerza los interfaces (una consecuencia de PE1 y PE2), pero las dos no son tratadas directamente. PE2 está todavía en el nivel de diseño, pero debido a PE3 se pide un interface que

⁹¹ Habitualmente este mecanismo es el recolector de basura (*garbage collection*), que se encarga en tiempo de ejecución de eliminar las variables y los objetos dinámicos que han dejado de utilizarse. En la liberación de elementos de estructuras dinámicas los programadores pueden dejar zonas de las estructuras dinámicas sin referencia al liberar indebidamente las variables referenciadas de ciertos punteros, produciéndose una corrupción del sistema en tiempo de ejecución. Los lenguajes Pascal y C++ no tienen este mecanismo dado que la liberación, se deja como responsabilidad directa del programador (en Pascal por medio de la función *Dispose*, en Turbo Pascal también se puede usar *FreeMem*). Sin embargo los lenguajes Smalltalk y Eiffel si tienen recolector de basura, descargándose al programador de esta responsabilidad. Pero la comodidad de desarrollo de los programadores puede pagarse en tiempo de ejecución por los usuarios, dado que cuando arranca el recolector de basura en tiempo de ejecución, se produce una ralentización de las aplicaciones, aunque cada día se mejoran las técnicas de recolección de basura.

⁹² *Dynamic binding*

sea tratado directamente por la implementación. Por lo tanto está claro que la encapsulación cubre por completo PE2. Además, si la fase de DOO se realiza mediante una red de objetos que se intercomunican, esto puede implementarse de acuerdo al punto PE3.

También existen algunas dificultades de PE2 y PE3 con la encapsulación. Supóngase que hay un coche orientado a objetos, que se estropea. ¿Es el coche el que debe repararse a sí mismo, o lo debe de hacer un mecánico? Este es el punto donde entra en juego PE3. Si queremos tener un modelo del mundo real, el mecánico debe de hacerlo, pero lo tiene prohibido por el concepto de encapsulación. Hay una forma en C++ que esto es posible, se llama: *funciones amigas*⁹³, a las cuales se les permite el acceso a los datos de los objetos de diferentes clases. Pero repetimos, esto viola la noción de encapsulación. Creemos que un mecanismo similar es necesario en todos los lenguajes orientados a objetos, si queremos cubrir por completo PE3. Este concepto de modificación es aceptable ya que una clase debe declarar a sus amigos de una forma explícita. En nuestro mundo no existen objetos activos independientes, en cambio hay muchos objetos que son pasivamente manipulados por otros objetos.

La encapsulación es compatible con PE4 si nos permitimos interpretar PE4 de una forma más liberal. La PE usa una descomposición algorítmica en la fase de diseño, la cual produce una descripción del algoritmo, el punto examinado en PE4. El DOO utiliza un tipo de descomposición orientado a objetos, que produce otro tipo de descripción. Si esta diferencia es aceptada, podemos concluir que los objetos encapsulados reflejan realmente el diseño de agentes independientes al nivel de implementación.

No se quiere tratar PE5 a un nivel formal. Se pueden utilizar razonamientos por enumeración, inducción matemática y abstracción como tres tipos de ayudas para realizar la verificación de programas. Sin embargo la abstracción y una buena comprensión de los programas se puede proponer como alternativa a técnicas más formales de verificación. La encapsulación es una de las técnicas mas fuertes para realizar la abstracción y una buena comprensión de los programas.

La encapsulación refuerza los interfaces, los cuales son necesarios para abstraerlos de la implementación. De una forma más general, los interfaces deben ser respetados para que no ocurran efectos laterales, haciendo los programas más fáciles de entender.

PE6 ha sido principalmente formulado para evitar el uso deliberado de "gotos". Incluye por supuesto llamadas a procedimientos, los cuales son un tipo de secuencia. Ya que la encapsulación es un mecanismo sintáctico para restringir el alcance de los identificadores (de datos), no es tratado en PE6, pero sí de una forma fuerte por PE7. En PE7 se dice que los identificadores deben ser locales siempre que sea posible y sólo globales cuando sea necesario. La encapsulación orientada a objetos tiene el mismo origen pero es reducido sólo a los datos (identificadores). Las clases (tipos) son globales, los métodos públicos también.

93 No disponibles en Turbo Pascal

COMPARACION

HERENCIA

Como se ha dicho antes, la herencia se puede ver como una relación conceptual y también como una técnica de programación. PE1 trata la primera interpretación, la cual es una cuestión de diseño y no de implementación. *Booch* clasifica la descomposición de un problema en descomposición algorítmica (diseño descendente) y descomposición orientada a objetos, y afirma que PE utiliza descomposición algorítmica. Mientras que éste es posiblemente cierto, PE1 es suficientemente general para permitir las dos interpretaciones, así como otros tipos de descomposiciones.

La *descomposición orientada a objetos* produce una partición plana de todos los objetos concernientes. Después de esto el proceso de descubrir la herencia comienza, produciendo un grafo de clases de objetos conectados por especialización y/o generalización. De ésta forma los dos métodos (diseño descendente y descomposición orientada a objetos) acaban con una estructura jerárquica. Pero con una descomposición algorítmica la raíz representa el problema original, mientras que con la descomposición orientada a objetos el problema es modelado por la interacción de las instancias de los nodos del grafo.

Se han tratado los interfaces (PE2) en la sección de encapsulación. La herencia también trata los interfaces, ya que la herencia establece un nuevo tipo de interface, no conocido en la programación convencional. En la POO los objetos son clasificados en proveedores y consumidores: *la programación orientada a objetos es una manera para encapsular funcionalidad por parte de los suministradores de código y enviárselo a los consumidores.*

Los interfaces orientados a objetos son a menudo tratados como interfaces externos, por ejemplo suministrando un interface que los consumidores pueden usar. Hay también un interfaz desde las clases padres hacia las clases heredadas, que se llamará por herencia al interfaz. Si se siguen los requerimientos de la encapsulación será posible la reimplementación de una clase, mientras se mantenga la herencia del interfaz, y todos los descendientes se comporten correctamente.

La herencia maneja PE3 de una forma clara. Si se ha detectado especialización y generalización en el proceso de diseño orientado a objetos, se usará la herencia para implementar las correspondientes clases. Desde que se utiliza un diseño adecuado, la solución se desarrolla de forma natural.

PE4 y PE5 también se tratan en la herencia, pero se discutirán en la siguiente sección, ya que estos aspectos están fuertemente relacionados con el polimorfismo.

PE6 no es tratado de ninguna manera por la herencia. Esta es una técnica de programación para reutilización de código, y no para el control de flujo de programas.

PE7 pide alguna estructura de bloque, la cual es implementada por los lenguajes convencionales principalmente por subprogramas y módulos. La POO utiliza las clases como principal mecanismo de estructuración. La mayor parte de los lenguajes orientados a objetos permiten construir clases dentro de módulos lo que concuerda con PE7.

POLIMORFISMO

El polimorfismo no está relacionado con PE1. PE2 tampoco está relacionado. Los métodos virtuales pueden verse como subprogramas genéricos que resuelven subproblemas, lo que concuerda con PE3.

Al definir métodos virtuales, se permite a diferentes clases redefinir distintos métodos con el mismo nombre. En tiempo de ejecución (enlace dinámico) se decide qué método se aplica en función del objeto al que se le aplica el método. En comparación con la programación tradicional creemos que un cambio en el tiempo de enlace no va contra PE4. Además, el tipo de sobrecarga del polimorfismo es también usado por los lenguajes convencionales, por ejemplo en Turbo Pascal el operador + realiza distintas operaciones: la suma binaria para enteros y reales, unión de conjuntos, y concatenación de cadenas⁹⁴.

Los problemas pueden aparecer con PE5, ya que con el polimorfismo y enlace dinámico es posible:

- Personalizar librerías o bibliotecas.
- Hacer llamadas a nuevo código desde código viejo.

La primera situación ocurre cuando se declara una clase de una biblioteca como padre de una clase que estamos definiendo. Al verificar nuestra nueva clase la única ayuda que se tiene es una caja negra que nos dice que el antecesor es correcto, lo que no es suficiente para el trabajo que se está realizando. Pero habitualmente se utilizan bibliotecas de clases verificadas, no habiendo diferencias con respecto a una llamada a un subprograma, que se toma de una biblioteca que ya está verificada.

La segunda situación sucede cuando se amplía un software que ya existe. No es suficiente con examinar el nuevo código, hay que revisar una vez más el código antiguo. Por eso la mayor parte de los fabricantes de bibliotecas de clases ofrecen siempre de alguna forma el código fuente. Sin embargo no es posible garantizar el correcto uso del código nuevo desde el código viejo.

Ya se ha generalizado PE6 con llamadas a métodos y subprogramas en el caso de la herencia.

El polimorfismo no está relacionado con PE7.

16.5 CONCLUSIONES

Los resultados de la comparación se resumen en la tabla 16.1. Las entradas horizontales de la tabla son las siete características de la PE, mientras que en las entradas verticales se colocan las

⁹⁴ En lenguajes como C++ también se pueden sobrecargar los operadores y las funciones, es decir se pueden redefinir para que hagan nuevas tareas en función de los parámetros del operador.

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

tres características de la POO. En la intersección de ambas entradas se coloca una de las tres afirmaciones siguientes: *no relacionadas*, *concuerdan*, *ad* (abreviatura de *algunas discrepancias*), y *contrarias*.

En líneas generales se puede indicar que las técnicas de la POO concuerdan con la PE, aunque en algunos casos hay discrepancias y en otros las técnicas de la POO no están relacionadas con la PE. La principal contrariedad está en la reutilización de código que puede dificultar la verificación de los programas según la regla PE5.

	Encapsulación	Herencia	Polimorfismo
PE1	<i>no relacionadas</i>	<i>concuerdan</i>	<i>no relacionadas</i>
PE2	<i>concuerdan/ad</i>	<i>concuerdan/ad</i>	<i>no relacionadas</i>
PE3	<i>concuerdan/ad</i>	<i>concuerdan</i>	<i>concuerdan</i>
PE4	<i>concuerdan</i>	<i>concuerdan</i>	<i>concuerdan</i>
PE5	<i>concuerdan</i>	<i>contrarios/ad</i>	<i>contrarios/ad</i>
PE6	<i>no relacionadas</i>	<i>no relacionadas</i>	<i>concuerdan</i>
PE7	<i>concuerdan/ad</i>	<i>concuerdan</i>	<i>no relacionadas</i>

Tabla 16.1 Comparación de la PE y la POO

16.6 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Muchas de las ideas de este capítulo están extraídas del artículo de *B. Muller* titulado *Is Object-Oriented Programming Structured Programming?* (ACM SIGPLAN NOTICES, Vol. 8, No. 9, 1993), aunque no concuerda todo lo expresado por *Muller* con lo expuesto en este capítulo. También puede consultarse la obra de *B. Meyer* titulada *Object-Oriented Software Construction* (Prentice-Hall, 1988) especialmente los cuatro primeros capítulos donde indica la transición hacia la POO.

Sobre análisis y diseño orientado a objetos la obra más clásica es la *G. Booch* titulada *Object-Oriented Analysis and Design with Applications* (Benjamin Cummmings, 2ª Ed. 1994). También puede consultarse la obra de *B. J. Cox* y *A. J. Novobilski* titulada *Programación orientada a objetos, un enfoque evolutivo* (Ed. Addison-Wesley/Díaz de Santos, 2ª edición 1993, traducción del original de 1991).

Sobre PE pueden consultarse la obras: *O.J. Dahl*, *E.W. Dijkstra*, y *C.A.R Hoare* titulada *Structured Programming* (Academic Press, 1972) y *Systematic Programming* de *N. Wirth* (Prentice-Hall, 1973).

PROGRAMACION ESTRUCTURADA *VERSUS* PROGRAMACION ORIENTADA A OBJETOS

ASCII



ANEXO I

ANEXO I: CONJUNTOS DE CARACTERES

CONTENIDOS

- I.1 ASCII
- I.2 EBCDIC
- I.3 OEM
- I.4 ANSI
- I.5 UNICODE

I1.1 ASCII

El conjunto de caracteres ASCII (*American Standard Code for Information Interchange*) es un sistema de codificación de 8 bits. Es decir a cada byte se le asocia con un símbolo, de esta forma se definen códigos desde el 0 al 255. La norma ASCII tan sólo define el uso de los 7 primeros bits (códigos del 0 al 127), el resto se dejaron para que cada fabricante los adaptase a los distintos idiomas o necesidades.

ANEXO I: CONJUNTOS DE CARACTERES

TABLA ASCII		
Decimal	Carácter	Comentario
0	NUL	Carácter nulo
1	SOH	Carácter de control
2	STX	Carácter de control
3	ETX	Carácter de control
4	EOT	Carácter de control
5	ENQ	Carácter de control
6	ACK	Carácter de control
7	BEL	Campanilla o pitido
8	BS	Carácter de control
9	TAB	Tabulador
10	LF	Carácter de control
11	VT	Carácter de control
12	FF	Salto de página
13	CR	Retorno de carro
14	SO	Carácter de control
15	SI	Carácter de control
16	DLE	Carácter de control
17	DC1	Carácter de control
18	DC2	Carácter de control
19	DC3	Carácter de control
20	DC4	Carácter de control
21	NAK	Carácter de control
22	SYN	Carácter de control
23	ETB	Carácter de control
24	CAN	Carácter de control
25	EM	Carácter de control
26	SUB	Carácter de control
27	ESC	Escape
28	FS	Carácter de control
29	GS	Carácter de control
30	RS	Carácter de control
31	US	Carácter de control
32		Espacio en blanco
33	!	Admiración
34	"	Comillas
35	#	Almohadilla
36	\$	Dolar
37	%	Porcentaje
38	&	And
39	'	Apóstrofo
40	(Abrir paréntesis
41)	Cerrar paréntesis
42	*	Asterisco
43	+	Mas
44	,	Coma
45	-	Menos
46	.	Punto
47	/	División
48	0	Dígito
49	1	Dígito
50	2	Dígito
51	3	Dígito
52	4	Dígito
53	5	Dígito
54	6	Dígito
55	7	Dígito
56	8	Dígito
57	9	Dígito

ASCII

TABLA ASCII		
Decimal	Carácter	Comentario
58	:	Dos puntos
59	,	Punto y coma
60	<	Menor
61	=	Igual
62	>	Mayor
63	?	Cerrar interrogación
64	@	Arroba
65	A	Letra mayúscula
66	B	Letra mayúscula
67	C	Letra mayúscula
68	D	Letra mayúscula
69	E	Letra mayúscula
70	F	Letra mayúscula
71	G	Letra mayúscula
72	H	Letra mayúscula
73	I	Letra mayúscula
74	J	Letra mayúscula
75	K	Letra mayúscula
76	L	Letra mayúscula
77	M	Letra mayúscula
78	N	Letra mayúscula
79	O	Letra mayúscula
80	P	Letra mayúscula
81	Q	Letra mayúscula
82	R	Letra mayúscula
83	S	Letra mayúscula
84	T	Letra mayúscula
85	U	Letra mayúscula
86	V	Letra mayúscula
87	W	Letra mayúscula
88	X	Letra mayúscula
89	Y	Letra mayúscula
90	Z	Letra mayúscula
91	[Abrir corchete
92	\	Barra hacia atrás
93]	Cerrar corchete
94	^	Acento circunflejo
95	_	Subrayado
96	`	Apóstrofo
97	a	Letra minúscula
98	b	Letra minúscula
99	c	Letra minúscula
100	d	Letra minúscula
101	e	Letra minúscula
102	f	Letra minúscula
103	g	Letra minúscula
104	h	Letra minúscula
105	i	Letra minúscula
106	j	Letra minúscula
107	k	Letra minúscula
108	l	Letra minúscula
109	m	Letra minúscula
110	n	Letra minúscula
111	o	Letra minúscula
112	p	Letra minúscula
113	q	Letra minúscula
114	r	Letra minúscula
115	s	Letra minúscula

ANEXO I: CONJUNTOS DE CARACTERES

TABLA ASCII		
Decimal	Carácter	Comentario
116	t	Letra minúscula
117	u	Letra minúscula
118	v	Letra minúscula
119	w	Letra minúscula
120	x	Letra minúscula
121	y	Letra minúscula
122	z	Letra minúscula
123	{	Abrir llave
124		Barra vertical
125	}	Cerrar llave
126	~	Tilde
127	DEL	Borrar, suprimir
128		ASCII extendido
...
254		ASCII extendido
255		ASCII extendido

EBCDIC

II.1 EBCDIC

IBM en un principio no se acogió a la norma ASCII y hoy en día algunos de sus minordenadores y ordenadores *mainframe* todavía mantienen otro conjunto de caracteres denominado EBCDIC (*Extended Binary Coded Decimal Information Code*).

Decimal	Carácter	Decimal	Carácter	Decimal	Carácter
064	blanco	132	d	200	H
074]	133	e	201	I
075	.	134	f	209	J
076	<	135	g	210	K
077	(136	h	211	L
078	+	137	i	212	M
079	!	145	j	213	N
080	&	146	k	214	O
090	[147	l	215	P
091	\$	148	m	216	Q
092	*	149	n	217	R
093)	150	o	226	S
094	;	151	p	227	T
095	^	152	q	228	U
096	-	153	r	229	V
097	/	162	s	230	W
108	,	163	t	231	X
109	%	164	u	232	Y
110	_	165	v	233	Z
111	>	166	w	240	0
112	?	167	x	241	1
122	:	168	y	242	2
123	#	169	z	243	3
124	@	193	A	244	4
125	'	194	B	245	5
126	=	195	C	246	6
127	>	196	D	247	7
129	a	197	E	248	8
130	b	198	F	249	9
131	c	199	G		

II.1 OEM

El sistema operativo DOS trabaja con un conjunto de caracteres ASCII denominado OEM (*Original Equipment Manufacturer*), que fija unos caracteres determinados desde los códigos 128 a 255, según sea la página de códigos cargada en el sistema:

437 Caracteres EE.UU. (grabada en ROM)

ANEXO I: CONJUNTOS DE CARACTERES

- 850 Caracteres de Europa Occidental (incluye a España)
- 852 Caracteres de Europa del Este (DOS 5.0 y posteriores)
- 860 Caracteres portugueses
- 861 Caracteres islandeses
- 863 Caracteres franco-canadienses
- 864 Caracteres nórdicos

El conjunto ANSI-OEM no es un estándar, aunque la extensión del DOS hace que sea uno de los más utilizados.

	!	"	#	\$	%	&	'	<	>	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	<		>	~	Δ
Ç	ü	é	â	ä	à	ç	ê	ë	è	ï	î	ì	ñ	æ	œ	ø	ö	ò	û	ù	ý	õ	ü	ç	£	¥	₹	₺	₱		
á	í	ó	ú	ñ	ñ	º	¿	¡	½	¾	¿	«	»	▒	▓	▔		†	‡	§	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	
Ł	ł	Ť	ť	-	†	‡	§	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	
α	β	Γ	Π	Σ	σ	μ	τ	ϑ	θ	Ω	δ	ω	ø	€	∞	≡	±	≥	≤	∫	∫	∫	∫	∫	∫	∫	∫	∫	∫	∫	

Conjunto de caracteres ASCII-OEM del sistema operativo DOS (página 437)

11.1 ANSI

El conjunto básico de Windows se denomina ANSI, y sí es un estándar normalizado. Además Windows permite cambiar el conjunto de caracteres con tan sólo cambiar la fuente de letra. El estándar ANSI no coincide con el OEM, por eso el compilador Borland Pascal incluye un programa de conversión entre ambos conjuntos de caracteres.

	!	"	#	\$	%	&	'	<	>	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	<		>	~	Δ
Ç	ü	é	â	ä	à	ç	ê	ë	è	ï	î	ì	ñ	æ	œ	ø	ö	ò	û	ù	ý	õ	ü	ç	£	¥	₹	₺	₱		
á	í	ó	ú	ñ	ñ	º	¿	¡	½	¾	¿	«	»	▒	▓	▔		†	‡	§	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	
Ł	ł	Ť	ť	-	†	‡	§	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	¶	
α	β	Γ	Π	Σ	σ	μ	τ	ϑ	θ	Ω	δ	ω	ø	€	∞	≡	±	≥	≤	∫	∫	∫	∫	∫	∫	∫	∫	∫	∫	∫	

Conjunto de caracteres ANSI de la fuente Terminal de Windows

UNICODE

	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
	ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Conjunto de caracteres ANSI de la fuente *System* de Windows

	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
`	a	b	c	d	e	f	g	h	i	j	l	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	+
·	·	·	f	·	·	·	†	‡	ˆ	%	§	€	€	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Conjunto de caracteres ANSI de la fuente *Times Roman* de Windows

	↖	↗	↘	↙	↕	↔	↞	↠	↡	↢	↣	↤	↥	↦	↧	↨	↩	↪	↫	↬	↭	↮	↯	↰	↱	↲	↳	↴	↵	↶	↷	
	↸	↹	↺	↻	↼	↽	↾	↿	⇀	⇁	⇂	⇃	⇄	⇅	⇆	⇇	⇈	⇉	⇊	⇋	⇌	⇍	⇎	⇏	⇐	⇑	⇒	⇓	⇔	⇕	⇖	⇗
∏	∑	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	
∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	
∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	∏	
→	↑	↓	↖	↗	↘	↙	↕	↔	↞	↠	↡	↢	↣	↤	↥	↦	↧	↨	↩	↪	↫	↬	↭	↮	↯	↰	↱	↲	↳	↴	↵	

Conjunto de caracteres ANSI de una fuente *True Type* de Windows

11.1 UNICODE

Las escrituras de los países asiáticos como China o Japón emplean miles de caracteres independientes que no pueden ser codificados con 8 bits. Para aceptar un conjunto más amplio de escrituras nace un nuevo estándar de representación de conjuntos de caracteres *Unicode*.

Unicode es un esquema de codificación de 16 bits, que puede representar 65536 caracteres (2¹⁶). Esta cifra es suficiente para incluir todos los caracteres de todos los lenguajes que se utilizan actualmente en los ordenadores, así como algunos lenguajes arcaicos con aplicaciones limitadas, como el sanscrito o los jeroglíficos egipcios. Los 7 primeros bits siguen siendo el código ASCII.

ANEXO I: CONJUNTOS DE CARACTERES

Unicode también incluye representaciones de signos de puntuación, caracteres matemáticos, símbolos gráficos (*dingbats*), escrituras griega, latina, cirílica, armenia, árabe, hebrea, chino, japonés y coreano. Aún quedan zonas sin definir para uso futuro.

El sistema operativo Windows NT incorpora el conjunto de caracteres Unicode.

INTRODUCCION

[PICTURE]

ANEXO II

ANEXO II: DIAGRAMAS SINTACTICOS

CONTENIDOS

- II.1 Introducción
- II.2 Pascal estándar
- II.3 Tipo objeto de Turbo Pascal

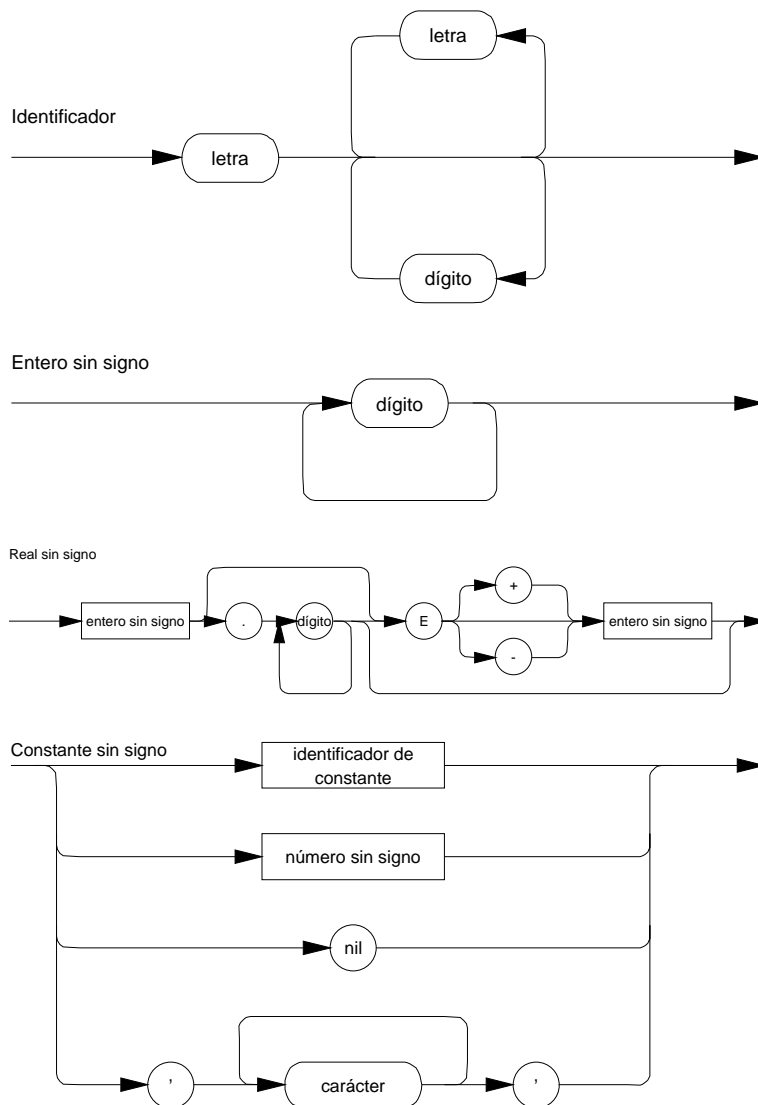
III.1 INTRODUCCION

Los diagramas sintácticos son un metalenguaje para la definición de la sintaxis de los lenguajes de programación. Constan de una serie de cajas o símbolos geométricos conectados por flechas donde se introducen los símbolos del lenguaje que se dividen en:

- **Símbolos terminales:** Son los que forman las sentencias del lenguaje y se introducen dentro de círculos o cajas de bordes redondeados.
- **Símbolos no terminales:** Son introducidos como elementos auxiliares y no figuran en las sentencias del lenguaje. Se representan por su nombre encerrado en un rectángulo o cuadrado.

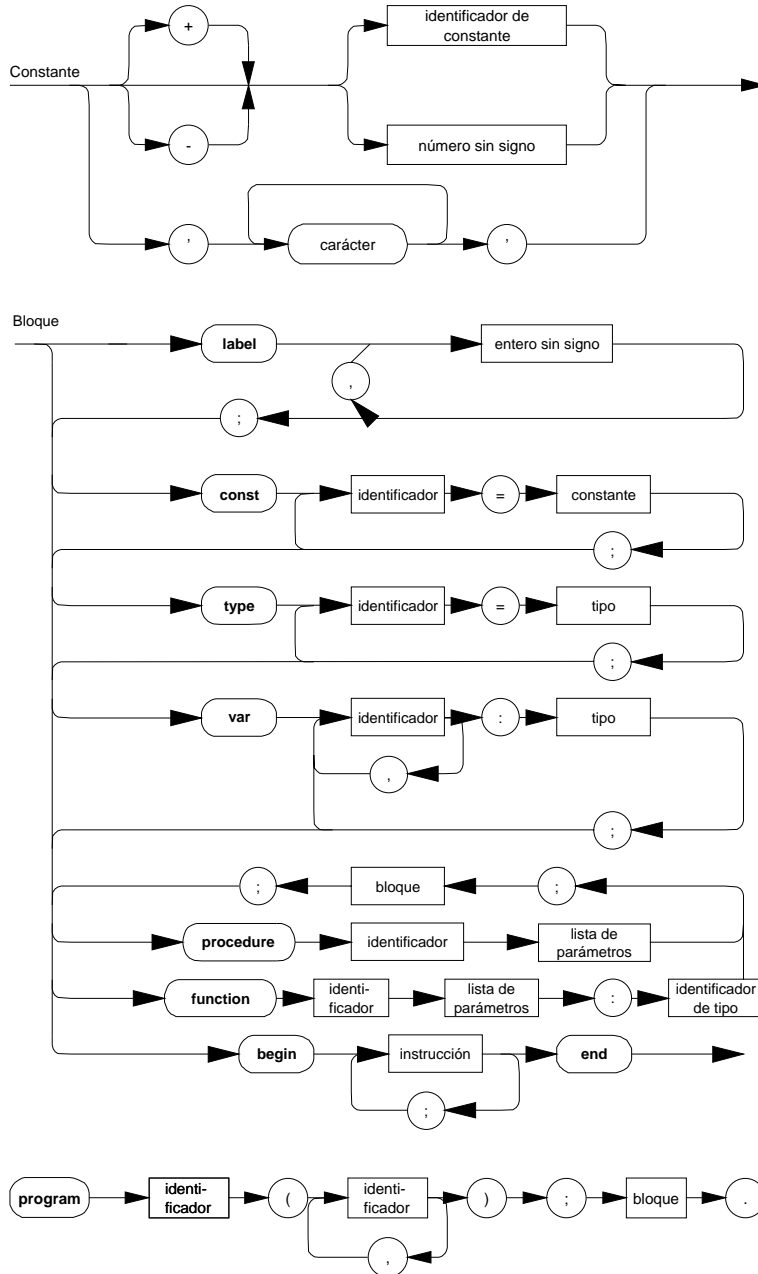
III.2 PASCAL ESTANDAR

Estos diagramas sintácticos son parte de la definición de la norma ISO⁹⁵ del lenguaje Pascal. Consultar el libro *User manual and report ISO Pascal Standard* de Jensen K. y N. Wirth (Springer-Verlag, 4ª edición, 1991). Letra y dígito se han supuesto como símbolos terminales.

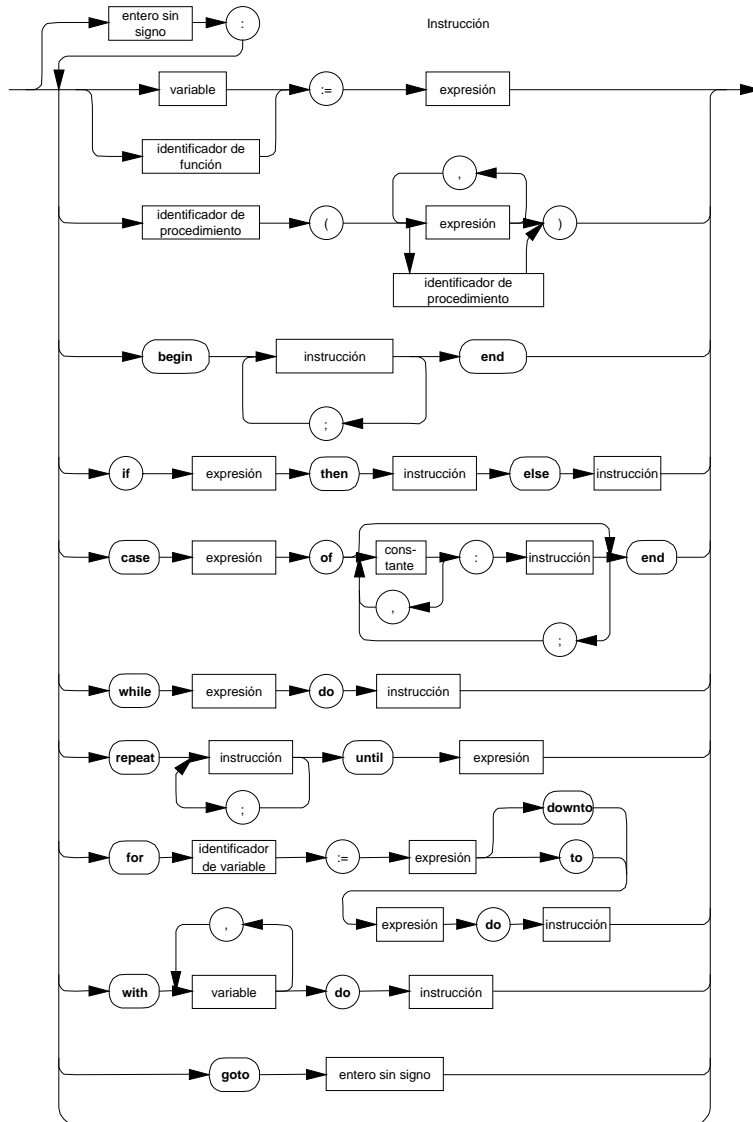


⁹⁵ ISO (International Standards Organization) es la Organización Internacional para la definición de normalizaciones.

PASCAL ESTANDAR

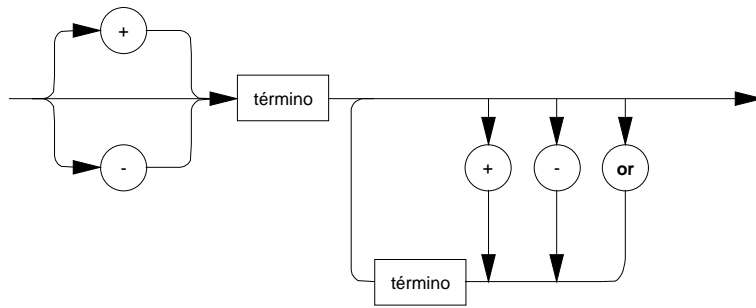


ANEXO II: DIAGRAMAS SINTACTICOS

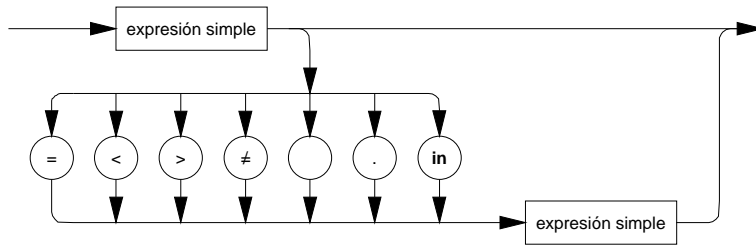


PASCAL ESTANDAR

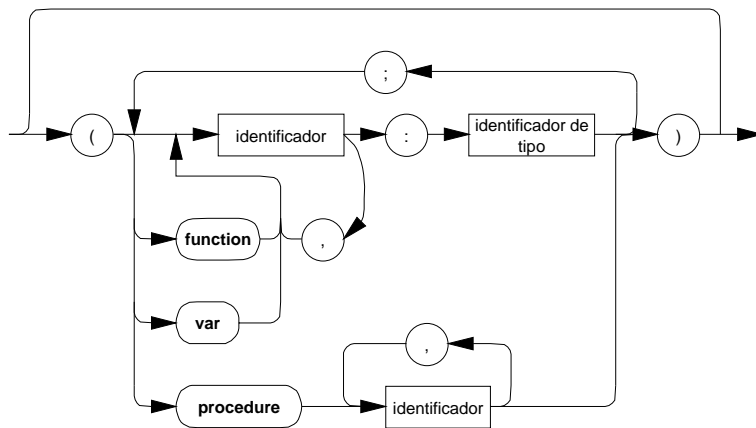
Expresión simple



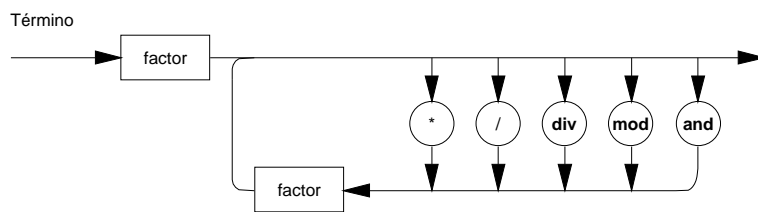
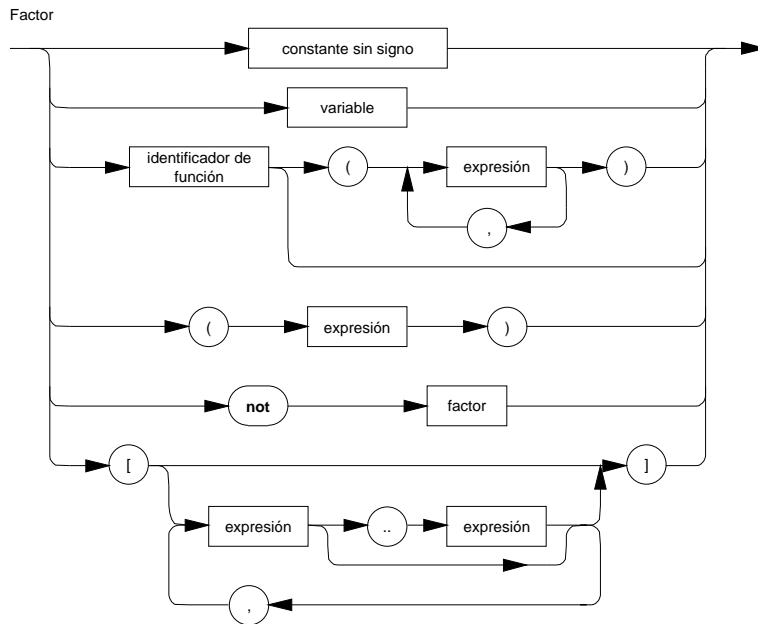
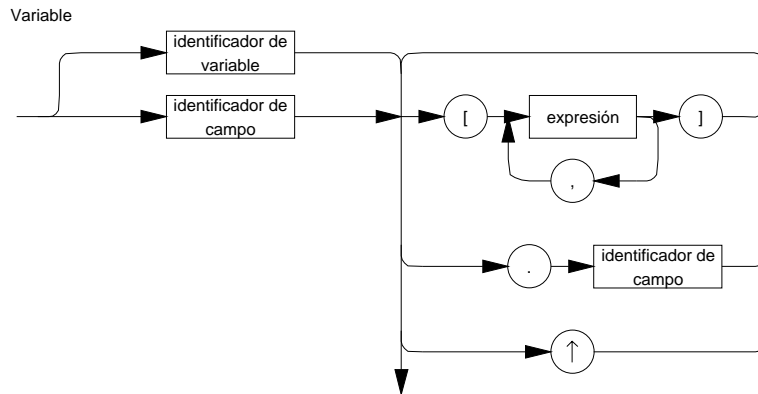
Expresión



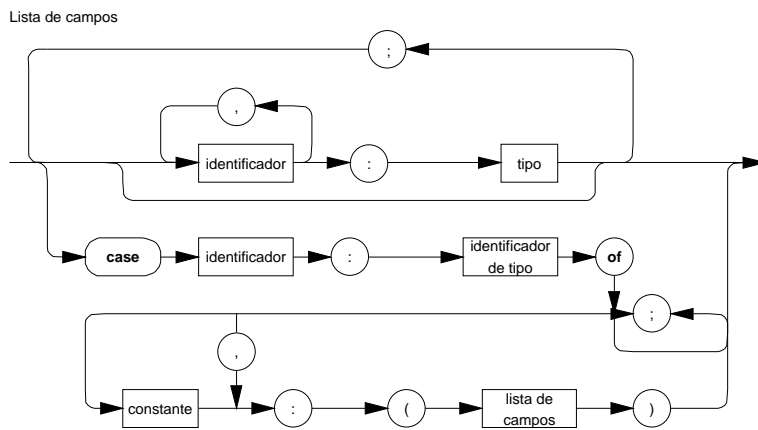
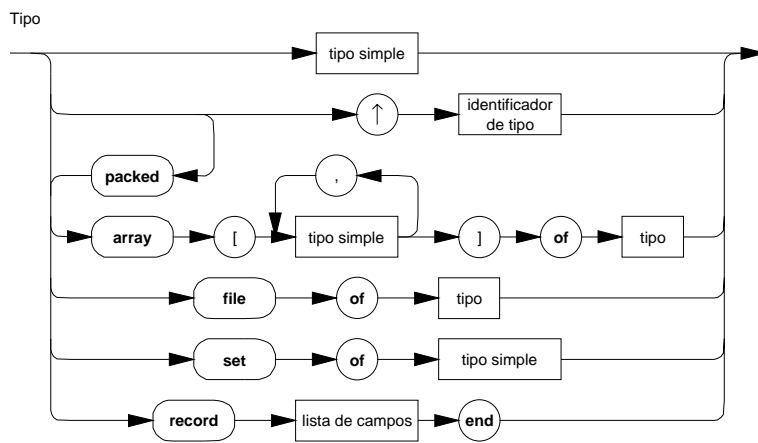
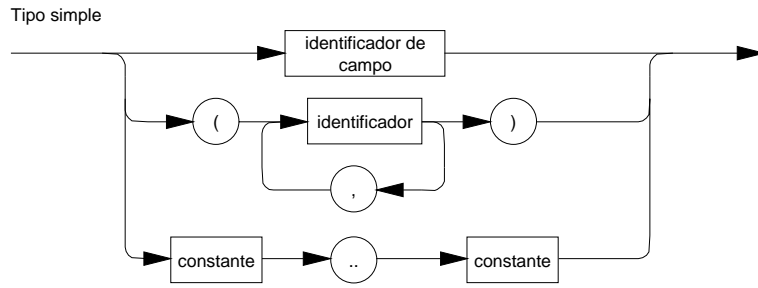
Lista de parámetros



ANEXO II: DIAGRAMAS SINTACTICOS

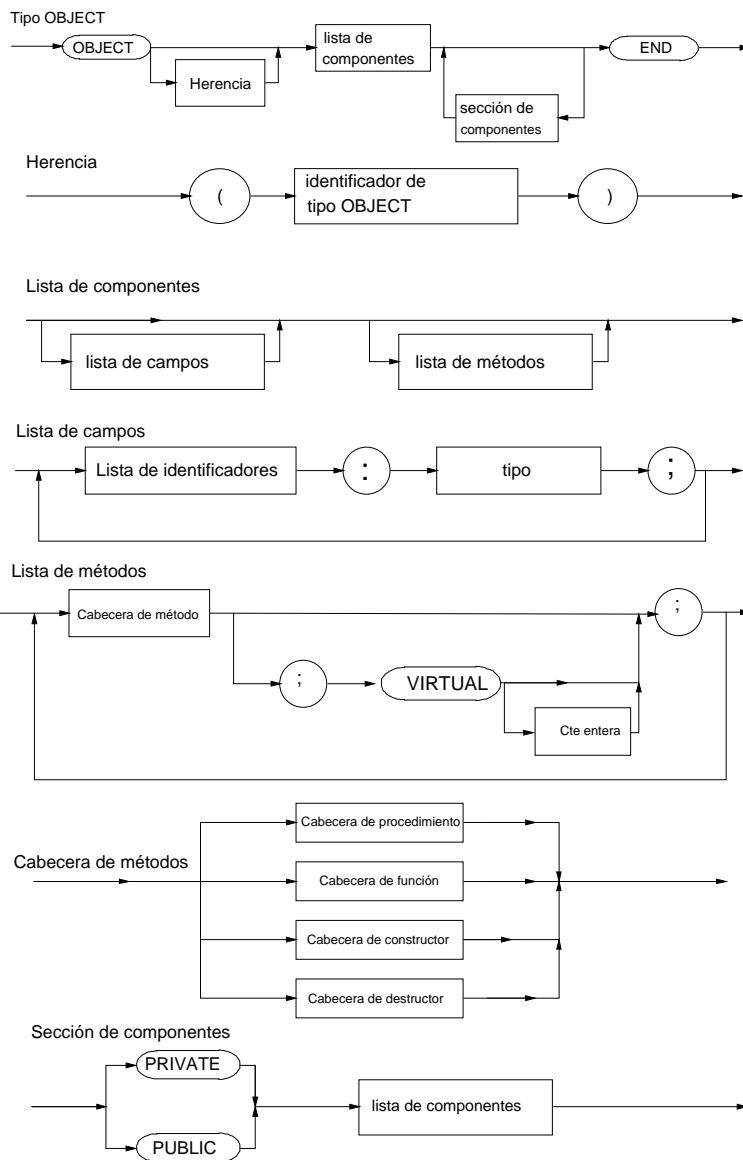


PASCAL ESTANDAR

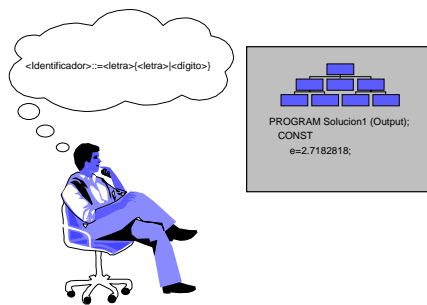


III.3 TIPO OBJETO DE TURBO PASCAL

A continuación se presentan el diagrama sintáctico del tipo objeto que soporta Turbo Pascal (versión 7), para la programación orientada a objetos.



INTRODUCCION



ANEXO III

ANEXO III: NOTACION EBNF

CONTENIDOS

- III.1 Introducción
- III.2 Pascal estándar
- III.3 Tipo objeto de Turbo Pascal

III.1 INTRODUCCION

La notación EBNF (*Extended Backus-Naur Form*) es un metalenguaje para describir la sintaxis de los lenguajes de programación.

La notación EBNF utiliza los siguientes metasímbolos:

< > Encierra conceptos definidos o por definir. Se utiliza para los símbolos no terminales.

::= Sirve para definir o indicar equivalencia.

| Separa las distintas alternativas.

{ } Indica que lo que aparece entre llaves puede repetirse cero o más veces. En algunos casos se indica con subíndices y superíndices el intervalo de repeticiones.

" " Indica que el metasímbolo que aparece entre comillas es un caracter que forma parte de la sintaxis del lenguaje.

() Se permite el uso de paréntesis para hacer agrupaciones.

III.2 PASCAL ESTANDAR

Esta gramática en notación EBNF es parte de la definición de la norma ISO⁹⁶ del lenguaje Pascal. Consultar el libro *User manual and report ISO Pascal Standard* de Jensen K. y N. Wirth (Springer-Verlag, 4ª edición, 1991).

```

<programa> ::= <encabezamiento del programa><bloque>.
<encabezamiento> ::= program <identificador> (<identificador de archivo>
{, <identificador de archivo>});
<identificador de archivo> ::= <identificador>
<identificador> ::= <letra> {<letra o dígito>}
<bloque> ::= <parte de declaración de rótulos>
<parte de definición de constantes>
<parte de definición de tipos> <parte de declaración de variables>
<parte de declaración de procedimientos y funciones> <sentencias>
<parte de declaración de rótulos> ::= <vacía> | label <rótulo>{, <rótulo>};
<rótulo> ::= <entero sin signo>
<parte de definición de constantes> ::= const <definición de constante>
{; <definición de constante>};
<definición de constante> ::= <identificador> = <constante>
<constante> ::= <número sin signo> | <signo><número sin signo> |
<identificador de constante> | <signo><identificador de constante> |
<cadena de caracteres>
<número sin signo> ::= <entero sin signo> | <real sin signo>
<entero sin signo> ::= <dígito> {<dígito>}
<real sin signo> ::= <entero sin signo> . <dígito> {<dígito>} |
<entero sin signo> . <dígito> {<dígito>} | E <factor de escala> |
<entero sin signo> E <factor de escala>
<factor de escala> ::= <entero sin signo> | <signo><entero sin signo>
<signo> ::= + | -
<identificador de constante> ::= <identificador>
<cadena de caracteres> ::= ' <carácter> {<carácter>}'
<parte de definición de tipos> ::= <vacía> |
type <definición de tipo> {; <definición de tipo>};
<definición de tipo> ::= <identificador> = <tipo>
<tipo> ::= <tipo simple> | <tipo estructurado> | <tipo puntero>
<tipo simple> ::= <tipo escalar> | <tipo subrango> | <tipo identificador>
<tipo escalar> ::= (<identificador>, {<identificador>})
<tipo subrango> ::= <constante> .. <constante>
<tipo identificador> ::= <identificador>
<tipo estructurado> ::= <tipo estructurado no empaquetado> |
packed <tipo estructurado no empaquetado>
<tipo estructurado no empaquetado> ::= <tipo arreglo> | <tipo registro> |
<tipo conjunto> | <tipo archivo>
<tipo arreglo> ::= array [<tipo índice> {, <tipo índice>}] of
<tipo componente>
<tipo índice> ::= <tipo simple>
<tipo componente> ::= <tipo>
<tipo registro> ::= record <lista de campos> end
<lista de campos> ::= <parte fija> | <parte fija>; <parte variable> |
<parte variable>

```

⁹⁶ ISO (International Standards Organization) es la Organización Internacional para la definición de normalizaciones.

PASCAL ESTANDAR

```
<parte fija> ::= <sección de registro> {; <sección de registro>}
<sección de registro> ::= <identificador de campo>
{, <identificador de campo>}:
<tipo> | <vacío>
<parte variable> ::= case <campo de etiquetas> <identificador de tipo> of
<variante> {; <variante>}
<campo de etiquetas> ::= <identificador del campo>: | <vacío>
<variante> ::= <lista de rótulos del case>: (<lista de campos> | <vacío>
<lista de rótulos del case> ::= <rótulo del case> {, <rótulo del case>}
<rótulo del case> ::= <constante>
<tipo conjunto> ::= set of <tipo base>
<tipo base> ::= <tipo simple>
<tipo archivo> ::= file of <tipo>
<tipo puntero> ::= ↑ <identificador de tipo>
<parte de declaración de variable> ::= <vacía> |
var <declaración de variable> {; <declaración de variable>
<declaración de variable> ::= <identificador> {, <identificador>}: <tipo>
<parte de declaración de procedimientos y funciones> ::=
<declaración de procedimientos> | <declaración de funciones>
<declaración de procedimientos> ::= <encabezamiento de procedimiento>
| <bloque>
<encabezamiento de procedimiento> ::= procedure <identificador>; |
procedure <identificador> (<sección de parámetros formales>
{; <sección de parámetros formales>});
<sección de parámetros formales> ::= <grupo de parámetros> |
var <grupo de parámetros> | function <grupo de parámetros> |
procedure <identificador> {, <identificador>}
<grupo de parámetros> ::= <identificador> {, <identificador>}:
<identificador de tipo>
<declaración de función> ::= <encabezamiento de función><bloque>
<encabezamiento de función> ::= function <identificador> :
<tipo de resultado>; |
function <identificador> (<sección de parámetros formales>
{; <sección de parámetros formales>}): <tipo de resultado>;
<tipo de resultado> ::= <identificador de tipo>
<parte de sentencias> ::= <sentencia compuesta>
<sentencia> ::= <sentencia sin rótulo> | <rótulo><sentencia sin rótulo>
<sentencia sin rótulo> ::= <sentencia simple> | <sentencia estructurada>
<sentencia simple> ::= <sentencia de asignación> |
<sentencia de procedimiento> |
<sentencia go to> | <sentencia vacía>
<sentencia de asignación> ::= <variable> := <expresión> |
<identificador de función> := <expresión>
<variable> ::= <variable completa> | <componente de variable> |
<referencia a variable>
<variable completa> ::= <identificador de variable>
<identificador de variable> ::= <identificador>
<componente de variable> ::= <variable indexada> |
<designador de campo> | <buffer de archivo>
<variable indexada> ::= <variable de arreglo> [<expresión> {, <expresión>}]
<variable de arreglo> ::= <variable>
<designador de campo> ::= <variable de registro> . <identificador de campo>
<variable de registro> ::= <variable>
<identificador de campo> ::= <identificador>
<buffer de archivo> ::= <variable de archivo>↑
<variable de archivo> ::= <variable>
<referencia a variable> ::= <variable puntero>↑
<expresión> ::= <expresión simple> | <expresión simple>
<operador relacional>
<expresión simple>
<operador relacional> ::= = | <> | < | > | <= | >= | in
<expresión simple> ::= <término> | <signo><término> |
<expresión simple><operador de adición><término>
<operador de adición> ::= + | - | or
<término> ::= <factor> | <término><operador de multiplicación><factor>
<operador de multiplicación> ::= * | / | div | mod | and
```

ANEXO III: NOTACION EBNF

```

<factor> ::= <variable> | <constante sin signo> | ( <expresión> ) |
<designador de función> | <conjunto> | not <factor>
<constante sin signo> ::= <numero sin signo> | <cadena de caracteres> |
<identificador de constantes> | nil
<designador de función> ::= <identificador de función> |
<identificador de función>
( <parámetro actual> { , <parámetro actual> } )
<identificador de función> ::= <identificador>
<conjunto> ::= { <lista de elementos> }
<lista de elementos> ::= <elemento> { , <elemento> } | <vacía>
<elemento> ::= <expresión> | <expresión> .. <expresión>
<sentencia de procedimiento> ::= <identificador de procedimiento> |
<identificador de procedimiento> ( <parámetro actual>
{ , <parámetro actual> } )
<identificador de procedimiento> ::= <identificador>
<parámetro actual> ::= <expresión> | <variable> |
<identificador de procedimiento> | <identificador de función>
<sentencia go to> ::= goto <rótulo>
<sentencia vacía> ::= <vacía>
<vacía> ::=
<sentencia estructurada> ::= <sentencia compuesta> |
<sentencia condicional> | <sentencias repetitivas> | <sentencia with>
<sentencia compuesta> ::= begin <sentencia> { ; <sentencia> } end
<sentencia condicional> ::= <sentencia if> | <sentencia case>
<sentencia if> ::= if <expresión> then <sentencia> |
if <expresión> then <sentencia> else <sentencia>
<sentencia case> ::= case <expresión> of <elemento de la lista case>
{ ; <elemento de la lista case> } end
<elemento de la lista case> ::= <lista de rótulos de case>: <sentencia> |
<vacía>
<lista de rótulos de case> ::= <rótulo de case> { , <rótulo de case> }
<sentencias repetitivas> ::= <sentencia while> | <sentencia repeat> |
<sentencia for>
<sentencia while> ::= while <expresión> do <sentencia>
<sentencia repeat> ::= repeat <sentencia> { ; <sentencia> }
until <expresión>
<sentencia for> ::= for <variable de control> := <lista for> do <sentencia>
<lista for> ::= <valor inicial> to <valor final> |
<valor inicial> downto <valor final>
<variable de control> ::= <variable>
<valor inicial> ::= <expresión>
<valor final> ::= <expresión>
<sentencia with> ::= with <lista de variables de registro> do <sentencia>
<lista de variables de registro> ::= <variable de registro>
{ , <variable de registro> }

```

III.3 TIPO OBJETO DE TURBO PASCAL

```

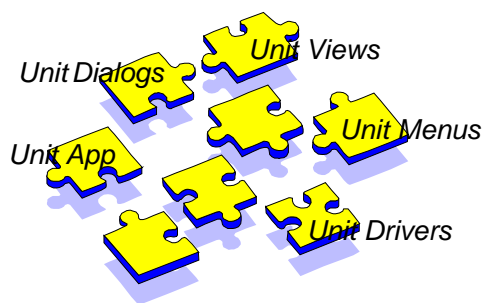
<Tipo OBJECT> ::= OBJECT (<herencia>|<vacío>
                        <lista de componentes>
                        {<sección de componentes>}
                        END
<herencia> ::= "(" <identificador de tipo OBJECT> ")"
<lista de componentes> ::= {<lista de campos>} {<lista de métodos>}
<sección de componentes> ::= ( PUBLIC | PRIVATE ) <lista de componentes>
<lista de campos> ::= {<lista de identificadores> : <tipo> ;}
<lista de métodos> ::= { <cabecera de métodos>
                        ( ; VIRTUAL
                          (<cte. entera> | <vacío>)
                          | <vacío> ) ; }

```

TIPO OBJETO DE TURBO PASCAL

```
<cabecera de métodos> ::= <cabecera de procedimiento> |  
                        <cabecera de función> |  
                        <cabecera de constructor> |  
                        <cabecera de destructor>  
  
<cabecera de constructor> ::= CONSTRUCTOR <identificador>; |  
                              CONSTRUCTOR <identificador>  
                              "(" <sección de parámetros formales>  
                              { ; <sección de parámetros formales> } ")"  
  
<cabecera de destructor> ::= DESTRUCTOR <identificador>; |  
                              DESTRUCTOR <identificador>  
                              "(" <sección de parámetros formales>  
                              { ; <sección de parámetros formales> } ")"
```

ANEXO III: NOTACION EBNF



ANEXO IV

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

CONTENIDOS

IV.1	Introducción
IV.2	Unit App
IV.3	Unit Colorsel
IV.4	Unit Dialogs
IV.5	Unit Drivers
IV.6	Unit Editors
IV.7	Unit Histlist
IV.8	Unit Memory
IV.9	Unit Menus
IV.10	Unit MsgBox
IV.11	Unit Objects
IV.12	Unit Outline
IV.13	Unit StdDlg
IV.14	Unit TextView
IV.15	Unit Validate
IV.16	Unit Views

IV1.1 INTRODUCCION

En este anexo se presenta el interfaz de las units de Turbo Vision, que incluyen constantes, tipos de datos, objetos y sus métodos de manipulación, así como funciones y procedimientos aportados por cada *Unit*. Los interfaces contienen comentarios e incluso la cláusula *uses* para poder observar las dependencias que cada *unit* tiene con otras *units*. Las Units *Objects* y *Validate* son comunes a las de **Object Windows**. Se puede observar en la Unit *Objects* las directivas de compilación condicional `{ $IFDEF Windows } { $IFNDEF Windows } { $ELSE } { $ENDIF }` para compilar una parte de código o no, según se esté trabajando con *Windows* o con *DOS*.

Las directivas de compilación condicional son:

`$IFDEF` `$IFNDEF` `$ELSE` `$ENDIF` `IFOPT`

\$IFDEF Compila el texto fuente que le sigue si *Nombre* está definido.

Sintaxis: `{ $IFDEF Nombre }`

Para definir *Nombre* se utiliza la directiva `$DEFINE`.

\$IFNDEF Compila el texto fuente que le sigue si *Nombre* no está definido.

Sintaxis: `{ $IFNDEF Nombre }`

\$ELSE Compila o ignora el texto fuente que le sigue. Dentro de un segmento de código fuente delimitado por un `$IFDEF` (o un `$IFNDEF`) y un `$ENDIF`, `$ELSE` compila el código que le sigue si la condición `$IFDEF` (o `$IFNDEF`) no se cumple. Si esa condición se cumple `$ELSE` ignora el código que le sigue.

Sintaxis: `{ $ELSE }`

\$ENDIF Finaliza la compilación condicional iniciada con la última directiva `$IFxxx`.

Sintaxis: `{ $ENDIF }`

\$IFOPT Compila el texto que le sigue si la *switch* está en el estado especificado. *switch* es una directiva de compilación seguida por + o -.

Sintaxis: `{ $IFOPT switch }`

Ejemplo: `{ $IFOPT I+ }`

\$DEFINE Permite definir símbolos condicionales con un nombre dado.

Sintaxis: `{ $DEFINE Nombre }`

Ejemplo: `{ $DEFINE Windows }`

El símbolo definido es reconocido durante el resto de la compilación a no ser que aparezca una directiva `{ $UNDEF Nombre }`.

`{ $DEFINE Nombre }` no tiene efecto si *Nombre* ya está definido.

\$UNDEF Elimina la definición de un símbolo condicional *Nombre* realizada previamente. El símbolo es olvidado para el resto de la compilación a no ser que vuelva a aparecer una directiva `{ $DEFINE Nombre }` posteriormente. `{ $UNDEF Nombre }` no tiene ningún efecto si *Nombre* no está definido.

UNIT App

Sintaxis: {\$UNDEF Nombre}
Ejemplo: {\$UNDEF Windows}

IV1.1 UNIT App

```
uses Objects, Drivers, Memory, HistList, Views, Menus, Dialogs;  
const  
{ Entrada de la paleta de TApplication }  
  apColor        = 0;  
  apBlackWhite = 1;  
  apMonochrome = 2;  
{ Paletas de TApplication }  
  { Paletas de color de Turbo Vision 1.0 }  
  CColor =  
    #$71#$70#$78#$74##$20##$28##$24##$17##$1F##$1A##$31##$31##$1E##$71##$1F +  
    ##$37##$3F##$3A##$13##$13##$3E##$21##$3F##$70##$7F##$7A##$13##$13##$70##$7F##$7E +  
    ##$70##$7F##$7A##$13##$13##$70##$70##$7F##$7E##$20##$2B##$2F##$78##$2E##$70##$30 +  
    ##$3F##$3E##$1F##$2F##$1A##$20##$72##$31##$31##$30##$2F##$3E##$31##$13##$38##$00;  
  CBlackWhite =  
    #$70##$70##$78##$7F##$07##$07##$0F##$07##$0F##$07##$70##$70##$07##$70##$0F +  
    ##$07##$0F##$07##$70##$70##$07##$70##$0F##$70##$7F##$7F##$70##$07##$70##$07##$0F +  
    ##$70##$7F##$7F##$70##$07##$70##$70##$7F##$7F##$07##$0F##$0F##$78##$0F##$78##$07 +  
    ##$0F##$0F##$0F##$70##$0F##$07##$70##$70##$70##$07##$70##$0F##$07##$07##$78##$00;  
  CMonochrome =  
    #$70##$07##$07##$0F##$70##$70##$70##$07##$0F##$07##$70##$70##$07##$70##$00 +  
    ##$07##$0F##$07##$70##$70##$07##$70##$00##$70##$70##$70##$07##$07##$70##$07##$00 +  
    ##$70##$70##$70##$07##$07##$70##$70##$70##$0F##$07##$07##$0F##$70##$0F##$70##$07 +  
    ##$0F##$0F##$07##$70##$07##$07##$70##$07##$07##$70##$0F##$07##$07##$70##$00;  
  { Paletas de color de Turbo Vision 2.0 }  
  CAppColor =  
    #$71#$70#$78#$74##$20##$28##$24##$17##$1F##$1A##$31##$31##$1E##$71##$1F +  
    ##$37##$3F##$3A##$13##$13##$3E##$21##$3F##$70##$7F##$7A##$13##$13##$70##$7F##$7E +  
    ##$70##$7F##$7A##$13##$13##$70##$70##$7F##$7E##$20##$2B##$2F##$78##$2E##$70##$30 +  
    ##$3F##$3E##$1F##$2F##$1A##$20##$72##$31##$31##$30##$2F##$3E##$31##$13##$38##$00 +  
    ##$17##$1F##$1A##$71##$71##$1E##$17##$1F##$1E##$20##$2B##$2F##$78##$2E##$10##$30 +  
    ##$3F##$3E##$70##$2F##$7A##$20##$12##$31##$31##$30##$2F##$3E##$31##$13##$38##$00 +  
    ##$37##$3F##$3A##$13##$13##$3E##$30##$3F##$3E##$20##$2B##$2F##$78##$2E##$30##$70 +  
    ##$7F##$7E##$1F##$2F##$1A##$20##$32##$31##$71##$70##$2F##$7E##$71##$13##$38##$00;  
  CAppBlackWhite =  
    #$70##$70##$78##$7F##$07##$07##$0F##$07##$0F##$07##$70##$70##$07##$70##$0F +  
    ##$07##$0F##$07##$70##$70##$07##$70##$0F##$70##$7F##$7F##$70##$07##$70##$07##$0F +  
    ##$70##$7F##$7F##$70##$07##$70##$70##$7F##$7F##$07##$0F##$0F##$78##$0F##$78##$07 +  
    ##$0F##$0F##$0F##$70##$0F##$07##$70##$70##$70##$07##$70##$0F##$07##$07##$78##$00 +  
    ##$07##$0F##$0F##$07##$70##$07##$07##$0F##$0F##$70##$78##$7F##$08##$7F##$08##$70 +  
    ##$7F##$7F##$7F##$0F##$70##$70##$07##$70##$70##$70##$07##$7F##$70##$07##$78##$00 +  
    ##$70##$7F##$7F##$70##$07##$70##$70##$7F##$7F##$07##$0F##$0F##$78##$0F##$78##$07 +  
    ##$0F##$0F##$0F##$70##$0F##$07##$70##$70##$70##$07##$70##$0F##$07##$07##$78##$00;  
  CAppMonochrome =  
    #$70##$07##$07##$0F##$70##$70##$70##$07##$0F##$07##$70##$70##$07##$70##$00 +  
    ##$07##$0F##$07##$70##$70##$07##$70##$00##$70##$70##$70##$07##$07##$70##$07##$00 +  
    ##$70##$70##$70##$07##$07##$70##$70##$70##$0F##$07##$07##$0F##$70##$0F##$70##$07 +  
    ##$0F##$0F##$07##$70##$07##$07##$70##$07##$07##$70##$0F##$07##$07##$70##$00 +  
    ##$70##$70##$70##$07##$07##$70##$70##$70##$0F##$07##$07##$0F##$70##$0F##$70##$07 +  
    ##$0F##$0F##$07##$70##$07##$07##$70##$07##$07##$70##$0F##$07##$07##$70##$00;  
{ Paleta de TBackground }  
  CBackground = #1;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

{ Comandos de aplicación Estándar }
cmNew      = 30;
cmOpen     = 31;
cmSave     = 32;
cmSaveAs   = 33;
cmSaveAll  = 34;
cmChangeDir = 35;
cmDosShell = 36;
cmCloseAll = 37;
{ Contexto de Ayuda de aplicación estándar }
{ Nota:
El rango $FF00 - $FFFF de contextos de ayuda están reservados por Borland }
hcNew      = $FF01;
hcOpen     = $FF02;
hcSave     = $FF03;
hcSaveAs   = $FF04;
hcSaveAll  = $FF05;
hcChangeDir = $FF06;
hcDosShell = $FF07;
hcExit     = $FF08;
hcUndo     = $FF10;
hcCut      = $FF11;
hcCopy     = $FF12;
hcPaste    = $FF13;
hcClear    = $FF14;
hcTile     = $FF20;
hcCascade  = $FF21;
hcCloseAll = $FF22;
hcResize   = $FF23;
hcZoom     = $FF24;
hcNext     = $FF25;
hcPrev     = $FF26;
hcClose    = $FF27;
type
{ Tipo objeto TBackground }
PBackground = ^TBackground;
TBackground = object(TView)
  Pattern: Char;
  constructor Init(var Bounds: TRect; APattern: Char);
  constructor Load(var S: TStream);
  procedure Draw; virtual;
  function GetPalette: PPalette; virtual;
  procedure Store(var S: TStream);
end;
{ Tipo objeto TDesktop }
PDesktop = ^TDesktop;
TDesktop = object(TGroup)
  Background: PBackground;
  TileColumnsFirst: Boolean;
  constructor Init(var Bounds: TRect);
  constructor Load(var S: TStream);
  procedure Cascade(var R: TRect);
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure InitBackground; virtual;
  procedure Store(var S: TStream);
  procedure Tile(var R: TRect);
  procedure TileError; virtual;
end;
{ Tipo objeto TProgram }
{ Distribución de la Paleta }
{ 1 = TBackground }
{ 2- 7 = TMenuView y TStatusLine }
{ 8-15 = TWindow(Azul) }
{ 16-23 = TWindow(Ciano) }
{ 24-31 = TWindow(Gris) }
{ 32-63 = TDialog }

```

UNIT *ColorSel*

```
PProgram = ^TProgram;
TProgram = object(TGroup)
  constructor Init;
  destructor Done; virtual;
  function CanMoveFocus: Boolean;
  function ExecuteDialog(P: PDialog; Data: Pointer): Word;
  procedure GetEvent(var Event: TEvent); virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure Idle; virtual;
  procedure InitDesktop; virtual;
  procedure InitMenuBar; virtual;
  procedure InitScreen; virtual;
  procedure InitStatusLine; virtual;
  function InsertWindow(P: PWindow): PWindow;
  procedure OutOfMemory; virtual;
  procedure PutEvent(var Event: TEvent); virtual;
  procedure Run; virtual;
  procedure SetScreenMode(Mode: Word);
  function ValidView(P: PView): PView;
end;
{ Tipo objeto TApplication }
PApplication = ^TApplication;
TApplication = object(TProgram)
  constructor Init;
  destructor Done; virtual;
  procedure Cascade;
  procedure DosShell;
  procedure GetTileRect(var R: TRect); virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure Tile;
  procedure WriteShellMsg; virtual;
end;
{ Menú Estandar y lineas de estado }
function StdStatusKeys(Next: PStatusItem): PStatusItem;
function StdFileMenuItems(Next: PMenuItem): PMenuItem;
function StdEditMenuItems(Next: PMenuItem): PMenuItem;
function StdWindowMenuItems(Next: PMenuItem): PMenuItem;
{ Procedure de registro de App }
procedure RegisterApp;
const
{ Variables Públicas }
  Application: PProgram = nil;
  Desktop: PDesktop = nil;
  StatusLine: PStatusLine = nil;
  MenuBar: PMenuView = nil;
  AppPalette: Integer = apColor;
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RBackground: TStreamRec = (
    ObjType: 30;
    VmtLink: ofs(EOF(TBackground)^);
    Load: @TBackground.Load;
    Store: @TBackground.Store);
const
  RDesktop: TStreamRec = (
    ObjType: 31;
    VmtLink: ofs(EOF(TDesktop)^);
    Load: @TDesktop.Load;
    Store: @TDesktop.Store);
```

IV1.1 UNIT *ColorSel*

uses Objects, Drivers, Views, Dialogs;

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

const
  cmColorForegroundColorChanged = 71;
  cmColorBackgroundColorChanged = 72;
  cmColorSet                     = 73;
  cmNewColorItem                 = 74;
  cmNewColorIndex                = 75;
  cmSaveColorIndex               = 76;
type
  { TColorItem }
  PColorItem = ^TColorItem;
  TColorItem = record
    Name: PString;
    Index: Byte;
    Next: PColorItem;
  end;
  { TColorGroup }
  PColorGroup = ^TColorGroup;
  TColorGroup = record
    Name: PString;
    Index: Byte;
    Items: PColorItem;
    Next: PColorGroup;
  end;
  { TColorIndexes }
  PColorIndex = ^TColorIndex;
  TColorIndex = record
    GroupIndex: byte;
    ColorSize: byte;
    ColorIndex: array[0..255] of byte;
  end;
  { TColorSelector }
  TColorSel = (csBackground, csForeground);
  PColorSelector = ^TColorSelector;
  TColorSelector = object(TView)
    Color: Byte;
    SelType: TColorSel;
    constructor Init(var Bounds: TRect; ASelType: TColorSel);
    constructor Load(var S: TStream);
    procedure Draw; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure Store(var S: TStream);
  end;
  { TMonoSelector }
  PMonoSelector = ^TMonoSelector;
  TMonoSelector = object(TCluster)
    constructor Init(var Bounds: TRect);
    procedure Draw; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    function Mark(Item: Integer): Boolean; virtual;
    procedure NewColor;
    procedure Press(Item: Integer); virtual;
    procedure MovedTo(Item: Integer); virtual;
  end;
  { TColorDisplay }
  PColorDisplay = ^TColorDisplay;
  TColorDisplay = object(TView)
    Color: ^Byte;
    Text: PString;
    constructor Init(var Bounds: TRect; AText: PString);
    constructor Load(var S: TStream);
    destructor Done; virtual;
    procedure Draw; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure SetColor(var AColor: Byte); virtual;
    procedure Store(var S: TStream);
  end;
  { TColorGroupList }

```

UNIT *ColorSel*

```

PColorGroupList = ^TColorGroupList;
TColorGroupList = object(TListViewer)
  Groups: PColorGroup;
  constructor Init(var Bounds: TRect; AScrollBar: PScrollBar;
    AGroups: PColorGroup);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure FocusItem(Item: Integer); virtual;
  function GetText(Item: Integer; MaxLen: Integer): String; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure Store(var S: TStream);
  procedure SetGroupIndex(GroupNum, ItemNum: Byte);
  function GetGroup(GroupNum: Byte): PColorGroup;
  function GetGroupIndex(GroupNum: Byte): Byte;
  function GetNumGroups: byte;
end;
{ TColorItemList }
PColorItemList = ^TColorItemList;
TColorItemList = object(TListViewer)
  Items: PColorItem;
  constructor Init(var Bounds: TRect; AScrollBar: PScrollBar;
    AItems: PColorItem);
  procedure FocusItem(Item: Integer); virtual;
  function GetText(Item: Integer; MaxLen: Integer): String; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
end;
{ TColorDialog }
PColorDialog = ^TColorDialog;
TColorDialog = object(TDialog)
  GroupIndex: byte;
  Display: PColorDisplay;
  Groups: PColorGroupList;
  ForLabel: PLabel;
  ForSel: PColorSelector;
  BakLabel: PLabel;
  BakSel: PColorSelector;
  MonoLabel: PLabel;
  MonoSel: PMonoSelector;
  Pal: TPalette;
  constructor Init(APalette: TPalette; AGroups: PColorGroup);
  constructor Load(var S: TStream);
  function DataSize: Word; virtual;
  procedure GetData(var Rec); virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure SetData(var Rec); virtual;
  procedure Store(var S: TStream);
  procedure GetIndexes(var Colors: PColorIndex);
  procedure SetIndexes(var Colors: PColorIndex);
end;
{ Puntero a los índices de los elementos de la lista de color salvados }
const
  ColorIndexes: PColorIndex = nil;
{ Rutinas para Cargar y Almacenar la paleta }
procedure StoreIndexes(var S: TStream);
procedure LoadIndexes(var S: TStream);
{ Rutinas para construir la lista de Color }
function ColorItem(const Name: String; Index: Byte;
  Next: PColorItem): PColorItem;
function ColorGroup(const Name: String; Items: PColorItem;
  Next: PColorGroup): PColorGroup;
{ Funciones de los elementos de Color Estándar }
function DesktopColorItems(const Next: PColorItem): PColorItem;
function MenuColorItems(const Next: PColorItem): PColorItem;
function DialogColorItems(Palette: Word; const Next: PColorItem): PColorItem;
function WindowColorItems(Palette: Word; const Next: PColorItem): PColorItem;
{ Procedure para registrar ColorSel }
procedure RegisterColorSel;

```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RColorSelector: TStreamRec = (
    ObjType: 21;
    VmtLink: Ofs(KindOf(TColorSelector)^);
    Load: @TColorSelector.Load;
    Store: @TColorSelector.Store
  );
const
  RMonoSelector: TStreamRec = (
    ObjType: 22;
    VmtLink: Ofs(KindOf(TMonoSelector)^);
    Load: @TMonoSelector.Load;
    Store: @TMonoSelector.Store
  );
const
  RColorDisplay: TStreamRec = (
    ObjType: 23;
    VmtLink: Ofs(KindOf(TColorDisplay)^);
    Load: @TColorDisplay.Load;
    Store: @TColorDisplay.Store
  );
const
  RColorGroupList: TStreamRec = (
    ObjType: 24;
    VmtLink: Ofs(KindOf(TColorGroupList)^);
    Load: @TColorGroupList.Load;
    Store: @TColorGroupList.Store
  );
const
  RColorItemList: TStreamRec = (
    ObjType: 25;
    VmtLink: Ofs(KindOf(TColorItemList)^);
    Load: @TColorItemList.Load;
    Store: @TColorItemList.Store
  );
const
  RColorDialog: TStreamRec = (
    ObjType: 26;
    VmtLink: Ofs(KindOf(TColorDialog)^);
    Load: @TColorDialog.Load;
    Store: @TColorDialog.Store
  );
```

IV1.1 UNIT Dialogs

uses Objects, Drivers, Views, Validate;

```
const
{ Paletas de Color }
CGrayDialog = #32#33#34#35#36#37#38#39#40#41#42#43#44#45#46#47 +
              #48#49#50#51#52#53#54#55#56#57#58#59#60#61#62#63;
CBlueDialog = #64#65#66#67#68#69#70#71#72#73#74#75#76#77#78#79 +
              #80#81#82#83#84#85#86#87#88#89#90#91#92#93#94#95;
CCyanDialog = #96#97#98#99#100#101#102#103#104#105#106#107#108 +
              #109#110#111#112#113#114#115#116#117#118#119#120 +
              #121#122#123#124#125#126#127;
CDialog = CGrayDialog;
CStaticText = #6;
CLabel = #7#8#9#9;
CButton = #10#11#12#13#14#14#15;
CCluster = #16#17#18#18#31;
CInputLine = #19#19#20#21;
```


UNIT Dialogs

```

CHistory      = #22#23;
CHistoryWindow = #19#19#21#24#25#19#20;
CHistoryViewer = #6#6#7#6#6;
{ Totalidad de Paletas de TDialog }
dpBlueDialog = 0;
dpCyanDialog = 1;
dpGrayDialog = 2;
{ Flags de TButton }
bfNormal      = $00;
bfDefault     = $01;
bfLeftJust    = $02;
bfBroadcast   = $04;
bfGrabFocus   = $08;
{ Flags de TMultiCheckboxes }
{ hibyte = número de bits }
{ lobyte = máscara binaria }
cfOneBit      = $0101;
cfTwoBits     = $0203;
cfFourBits    = $040F;
cfEightBits   = $08FF;
type
{ Tipo objeto TDialog }
{ Distribución de la Paleta }
{ 1 = Marco pasivo }
{ 2 = Marco activo }
{ 3 = Icono de marco }
{ 4 = Area de pagina de barra de desplazamiento }
{ 5 = Controles de barra de desplazamiento }
{ 6 = Texto estático }
{ 7 = Etiqueta normal }
{ 8 = Etiqueta seleccionada }
{ 9 = Atajo(shortcut) de etiqueta }
{ 10 = Botón normal }
{ 11 = Botón por defecto }
{ 12 = Botón seleccionado }
{ 13 = Botón inhabilitado }
{ 14 = Atajo de botón }
{ 15 = Sombra de botón }
{ 16 = Cluster normal }
{ 17 = Cluster seleccionado }
{ 18 = Atajo de cluster }
{ 19 = Texto de Linea de Entrada normal }
{ 20 = Texto de Linea de Entrada seleccionada }
{ 21 = Flechas de Linea de Entrada }
{ 22 = Flechas de historial(history) }
{ 23 = Laterales de historial }
{ 24 = Area de página de barra de desplazamiento de la ventana de historial }
{ 25 = Controles de barra de desplazamiento de la ventana de historial }
{ 26 = Visor de lista normal }
{ 27 = Visor de lista con foco (focused) }
{ 28 = Visor de lista seleccionado }
{ 29 = Divisor de Visor de lista }
{ 30 = Panel de información (InfoPane) }
{ 31 = Cluster inhabilitado }
{ 32 = Reservado }
PDialog = ^TDialog;
TDialog = object(TWindow)
  constructor Init(var Bounds: TRect; ATitle: TTitleStr);
  constructor Load(var S: TStream);
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  function Valid(Command: Word): Boolean; virtual;
end;
{ TSItem }

```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

PSItem = ^TSItem;
TSItem = record
  Value: PString;
  Next: PSItem;
end;
{ Tipo objeto TInputLine }
{ Distribucción de la Paleta }
{ 1 = Pasiva }
{ 2 = Activa }
{ 3 = Seleccionada }
{ 4 = Flechas }
PInputLine = ^TInputLine;
TInputLine = object(TView)
  Data: PString;
  MaxLen: Integer;
  CurPos: Integer;
  FirstPos: Integer;
  SelStart: Integer;
  SelEnd: Integer;
  Validator: PValidator;
  constructor Init(var Bounds: TRect; AMaxLen: Integer);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  function DataSize: Word; virtual;
  procedure Draw; virtual;
  procedure GetData(var Rec); virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure SelectAll(Enable: Boolean);
  procedure SetData(var Rec); virtual;
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure SetValidator(AValid: PValidator);
  procedure Store(var S: TStream);
  function Valid(Command: Word): Boolean; virtual;
private
  function CanScroll(Delta: Integer): Boolean;
end;
{ Tipo objeto TButton }
{ Distribucción de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto por defecto }
{ 3 = Texto seleccionado }
{ 4 = Texto inhabilitado }
{ 5 = Atajo(shortcut) normal }
{ 6 = Atajo por defecto }
{ 7 = Atajo seleccionado }
{ 8 = Sombra }
PButton = ^TButton;
TButton = object(TView)
  Title: PString;
  Command: Word;
  Flags: Byte;
  AmDefault: Boolean;
  constructor Init(var Bounds: TRect; ATitle: TTitleStr; ACommand: Word;
    AFlags: Word);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Draw; virtual;
  procedure DrawState(Down: Boolean);
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure MakeDefault(Enable: Boolean);
  procedure Press; virtual;
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure Store(var S: TStream);
end;
{ TCluster }

```

UNIT Dialogs

```
{ Distribucción de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto seleccionado }
{ 3 = Atajo(shortcut) normal }
{ 4 = Atajo seleccionado }
{ 5 = Texto inhabilitado }

PCluster = ^TCluster;
TCluster = object(TView)
  Value: LongInt;
  Sel: Integer;
  EnableMask: LongInt;
  Strings: TStringCollection;
  constructor Init(var Bounds: TRect; AStrings: PString);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  function ButtonState(Item: Integer): Boolean;
  function DataSize: Word; virtual;
  procedure DrawBox(const Icon: String; Marker: Char);
  procedure DrawMultiBox(const Icon, Marker: String);
  procedure GetData(var Rec); virtual;
  function GetHelpCtx: Word; virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  function Mark(Item: Integer): Boolean; virtual;
  function MultiMark(Item: Integer): Byte; virtual;
  procedure Press(Item: Integer); virtual;
  procedure MovedTo(Item: Integer); virtual;
  procedure SetButtonState(AMask: Longint; Enable: Boolean);
  procedure SetData(var Rec); virtual;
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure Store(var S: TStream);
private
  function Column(Item: Integer): Integer;
  function FindSel(P: TPoint): Integer;
  function Row(Item: Integer): Integer;
end;
{ TRadioButtons }
{ Distribucción de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto seleccionado }
{ 3 = Atajo(shortcut) normal }
{ 4 = Atajo seleccionado }
PRadioButtons = ^TRadioButtons;
TRadioButtons = object(TCluster)
  procedure Draw; virtual;
  function Mark(Item: Integer): Boolean; virtual;
  procedure MovedTo(Item: Integer); virtual;
  procedure Press(Item: Integer); virtual;
  procedure SetData(var Rec); virtual;
end;
{ TCheckBoxes }
{ Distribucción de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto seleccionado }
{ 3 = Atajo(shortcut) normal }
{ 4 = Atajo seleccionado }

PCheckBoxes = ^TCheckBoxes;
TCheckBoxes = object(TCluster)
  procedure Draw; virtual;
  function Mark(Item: Integer): Boolean; virtual;
  procedure Press(Item: Integer); virtual;
end;
{ TMultiCheckBoxes }
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

{ Distribución de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto seleccionado }
{ 3 = Atajo(shortcut) normal }
{ 4 = Atajo seleccionado }

PMultiCheckBoxes = ^TMultiCheckBoxes;
TMultiCheckBoxes = object(TCluster)
  SelRange: Byte;
  Flags: Word;
  States: PString;
  constructor Init(var Bounds: TRect; AStrings: PSItem;
    ASelRange: Byte; AFlags: Word; const AStates: String);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  function DataSize: Word; virtual;
  procedure Draw; virtual;
  procedure GetData(var Rec); virtual;
  function MultiMark(Item: Integer): Byte; virtual;
  procedure Press(Item: Integer); virtual;
  procedure SetData(var Rec); virtual;
  procedure Store(var S: TStream);
end;
{ TListBox }
{ Distribución de la Paleta }
{ 1 = Activo }
{ 2 = Inactivo }
{ 3 = Con foco (focused) }
{ 4 = Seleccionado }
{ 5 = Divisor }
PListBox = ^TListBox;
TListBox = object(TListViewer)
  List: PCollection;
  constructor Init(var Bounds: TRect; ANumCols: Word;
    AScrollBar: PScrollBar);
  constructor Load(var S: TStream);
  function DataSize: Word; virtual;
  procedure GetData(var Rec); virtual;
  function GetText(Item: Integer; MaxLen: Integer): String; virtual;
  procedure NewList(AList: PCollection); virtual;
  procedure SetData(var Rec); virtual;
  procedure Store(var S: TStream);
end;
{ TStaticText }
{ Distribución de la Paleta }
{ 1 = Texto }
PStaticText = ^TStaticText;
TStaticText = object(TView)
  Text: PString;
  constructor Init(var Bounds: TRect; const AText: String);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Draw; virtual;
  function GetPalette: PPalette; virtual;
  procedure GetText(var S: String); virtual;
  procedure Store(var S: TStream);
end;
{ TParamText }
{ Distribución de la Paleta }
{ 1 = Texto }

PParamText = ^TParamText;
TParamText = object(TStaticText)
  ParamCount: Integer;
  ParamList: Pointer;
  constructor Init(var Bounds: TRect; const AText: String;
    AParamCount: Integer);

```

UNIT Dialogs

```

    constructor Load(var S: TStream);
    function DataSize: Word; virtual;
    procedure GetText(var S: String); virtual;
    procedure SetData(var Rec); virtual;
    procedure Store(var S: TStream);
end;
{ TLabel }
{ Distribución de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto seleccionado }
{ 3 = Atajo(shortcut) normal }
{ 4 = Atajo seleccionado }
PLabel = ^TLabel;
TLabel = object(TStaticText)
    Link: PView;
    Light: Boolean;
    constructor Init(var Bounds: TRect; const AText: String; ALink: PView);
    constructor Load(var S: TStream);
    procedure Draw; virtual;
    function GetPalette: PPalette; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure Store(var S: TStream);
end;
{ THistoryViewer }
{ Distribución de la Paleta }
{ 1 = Activo }
{ 2 = Inactivo }
{ 3 = Con foco (focused) }
{ 4 = Seleccionado }
{ 5 = Divisor }
PHistoryViewer = ^THistoryViewer;
THistoryViewer = object(TListViewer)
    HistoryId: Word;
    constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar;
        AHistoryId: Word);
    function GetPalette: PPalette; virtual;
    function GetText(Item: Integer; MaxLen: Integer): String; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    function HistoryWidth: Integer;
end;
{ THistoryWindow }
{ Distribución de la Paleta }
{ 1 = Marco pasivo }
{ 2 = Marco activo }
{ 3 = Icono de marco }
{ 4 = Area de pagina de barra de desplazamiento }
{ 5 = Controles de barra de desplazamiento }
{ 6 = Texto normal del Visor de historial }
{ 7 = Texto seleccionado del Visor de historial }
PHistoryWindow = ^THistoryWindow;
THistoryWindow = object(TWindow)
    Viewer: PListViewer;
    constructor Init(var Bounds: TRect; HistoryId: Word);
    function GetPalette: PPalette; virtual;
    function GetSelection: String; virtual;
    procedure InitViewer(HistoryId: Word); virtual;
end;
{ THistory }
{ Distribución de la Paleta }
{ 1 = Flechas }
{ 2 = Laterales }

```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

PHistory = ^THistory;
THistory = object(TView)
  Link: PInputLine;
  HistoryId: Word;
  constructor Init(var Bounds: TRect; ALink: PInputLine; AHistoryId: Word);
  constructor Load(var S: TStream);
  procedure Draw; virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  function InitHistoryWindow(var Bounds: TRect): PHistoryWindow; virtual;
  procedure RecordHistory(const S: String); virtual;
  procedure Store(var S: TStream);
end;
{ Rutinas SItem }
function NewSItem(const Str: String; ANext: PSItem): PSItem;
{ Procedure para registrar dialogos }
procedure RegisterDialogs;
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RDialog: TStreamRec = (
    ObjType: 10;
    VmtLink: Ofs(KindOf(TDialog)^);
    Load: @TDialog.Load;
    Store: @TDialog.Store
  );
const
  RInputLine: TStreamRec = (
    ObjType: 11;
    VmtLink: Ofs(KindOf(TInputLine)^);
    Load: @TInputLine.Load;
    Store: @TInputLine.Store
  );
const
  RButton: TStreamRec = (
    ObjType: 12;
    VmtLink: Ofs(KindOf(TButton)^);
    Load: @TButton.Load;
    Store: @TButton.Store
  );
const
  RCluster: TStreamRec = (
    ObjType: 13;
    VmtLink: Ofs(KindOf(TCluster)^);
    Load: @TCluster.Load;
    Store: @TCluster.Store
  );
const
  RRadioButtons: TStreamRec = (
    ObjType: 14;
    VmtLink: Ofs(KindOf(TRadioButtons)^);
    Load: @TRadioButtons.Load;
    Store: @TRadioButtons.Store
  );
const
  RCheckBoxes: TStreamRec = (
    ObjType: 15;
    VmtLink: Ofs(KindOf(TCheckBoxes)^);
    Load: @TCheckBoxes.Load;
    Store: @TCheckBoxes.Store
  );
const
  RMultiCheckBoxes: TStreamRec = (
    ObjType: 27;
    VmtLink: Ofs(KindOf(TMultiCheckBoxes)^);
    Load: @TMultiCheckBoxes.Load;
    Store: @TMultiCheckBoxes.Store
  );

```

UNIT Drivers

```
const
  RListBox: TStreamRec = (
    ObjType: 16;
    VmtLink: Ofs(KindOf(TListBox)^);
    Load:   @TListBox.Load;
    Store:   @TListBox.Store
  );
const
  RStaticText: TStreamRec = (
    ObjType: 17;
    VmtLink: Ofs(KindOf(TStaticText)^);
    Load:   @TStaticText.Load;
    Store:   @TStaticText.Store
  );
const
  RLabel: TStreamRec = (
    ObjType: 18;
    VmtLink: Ofs(KindOf(TLabel)^);
    Load:   @TLabel.Load;
    Store:   @TLabel.Store
  );
const
  RHistory: TStreamRec = (
    ObjType: 19;
    VmtLink: Ofs(KindOf(THistory)^);
    Load:   @THistory.Load;
    Store:   @THistory.Store
  );
const
  RParamText: TStreamRec = (
    ObjType: 20;
    VmtLink: Ofs(KindOf(TParamText)^);
    Load:   @TParamText.Load;
    Store:   @TParamText.Store
  );
const
{ Comandos de difusión de Dialog }
  cmRecordHistory = 60;
```

IV1.1 UNIT Drivers

```
uses Objects;
{ ***** MANEJO DE EVENTOS ***** }
const
{ Códigos de eventos }
  evMouseDown = $0001;
  evMouseUp   = $0002;
  evMouseMove = $0004;
  evMouseAuto = $0008;
  evKeyDown   = $0010;
  evCommand   = $0100;
  evBroadcast = $0200;
{ Máscaras de eventos }
  evNothing   = $0000;
  evMouse     = $000F;
  evKeyboard  = $0010;
  evMessage   = $FF00;
{ Códigos de teclas extendidas }
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

kbEsc      = $011B; kbAltSpace = $0200; kbCtrlIns  = $0400;
kbShiftIns = $0500; kbCtrlDel  = $0600; kbShiftDel = $0700;
kbBack     = $0E08; kbCtrlBack = $0E7F; kbShiftTab = $0F00;
kbTab      = $0F09; kbAltQ     = $1000; kbAltW     = $1100;
kbAltE     = $1200; kbAltR     = $1300; kbAltT     = $1400;
kbAltY     = $1500; kbAltU     = $1600; kbAltI     = $1700;
kbAltO     = $1800; kbAltP     = $1900; kbCtrlEnter = $1C0A;
kbEnter    = $1C0D; kbAltA     = $1E00; kbAltS     = $1F00;
kbAltD     = $2000; kbAltF     = $2100; kbAltG     = $2200;
kbAltH     = $2300; kbAltJ     = $2400; kbAltK     = $2500;
kbAltL     = $2600; kbAltZ     = $2C00; kbAltX     = $2D00;
kbAltC     = $2E00; kbAltV     = $2F00; kbAltB     = $3000;
kbAltN     = $3100; kbAltM     = $3200; kbF1       = $3B00;
kbF2       = $3C00; kbF3       = $3D00; kbF4       = $3E00;
kbF5       = $3F00; kbF6       = $4000; kbF7       = $4100;
kbF8       = $4200; kbF9       = $4300; kbF10      = $4400;
kbHome     = $4700; kbUp       = $4800; kbPgUp     = $4900;
kbGrayMinus = $4A2D; kbLeft    = $4B00; kbRight    = $4D00;
kbGrayPlus = $4E2B; kbEnd      = $4F00; kbDown     = $5000;
kbPgDn     = $5100; kbIns      = $5200; kbDel      = $5300;
kbShiftF1  = $5400; kbShiftF2  = $5500; kbShiftF3  = $5600;
kbShiftF4  = $5700; kbShiftF5  = $5800; kbShiftF6  = $5900;
kbShiftF7  = $5A00; kbShiftF8  = $5B00; kbShiftF9  = $5C00;
kbShiftF10 = $5D00; kbCtrlF1   = $5E00; kbCtrlF2   = $5F00;
kbCtrlF3   = $6000; kbCtrlF4   = $6100; kbCtrlF5   = $6200;
kbCtrlF6   = $6300; kbCtrlF7   = $6400; kbCtrlF8   = $6500;
kbCtrlF9   = $6600; kbCtrlF10  = $6700; kbAltF1    = $6800;
kbAltF2    = $6900; kbAltF3    = $6A00; kbAltF4    = $6B00;
kbAltF5    = $6C00; kbAltF6    = $6D00; kbAltF7    = $6E00;
kbAltF8    = $6F00; kbAltF9    = $7000; kbAltF10   = $7100;
kbCtrlPrtSc = $7200; kbCtrlLeft = $7300; kbCtrlRight = $7400;
kbCtrlEnd  = $7500; kbCtrlPgDn = $7600; kbCtrlHome = $7700;
kbAlt1     = $7800; kbAlt2     = $7900; kbAlt3     = $7A00;
kbAlt4     = $7B00; kbAlt5     = $7C00; kbAlt6     = $7D00;
kbAlt7     = $7E00; kbAlt8     = $7F00; kbAlt9     = $8000;
kbAlt0     = $8100; kbAltMinus = $8200; kbAltEqual  = $8300;
kbCtrlPgUp = $8400; kbAltBack  = $0800; kbNoKey    = $0000;

```

```

{ Estado del teclado y máscaras de desplazamiento (shift) }

```

```

kbRightShift = $0001;
kbLeftShift  = $0002;
kbCtrlShift  = $0004;
kbAltShift   = $0008;
kbScrollState = $0010;
kbNumState   = $0020;
kbCapsState  = $0040;
kbInsState   = $0080;

```

```

{ Máscaras de estado de los botones del ratón }

```

```

mbLeftButton = $01;
mbRightButton = $02;

```

```

type

```

```

{ Estructura Record de eventos }

```

```

PEvent = ^TEvent;
TEvent = record
  What: Word;
  case Word of
    evNothing: ();
    evMouse: (
      Buttons: Byte;
      Double: Boolean;
      Where: TPoint);
    evKeyDown: (
      case Integer of
        0: (KeyCode: Word);
        1: (CharCode: Char;
           ScanCode: Byte));
    evMessage: (
      Command: Word;

```


UNIT Drivers

```
    case Word of
      0: (InfoPtr: Pointer);
      1: (InfoLong: Longint);
      2: (InfoWord: Word);
      3: (InfoInt: Integer);
      4: (InfoByte: Byte);
      5: (InfoChar: Char));
  end;
const
{ Variables inicializadas }
  ButtonCount: Byte = 0;
  MouseEvents: Boolean = False;
  MouseReverse: Boolean = False;
  DoubleDelay: Word = 8;
  RepeatDelay: Word = 8;
var
{ Variables sin inicializar }
  MouseIntFlag: Byte;
  MouseButtons: Byte;
  MouseWhere: TPoint;
{ Rutinas de manejo de eventos }
procedure InitEvents;
procedure DoneEvents;
procedure ShowMouse;
procedure HideMouse;
procedure GetMouseEvent(var Event: TEvent);
procedure GetKeyEvent(var Event: TEvent);
function GetShiftState: Byte;
{ ***** MANEJO DE PANTALLA ***** }
const
{ Modos de pantalla }
  smBW80    = $0002;
  smCO80    = $0003;
  smMono    = $0007;
  smFont8x8 = $0100;
const
{ Variables inicializadas }
  StartupMode: Word = $FFFF;
var
{ Variables sin inicializar }
  ScreenMode: Word;
  ScreenWidth: Byte;
  ScreenHeight: Byte;
  HiResScreen: Boolean;
  CheckSnow: Boolean;
  ScreenBuffer: Pointer;
  CursorLines: Word;
{ Rutinas de manejo de pantalla }
procedure InitVideo;
procedure DoneVideo;
procedure SetVideoMode(Mode: Word);
procedure ClearScreen;
{ ***** MANEJADOR DE ERRORES DE SISTEMA ***** }
type
{ Tipo de las funciones para manejo de errores de sistemas }
  TSysErrorFunc = function(ErrorCode: Integer; Drive: Byte): Integer;
{ Rutina por defecto para manejo de errores de sistema }
function SystemError(ErrorCode: Integer; Drive: Byte): Integer;
const
{ Variables sin inicializar }
  SaveInt09: Pointer = nil;
  SysErrorFunc: TSysErrorFunc = SystemError;
  SysColorAttr: Word = $4E4F;
  SysMonoAttr: Word = $7070;
  CtrlBreakHit: Boolean = False;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```
SaveCtrlBreak: Boolean = False;
SysErrActive: Boolean = False;
FailSysErrors: Boolean = False;
{ Rutinas para manejo de errores de sistema }
procedure InitSysError;
procedure DoneSysError;
{ ***** RUTINAS DE UTILIDADES ***** }
{ Rutinas de soporte de teclado }
function GetAltChar(KeyCode: Word): Char;
function GetAltCode(Ch: Char): Word;
function GetCtrlChar(KeyCode: Word): Char;
function GetCtrlCode(Ch: Char): Word;
function CtrlToArrow(KeyCode: Word): Word;
{ Rutinas para String }
procedure FormatStr(var Result: String; const Format: String; var Params);
procedure PrintStr(const S: String);
{ Rutinas para movimientos a buffer }
procedure MoveBuf (var Dest; var Source; Attr: Byte; Count: Word);
procedure MoveChar(var Dest; C: Char; Attr: Byte; Count: Word);
procedure MoveCStr(var Dest; const Str: String; Attrs: Word);
procedure MoveStr (var Dest; const Str: String; Attr: Byte);
function CStrLen (const S: String): Integer;
```

IV1.1 UNIT *Editors*

uses Drivers, Objects, Views, Dialogs;

```
const
  cmFind      = 82;
  cmReplace   = 83;
  cmSearchAgain = 84;
const
  cmCharLeft  = 500;
  cmCharRight = 501;
  cmWordLeft  = 502;
  cmWordRight = 503;
  cmLineStart = 504;
  cmLineEnd   = 505;
  cmLineUp    = 506;
  cmLineDown  = 507;
  cmPageUp    = 508;
  cmPageDown  = 509;
  cmTextStart = 510;
  cmTextEnd   = 511;
  cmNewLine   = 512;
  cmBackSpace = 513;
  cmDelChar   = 514;
  cmDelWord   = 515;
  cmDelStart  = 516;
  cmDelEnd    = 517;
  cmDelLine   = 518;
  cmInsMode   = 519;
  cmStartSelect = 520;
  cmHideSelect = 521;
  cmIndentMode = 522;
  cmUpdateTitle = 523;
const
  edOutOfMemory = 0;
  edReadError   = 1;
  edWriteError  = 2;
  edCreateError = 3;
  edSaveModify  = 4;
  edSaveUntitled = 5;
  edSaveAs      = 6;
  edFind        = 7;
```

UNIT Editors

```
edSearchFailed = 8;
edReplace      = 9;
edReplacePrompt = 10;
const
  efCaseSensitive   = $0001;
  efWholeWordsOnly = $0002;
  efPromptOnReplace = $0004;
  efReplaceAll      = $0008;
  efDoReplace       = $0010;
  efBackupFiles     = $0100;
const
  CIndicator = #2#3;
  CEditor    = #6#7;
  CMemo      = #26#27;
const
  MaxLineLength = 256;
type
  TEditorDialog = function(Dialog: Integer; Info: Pointer): Word;
type
  PIndicator = ^TIndicator;
  TIndicator = object(TView)
    Location: TPoint;
    Modified: Boolean;
    constructor Init(var Bounds: TRect);
    procedure Draw; virtual;
    function GetPalette: PPalette; virtual;
    procedure SetState(AState: Word; Enable: Boolean); virtual;
    procedure SetValue(ALocation: TPoint; AModified: Boolean);
  end;
type
  PEditBuffer = ^TEditBuffer;
  TEditBuffer = array[0..65519] of Char;
type
  PEditor = ^TEditor;
  TEditor = object(TView)
    HScrollBar: PScrollBar;
    VScrollBar: PScrollBar;
    Indicator: PIndicator;
    Buffer: PEditBuffer;
    BufSize: Word;
    BufLen: Word;
    GapLen: Word;
    SelStart: Word;
    SelEnd: Word;
    CurPtr: Word;
    CurPos: TPoint;
    Delta: TPoint;
    Limit: TPoint;
    DrawLine: Integer;
    DrawPtr: Word;
    DelCount: Word;
    InsCount: Word;
    IsValid: Boolean;
    CanUndo: Boolean;
    Modified: Boolean;
    Selecting: Boolean;
    Overwrite: Boolean;
    AutoIndent: Boolean;
    constructor Init(var Bounds: TRect;
      AHScrollBar, AVScrollBar: PScrollBar;
      AIndicator: PIndicator; ABufSize: Word);
    constructor Load(var S: TStream);
    destructor Done; virtual;
    function BufChar(P: Word): Char;
    function BufPtr(P: Word): Word;
    procedure ChangeBounds(var Bounds: TRect); virtual;
    procedure ConvertEvent(var Event: TEvent); virtual;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

function CursorVisible: Boolean;
procedure DeleteSelect;
procedure DoneBuffer; virtual;
procedure Draw; virtual;
function GetPalette: PPalette; virtual;
procedure HandleEvent(var Event: TEvent); virtual;
procedure InitBuffer; virtual;
function InsertBuffer(var P: PEditBuffer; Offset, Length: Word;
  AllowUndo, SelectText: Boolean): Boolean;
function InsertFrom(Editor: PEditor): Boolean; virtual;
function InsertText(Text: Pointer; Length: Word;
  SelectText: Boolean): Boolean;
procedure ScrollTo(X, Y: Integer);
function Search(const FindStr: String; Opts: Word): Boolean;
function SetBufSize(NewSize: Word): Boolean; virtual;
procedure SetCmdState(Command: Word; Enable: Boolean);
procedure SetSelect(NewStart, NewEnd: Word; CurStart: Boolean);
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Store(var S: TStream);
procedure TrackCursor(Center: Boolean);
procedure Undo;
procedure UpdateCommands; virtual;
function Valid(Command: Word): Boolean; virtual;
private
  LockCount: Byte;
  UpdateFlags: Byte;
  KeyState: Integer;
  function CharPos(P, Target: Word): Integer;
  function CharPtr(P: Word; Target: Integer): Word;
  function ClipCopy: Boolean;
  procedure ClipCut;
  procedure ClipPaste;
  procedure DeleteRange(StartPtr, EndPtr: Word; DelSelect: Boolean);
  procedure DoUpdate;
  procedure DoSearchReplace;
  procedure DrawLines(Y, Count: Integer; LinePtr: Word);
  procedure FormatLine(var DrawBuf; LinePtr: Word;
    Width: Integer; Colors: Word);
  procedure Find;
  function GetMousePtr(Mouse: TPoint): Word;
  function HasSelection: Boolean;
  procedure HideSelect;
  function IsClipboard: Boolean;
  function LineEnd(P: Word): Word;
  function LineMove(P: Word; Count: Integer): Word;
  function LineStart(P: Word): Word;
  procedure Lock;
  procedure NewLine;
  function NextChar(P: Word): Word;
  function NextLine(P: Word): Word;
  function NextWord(P: Word): Word;
  function PrevChar(P: Word): Word;
  function PrevLine(P: Word): Word;
  function PrevWord(P: Word): Word;
  procedure Replace;
  procedure SetBufLen(Length: Word);
  procedure SetCurPtr(P: Word; SelectMode: Byte);
  procedure StartSelect;
  procedure ToggleInsMode;
  procedure Unlock;
  procedure Update(AFlags: Byte);
end;
type
  TMemodata = record
    Length: Word;
    Buffer: TEditBuffer;
  end;
end;

```

UNIT Editors

```
type
  PMemo = ^TMemo;
  TMemo = object(TEditor)
    constructor Load(var S: TStream);
    function DataSize: Word; virtual;
    procedure GetData(var Rec); virtual;
    function GetPalette: PPalette; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure SetData(var Rec); virtual;
    procedure Store(var S: TStream);
  end;
type
  PFileEditor = ^TFileEditor;
  TFileEditor = object(TEditor)
    FileName: FNameStr;
    constructor Init(var Bounds: TRect;
      AHScrollBar, AVScrollBar: PScrollBar;
      AIndicator: PIndicator; AFileName: FNameStr);
    constructor Load(var S: TStream);
    procedure DoneBuffer; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitBuffer; virtual;
    function LoadFile: Boolean;
    function Save: Boolean;
    function SaveAs: Boolean;
    function SaveFile: Boolean;
    function SetBufSize(NewSize: Word): Boolean; virtual;
    procedure Store(var S: TStream);
    procedure UpdateCommands; virtual;
    function Valid(Command: Word): Boolean; virtual;
  end;
type
  PEditWindow = ^TEditWindow;
  TEditWindow = object(TWindow)
    Editor: PFileEditor;
    constructor Init(var Bounds: TRect;
      FileName: FNameStr; ANumber: Integer);
    constructor Load(var S: TStream);
    procedure Close; virtual;
    function GetTitle(MaxSize: Integer): TTitleStr; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure SizeLimits(var Min, Max: TPoint); virtual;
    procedure Store(var S: TStream);
  end;
function DefEditorDialog(Dialog: Integer; Info: Pointer): Word;
function CreateFindDialog: PDialog;
function CreateReplaceDialog: PDialog;
function StdEditorDialog(Dialog: Integer; Info: Pointer): Word;
const
  WordChars: set of Char = ['0'..'9', 'A'..'Z', '_', 'a'..'z'];
  EditorDialog: TEditorDialog = DefEditorDialog;
  EditorFlags: Word = efBackupFiles + efPromptOnReplace;
  FindStr: String[80] = '';
  ReplaceStr: String[80] = '';
  Clipboard: PEditor = nil;
type
  TFindDialogRec = record
    Find: String[80];
    Options: Word;
  end;
type
  TReplaceDialogRec = record
    Find: String[80];
    Replace: String[80];
    Options: Word;
  end;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

const
  REditor: TStreamRec = (
    ObjType: 70;
    VmtLink: Ofs(KindOf(TEditor)^);
    Load: @TEditor.Load;
    Store: @TEditor.Store
  );
  RMemo: TStreamRec = (
    ObjType: 71;
    VmtLink: Ofs(KindOf(TMemo)^);
    Load: @TMemo.Load;
    Store: @TMemo.Store
  );
  RFileEditor: TStreamRec = (
    ObjType: 72;
    VmtLink: Ofs(KindOf(TFileEditor)^);
    Load: @TFileEditor.Load;
    Store: @TFileEditor.Store
  );
  RIndicator: TStreamRec = (
    ObjType: 73;
    VmtLink: Ofs(KindOf(TIndicator)^);
    Load: @TIndicator.Load;
    Store: @TIndicator.Store
  );
  REditWindow: TStreamRec = (
    ObjType: 74;
    VmtLink: Ofs(KindOf(TEditWindow)^);
    Load: @TEditWindow.Load;
    Store: @TEditWindow.Store
  );
procedure RegisterEditors;

```

IV1.1 UNIT *Histlist*

```

{*****
  Estructura del buffer de un historial (History):
  Byte Byte      String      Byte Byte      String
+-----+-----+-----+-----+-----+-----+-----+
| 0 | Id | String de Historial | 0 | Id | String del Historial |
+-----+-----+-----+-----+-----+-----+
*****}

```

```

uses Objects;
const
  HistoryBlock: Pointer = nil;
  HistorySize: Word = 1024;
  HistoryUsed: Word = 0;
procedure HistoryAdd(Id: Byte; const Str: String);
function HistoryCount(Id: Byte): Word;
function HistoryStr(Id: Byte; Index: Integer): String;
procedure ClearHistory;
procedure InitHistory;
procedure DoneHistory;
procedure StoreHistory(var S: TStream);
procedure LoadHistory(var S: TStream);

```

UNIT Memory

IV1.1 UNIT Memory

```
const
  MaxHeapSize: Word = 655360 div 16;   { 640K }
  LowMemSize: Word = 4096 div 16;     { 4K }
  MaxBufMem: Word = 65536 div 16;    { 64K }
procedure InitMemory;
procedure DoneMemory;
procedure InitDosMem;
procedure DoneDosMem;
function LowMemory: Boolean;
function MemAlloc(Size: Word): Pointer;
function MemAllocSeg(Size: Word): Pointer;
procedure NewCache(var P: Pointer; Size: Word);
procedure DisposeCache(P: Pointer);
procedure NewBuffer(var P: Pointer; Size: Word);
procedure DisposeBuffer(P: Pointer);
function GetBufferSize(P: Pointer): Word;
function SetBufferSize(P: Pointer; Size: Word): Boolean;
{$IFDEF DPMI}
procedure GetBufMem(var P: Pointer; Size: Word);
procedure FreeBufMem(P: Pointer);
procedure SetMemTop(MemTop: Pointer);
{$ENDIF}
```

IV1.1 UNIT Menus

```
uses Objects, Drivers, Views;
const
  { Paletas de Color }
  CMenuView = #2#3#4#5#6#7;
  CStatusLine = #2#3#4#5#6#7;
type
  { Tipos de TMenu }
  TMenuStr = string[31];
  PMenu = ^TMenu;
  PMenuItem = ^TMenuItem;
  TMenuItem = record
    Next: PMenuItem;
    Name: PString;
    Command: Word;
    Disabled: Boolean;
    KeyCode: Word;
    HelpCtx: Word;
    case Integer of
      0: (Param: PString);
      1: (SubMenu: PMenu);
    end;
  TMenu = record
    Items: PMenuItem;
    Default: PMenuItem;
  end;
  { Tipo objeto TMenuView }
  { Distribucción de la Paleta }
  { 1 = Texto normal }
  { 2 = Texto inhabilitado }
  { 3 = Texto de atajo }
  { 4 = Selección Normal }
  { 5 = Selección inhabilitada }
  { 6 = Selección de atajo }
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

PMenuView = ^TMenuView;
TMenuView = object(TView)
  ParentMenu: PMenuView;
  Menu: PMenu;
  Current: PMenuItem;
  constructor Init(var Bounds: TRect);
  constructor Load(var S: TStream);
  function Execute: Word; virtual;
  function FindItem(Ch: Char): PMenuItem;
  procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
  function GetHelpCtx: Word; virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  function HotKey(KeyCode: Word): PMenuItem;
  function NewSubView(var Bounds: TRect; AMenu: PMenu;
    AParentMenu: PMenuView): PMenuView; virtual;
  procedure Store(var S: TStream);
end;
{ Tipo objeto TMenuBar }
{ Distribución de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto inabilitado }
{ 3 = Texto de atajo }
{ 4 = Selección Normal }
{ 5 = Selección inabilitada }
{ 6 = Selección de atajo }
PMenuBar = ^TMenuBar;
TMenuBar = object(TMenuView)
  constructor Init(var Bounds: TRect; AMenu: PMenu);
  destructor Done; virtual;
  procedure Draw; virtual;
  procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
end;
{ Tipo objeto TMenuBox }
{ Distribución de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto inabilitado }
{ 3 = Texto de atajo }
{ 4 = Selección Normal }
{ 5 = Selección inabilitada }
{ 6 = Selección de atajo }
PMenuBox = ^TMenuBox;
TMenuBox = object(TMenuView)
  constructor Init(var Bounds: TRect; AMenu: PMenu;
    AParentMenu: PMenuView);
  procedure Draw; virtual;
  procedure GetItemRect(Item: PMenuItem; var R: TRect); virtual;
end;
{ Tipo objeto TMenuPopup }
{ Distribución de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto inabilitado }
{ 3 = Texto de atajo }
{ 4 = Selección Normal }
{ 5 = Selección inabilitada }
{ 6 = Selección de atajo }
PMenuPopup = ^TMenuPopup;
TMenuPopup = object(TMenuBox)
  constructor Init(var Bounds: TRect; AMenu: PMenu);
  procedure HandleEvent(var Event: TEvent); virtual;
end;
{ TStatusItem }

```


UNIT Menus

```
PStatusItem = ^TStatusItem;
TStatusItem = record
  Next: PStatusItem;
  Text: PString;
  KeyCode: Word;
  Command: Word;
end;
{ TStatusDef }
PStatusDef = ^TStatusDef;
TStatusDef = record
  Next: PStatusDef;
  Min, Max: Word;
  Items: PStatusItem;
end;
{ TStatusLine }
{ Distribucción de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto inabilitado }
{ 3 = Texto de atajo }
{ 4 = Selección Normal }
{ 5 = Selección inabilitada }
{ 6 = Selección de atajo }
PStatusLine = ^TStatusLine;
TStatusLine = object(TView)
  Items: PStatusItem;
  Defs: PStatusDef;
  constructor Init(var Bounds: TRect; ADefs: PStatusDef);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Draw; virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  function Hint(AHelpCtx: Word): String; virtual;
  procedure Store(var S: TStream);
  procedure Update; virtual;
private
  procedure DrawSelect(Selected: PStatusItem);
  procedure FindItems;
end;
{ Rutinas TMenuItem }
function NewItem(Name, Param: TMenuStr; KeyCode: Word; Command: Word;
  AHelpCtx: Word; Next: PMenuItem): PMenuItem;
function NewLine(Next: PMenuItem): PMenuItem;
function NewSubMenu(Name: TMenuStr; AHelpCtx: Word; SubMenu: PMenu;
  Next: PMenuItem): PMenuItem;
{ Rutinas TMenu }
function NewMenu(Items: PMenuItem): PMenu;
procedure DisposeMenu(Menu: PMenu);
{ Rutinas TStatusLine }
function NewStatusDef(Amin, AMax: Word; AItems: PStatusItem;
  ANext: PStatusDef): PStatusDef;
function NewStatusKey(const AText: String; AKeyCode: Word; ACommand: Word;
  ANext: PStatusItem): PStatusItem;
{ Procedure para registrar Menus }
procedure RegisterMenus;
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RMenuBar: TStreamRec = (
    ObjType: 40;
    VmtLink: Ofs(KindOf(TMenuBar)^);
    Load: @TMenuBar.Load;
    Store: @TMenuBar.Store
  );
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

const
  RMenuBox: TStreamRec = (
    ObjType: 41;
    VmtLink: Ofs(KindOf(TMenuBox)^);
    Load:   @TMenuBox.Load;
    Store:   @TMenuBox.Store
  );
const
  RStatusLine: TStreamRec = (
    ObjType: 42;
    VmtLink: Ofs(KindOf(TStatusLine)^);
    Load:   @TStatusLine.Load;
    Store:   @TStatusLine.Store
  );
const
  RMenuPopup: TStreamRec = (
    ObjType: 43;
    VmtLink: Ofs(KindOf(TMenuPopup)^);
    Load:   @TMenuPopup.Load;
    Store:   @TMenuPopup.Store
  );

```

IV.1.1 UNIT *MsgBox*

```

uses Objects;
const
{ Clases de cajas de Mensaje }
mfWarning    = $0000; { Visualiza una caja de error Level (Warning) }
mfError      = $0001; { " una caja de error }
mfInformation = $0002; { " una caja de Información }
mfConfirmation = $0003; { " una " de Confirmación }
mfInsertInApp = $0080; { Inserta una caja de mensajes en una
                        aplicacion en lugar del Desktop }

{ Flags de botones de una caja de Mensajes }
mfYesButton  = $0100; { Poner un botón Yes en el diálogo }
mfNoButton   = $0200; { Poner un botón No en el diálogo }
mfOKButton   = $0400; { Poner un botón OK en el diálogo }
mfCancelButton = $0800; { Poner un botón Cancel en el diálogo }
mfYesNoCancel = mfYesButton + mfNoButton + mfCancelButton;
mfOKCancel   = mfOKButton + mfCancelButton;
{ Diálogos estándar Yes, No, Cancel }
{ Diálogos estándar OK, Cancel }

{ MessageBox visualiza la cadena indicada en una caja de diálogo de
  tamaño estándar. Antes de que el diálogo sea visualizado, Msg y
  Params se pasan a FormatStr. La cadena resultante se visualiza
  como una vista TStaticText en el diálogo. }
function MessageBox(const Msg: String; Params: Pointer;
  AOptions: Word): Word;
{ MessageBoxRec permite la especificación de un TRect que determina
  la ocupación de la caja de mensaje. }
function MessageBoxRect(var R: TRect; const Msg: String; Params: Pointer;
  AOptions: Word): Word;
{ InputBox visualiza un sencillo diálogo que permite al usuario
  introducir una cadena de caracteres. }
function InputBox(const Title, ALabel: String; var S: String;
  Limit: Byte): Word;
{ InputBoxRect es similar a InputBox pero permitir especificar
  un rectángulo. }
function InputBoxRect(var Bounds: TRect; const Title, ALabel: String;
  var S: String; Limit: Byte): Word;

```

IV1.1 UNIT Objects

```

const
{ Modos de acceso a TStream }
  stCreate   = $3C00;      { Crear nuevo fichero }
  stOpenRead = $3D00;      { Acceso de sólo lectura }
  stOpenWrite = $3D01;     { Acceso de sólo escritura }
  stOpen     = $3D02;     { Acceso de lectura y escritura }
{ Códigos de error de TStream }
  stOk       = 0;         { Sin error }
  stError    = -1;        { Error de acceso }
  stInitError = -2;       { No se puede inicializar stream }
  stReadError = -3;       { Lectura después de fin de stream }
  stWriteError = -4;      { No se puede expandir el stream }
  stGetError  = -5;       { Se quiere obtener un tipo de objeto no registrado }
  stPutError  = -6;       { Se quiere escribir un tipo de objeto no registrado }
{ Tamaño de TCollection máximo }
  MaxCollectionSize = 65520 div SizeOf(Pointer);
{ Códigos de error de TCollection }
  coIndexError = -1;      { Índice fuera de rango }
  coOverflow   = -2;      { Overflow }
{ Tamaño de cabecera VMT }
  vmtHeaderSize = 8;
type
{ Registros de conversión de Tipos }
  WordRec = record
    Lo, Hi: Byte;
  end;
  LongRec = record
    Lo, Hi: Word;
  end;
  PtrRec = record
    Ofs, Seg: Word;
  end;
{ Puntero a String }
  PString = ^String;
{ Tipo de conjuntos de caracteres }
  PCharSet = ^TCharSet;
  TCharSet = set of Char;
{ Arrays generales }
  PByteArray = ^TByteArray;
  TByteArray = array[0..32767] of Byte;
  PWordArray = ^TWordArray;
  TWordArray = array[0..16383] of Word;
{ Tipo objeto base TObject }
  PObject = ^TObject;
  TObject = object
    constructor Init;
    procedure Free;
    destructor Done; virtual;
  end;
{ TStreamRec }
  PStreamRec = ^TStreamRec;
  TStreamRec = record
    ObjType: Word;
    VmtLink: Word;
    Load: Pointer;
    Store: Pointer;
    Next: Word;
  end;
{ TStream }

```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

PStream = ^TStream;
TStream = object(TObject)
  Status: Integer;
  ErrorInfo: Integer;
  constructor Init;
  procedure CopyFrom(var S: TStream; Count: Longint);
  procedure Error(Code, Info: Integer); virtual;
  procedure Flush; virtual;
  function Get: PObject;
  function GetPos: Longint; virtual;
  function GetSize: Longint; virtual;
  procedure Put(P: PObject);
  procedure Read(var Buf; Count: Word); virtual;
  function ReadStr: PString;
  procedure Reset;
  procedure Seek(Pos: Longint); virtual;
  function StrRead: PChar;
  procedure StrWrite(P: PChar);
  procedure Truncate; virtual;
  procedure Write(var Buf; Count: Word); virtual;
  procedure WriteStr(P: PString);
end;
{ String para nombres de ficheros DOS }
{$IFDEF Windows}
  FNameStr = PChar;
{$ELSE}
  FNameStr = string[79];
{$ENDIF}
{ TDosStream }
PDosStream = ^TDosStream;
TDosStream = object(TStream)
  Handle: Word;
  constructor Init(FileName: FNameStr; Mode: Word);
  destructor Done; virtual;
  function GetPos: Longint; virtual;
  function GetSize: Longint; virtual;
  procedure Read(var Buf; Count: Word); virtual;
  procedure Seek(Pos: Longint); virtual;
  procedure Truncate; virtual;
  procedure Write(var Buf; Count: Word); virtual;
end;
{ TBufStream }
PBufStream = ^TBufStream;
TBufStream = object(TDosStream)
  Buffer: Pointer;
  BufSize: Word;
  BufPtr: Word;
  BufEnd: Word;
  constructor Init(FileName: FNameStr; Mode, Size: Word);
  destructor Done; virtual;
  procedure Flush; virtual;
  function GetPos: Longint; virtual;
  function GetSize: Longint; virtual;
  procedure Read(var Buf; Count: Word); virtual;
  procedure Seek(Pos: Longint); virtual;
  procedure Truncate; virtual;
  procedure Write(var Buf; Count: Word); virtual;
end;
{ TEmsStream }
PEmsStream = ^TEmsStream;
TEmsStream = object(TStream)
  Handle: Word;
  PageCount: Word;
  Size: Longint;
  Position: Longint;
  constructor Init(MinSize, MaxSize: Longint);
  destructor Done; virtual;

```

UNIT Objects

```
function GetPos: Longint; virtual;
function GetSize: Longint; virtual;
procedure Read(var Buf; Count: Word); virtual;
procedure Seek(Pos: Longint); virtual;
procedure Truncate; virtual;
procedure Write(var Buf; Count: Word); virtual;
end;
{ TMemoryStream }
PMemoryStream = ^TMemoryStream;
TMemoryStream = object(TStream)
  SegCount: Integer;
  SegList: PWordArray;
  CurSeg: Integer;
  BlockSize: Integer;
  Size: Longint;
  Position: Longint;
  constructor Init(ALimit: Longint; ABlockSize: Word);
  destructor Done; virtual;
  function GetPos: Longint; virtual;
  function GetSize: Longint; virtual;
  procedure Read(var Buf; Count: Word); virtual;
  procedure Seek(Pos: Longint); virtual;
  procedure Truncate; virtual;
  procedure Write(var Buf; Count: Word); virtual;
private
  function ChangeListSize(ALimit: Word): Boolean;
end;
{ Tipos de TCollection }
PItemList = ^TItemList;
TItemList = array[0..MaxCollectionSize - 1] of Pointer;
{ Tipo objeto TCollection }
PCollection = ^TCollection;
TCollection = object(TObject)
  Items: PItemList;
  Count: Integer;
  Limit: Integer;
  Delta: Integer;
  constructor Init(ALimit, ADelta: Integer);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  function At(Index: Integer): Pointer;
  procedure AtDelete(Index: Integer);
  procedure AtFree(Index: Integer);
  procedure AtInsert(Index: Integer; Item: Pointer);
  procedure AtPut(Index: Integer; Item: Pointer);
  procedure Delete(Item: Pointer);
  procedure DeleteAll;
  procedure Error(Code, Info: Integer); virtual;
  function FirstThat(Test: Pointer): Pointer;
  procedure ForEach(Action: Pointer);
  procedure Free(Item: Pointer);
  procedure FreeAll;
  procedure FreeItem(Item: Pointer); virtual;
  function GetItem(var S: TStream): Pointer; virtual;
  function IndexOf(Item: Pointer): Integer; virtual;
  procedure Insert(Item: Pointer); virtual;
  function LastThat(Test: Pointer): Pointer;
  procedure Pack;
  procedure PutItem(var S: TStream; Item: Pointer); virtual;
  procedure SetLimit(ALimit: Integer); virtual;
  procedure Store(var S: TStream);
end;
{ Tipo objeto TSortedCollection }
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

PSortedCollection = ^TSortedCollection;
TSortedCollection = object(TCollection)
  Duplicates: Boolean;
  constructor Init(ALimit, ADelta: Integer);
  constructor Load(var S: TStream);
  function Compare(Key1, Key2: Pointer): Integer; virtual;
  function IndexOf(Item: Pointer): Integer; virtual;
  procedure Insert(Item: Pointer); virtual;
  function KeyOf(Item: Pointer): Pointer; virtual;
  function Search(Key: Pointer; var Index: Integer): Boolean; virtual;
  procedure Store(var S: TStream);
end;
{ Tipo objeto TStringCollection }
PStringCollection = ^TStringCollection;
TStringCollection = object(TSortedCollection)
  function Compare(Key1, Key2: Pointer): Integer; virtual;
  procedure FreeItem(Item: Pointer); virtual;
  function GetItem(var S: TStream): Pointer; virtual;
  procedure PutItem(var S: TStream; Item: Pointer); virtual;
end;
{ Tipo objeto TStrCollection }
PStrCollection = ^TStrCollection;
TStrCollection = object(TSortedCollection)
  function Compare(Key1, Key2: Pointer): Integer; virtual;
  procedure FreeItem(Item: Pointer); virtual;
  function GetItem(var S: TStream): Pointer; virtual;
  procedure PutItem(var S: TStream; Item: Pointer); virtual;
end;
{$IFDEF Windows}
{ Tipo objeto TResourceCollection }
PResourceCollection = ^TResourceCollection;
TResourceCollection = object(TStringCollection)
  procedure FreeItem(Item: Pointer); virtual;
  function GetItem(var S: TStream): Pointer; virtual;
  function KeyOf(Item: Pointer): Pointer; virtual;
  procedure PutItem(var S: TStream; Item: Pointer); virtual;
end;
{ Tipo objeto TResourceFile }
PResourceFile = ^TResourceFile;
TResourceFile = object(TObject)
  Stream: PStream;
  Modified: Boolean;
  constructor Init(AStream: PStream);
  destructor Done; virtual;
  function Count: Integer;
  procedure Delete(Key: String);
  procedure Flush;
  function Get(Key: String): PObject;
  function KeyAt(I: Integer): String;
  procedure Put(Item: PObject; Key: String);
  function SwitchTo(AStream: PStream; Pack: Boolean): PStream;
private
  BasePos: Longint;
  IndexPos: Longint;
  Index: TResourceCollection;
end;
{ Tipo objeto TStringList }
TStrIndexRec = record
  Key, Count, Offset: Word;
end;
PStrIndex = ^TStrIndex;
TStrIndex = array[0..9999] of TStrIndexRec;

```

UNIT Objects

```
PStringList = ^TStringList;
TStringList = object(TObject)
  constructor Load(var S: TStream);
  destructor Done; virtual;
  function Get(Key: Word): String;
private
  Stream: PStream;
  BasePos: Longint;
  IndexSize: Integer;
  Index: PStrIndex;
  procedure ReadStr(var S: String; Offset, Skip: Word);
end;
{ Tipo objeto TStrListMaker }
PStrListMaker = ^TStrListMaker;
TStrListMaker = object(TObject)
  constructor Init(AStrSize, AIndexSize: Word);
  destructor Done; virtual;
  procedure Put(Key: Word; S: String);
  procedure Store(var S: TStream);
private
  StrPos: Word;
  StrSize: Word;
  Strings: PByteArray;
  IndexPos: Word;
  IndexSize: Word;
  Index: PStrIndex;
  Cur: TStrIndexRec;
  procedure CloseCurrent;
end;
{ Tipo objeto TPoint }
TPoint = object
  X, Y: Integer;
end;
{ Tipo objeto Rectangle }
TRect = object
  A, B: TPoint;
  procedure Assign(XA, YA, XB, YB: Integer);
  procedure Copy(R: TRect);
  procedure Move(ADX, ADY: Integer);
  procedure Grow(ADX, ADY: Integer);
  procedure Intersect(R: TRect);
  procedure Union(R: TRect);
  function Contains(P: TPoint): Boolean;
  function Equals(R: TRect): Boolean;
  function Empty: Boolean;
end;
{$ENDIF}
{ Rutinas de manejo de String Dinamico }
function NewStr(const S: String): PString;
procedure DisposeStr(P: PString);
{ Rutinas de Longint }
function LongMul(X, Y: Integer): Longint;
inline($5A/$58/$F7/$EA);
function LongDiv(X: Longint; Y: Integer): Integer;
inline($59/$58/$5A/$F7/$F9);
{ Rutinas de Stream }
procedure RegisterType(var S: TStreamRec);
{ Procedure de notificación Abstract }
procedure Abstract;
{ Estructuras Record para registrar objetos para su uso con Stream }
procedure RegisterObjects;
const
{ Procedure de error de Stream }
StreamError: Pointer = nil;
{ Variables de estado de EMS stream }
EmsCurHandle: Word = $FFFF;
EmsCurPage: Word = $FFFF;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```
{ Registros para registrar Stream }
const
  RCollection: TStreamRec = (
    ObjType: 50;
    VmtLink: Ofs(KindOf(TCollection)^);
    Load: @TCollection.Load;
    Store: @TCollection.Store);
const
  RStringCollection: TStreamRec = (
    ObjType: 51;
    VmtLink: Ofs(KindOf(TStringCollection)^);
    Load: @TStringCollection.Load;
    Store: @TStringCollection.Store);
const
  RStrCollection: TStreamRec = (
    ObjType: 69;
    VmtLink: Ofs(KindOf(TStrCollection)^);
    Load: @TStrCollection.Load;
    Store: @TStrCollection.Store);
{$IFDEF Windows }
const
  RStringList: TStreamRec = (
    ObjType: 52;
    VmtLink: Ofs(KindOf(TStringList)^);
    Load: @TStringList.Load;
    Store: nil);
const
  RStrListMaker: TStreamRec = (
    ObjType: 52;
    VmtLink: Ofs(KindOf(TStrListMaker)^);
    Load: nil;
    Store: @TStrListMaker.Store);
{$ENDIF}
```

IV1.1 UNIT Outline

```
uses Objects, Drivers, Views;
const
  ovExpanded = $01;
  ovChildren = $02;
  ovLast = $04;
const
  cmOutlineItemSelected = 301;
const
  COutlineViewer = CScroller + #8#8;
type
{ Tipo objeto TOutlineViewer }
{ Distribución de la Paleta }
{ 1 = Color normal }
{ 2 = Color foco }
{ 3 = Color selección }
{ 4 = Color no expandible }
POutlineViewer = ^TOutlineViewer;
TOutlineViewer = object(TScroller)
  Foc: Integer;
  constructor Init(var Bounds: TRect; AHScrollBar,
    AVScrollBar: PScrollBar);
  constructor Load(var S: TStream);
  procedure Adjust(Node: Pointer; Expand: Boolean); virtual;
  function CreateGraph(Level: Integer; Lines: LongInt; Flags: Word;
    LevWidth, EndWidth: Integer; const Chars: String): String;
  procedure Draw; virtual;
  procedure ExpandAll(Node: Pointer);
  function FirstThat(Test: Pointer): Pointer;
  procedure Focused(I: Integer); virtual;
```


UNIT Outline

```
function ForEach(Action: Pointer): Pointer;
function GetChild(Node: Pointer; I: Integer): Pointer; virtual;
function GetGraph(Level: Integer; Lines: LongInt; Flags: Word): String;
virtual;
function GetNumChildren(Node: Pointer): Integer; virtual;
function GetNode(I: Integer): Pointer;
function GetPalette: PPalette; virtual;
function GetRoot: Pointer; virtual;
function GetText(Node: Pointer): String; virtual;
procedure HandleEvent(var Event: TEvent); virtual;
function HasChildren(Node: Pointer): Boolean; virtual;
function IsExpanded(Node: Pointer): Boolean; virtual;
function IsSelected(I: Integer): Boolean; virtual;
procedure Selected(I: Integer); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Store(var S: TStream);
procedure Update;
private
  procedure AdjustFocus(NewFocus: Integer);
  function Iterate(Action: Pointer; CallerFrame: Word;
    CheckRslt: Boolean): Pointer;
end;
{ TNode }
PNode = ^TNode;
TNode = record
  Next: PNode;
  Text: PString;
  ChildList: PNode;
  Expanded: Boolean;
end;
{ Tipo objeto TOutline }
{ Distribución de la Paleta }
{ 1 = Color normal }
{ 2 = Color foco }
{ 3 = Color selección }

POutline = ^TOutline;
TOutline = object(TOutlineViewer)
  Root: PNode;
  constructor Init(var Bounds: TRect; AHScrollBar,
    AVScrollBar: PScrollBar; ARoot: PNode);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Adjust(Node: Pointer; Expand: Boolean); virtual;
  function GetRoot: Pointer; virtual;
  function GetNumChildren(Node: Pointer): Integer; virtual;
  function GetChild(Node: Pointer; I: Integer): Pointer; virtual;
  function GetText(Node: Pointer): String; virtual;
  function IsExpanded(Node: Pointer): Boolean; virtual;
  function HasChildren(Node: Pointer): Boolean; virtual;
  procedure Store(var S: TStream);
end;
const
  ROutline: TStreamRec = (
    ObjType: 91;
    VmtLink: Ofs(KindOf(TOutline));
    Load: @TOutline.Load;
    Store: @TOutline.Store
  );
procedure RegisterOutline;
function NewNode(const AText: String; AChildren, ANext: PNode): PNode;
procedure DisposeNode(Node: PNode);
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

IV1.1 UNIT *StdDlg*

```

uses Objects, Drivers, Views, Dialogs, Dos;
const
{ Comandos }
cmFileOpen   = 800; { Devuelto desde TFileDialog cuando se pulsa Open }
cmFileReplace = 801; { Devuelto desde TFileDialog cuando se pulsa Replace }
cmFileClear  = 802; { Devuelto desde TFileDialog cuando se pulsa Clear }
cmFileInit   = 803; { Utilizado por TFileDialog internamente }
cmChDir      = 804; { Utilizado por TChDirDialog internamente }
cmRevert     = 805; { Utilizado por TChDirDialog internamente }
{ Mensajes }
cmFileFocused = 806; { Un nuevo fichero ha obtenido el foco en TFileList }
cmFileDoubleClicked = 807; { Un nuevo fichero se ha seleccionado en TFileList }
type
{ TSearchRec }
{ Record utilizado para guardar la información de directorio por TFileDialog }
TSearchRec = record
  Attr: Byte;
  Time: Longint;
  Size: Longint;
  Name: string[12];
end;
type
{ TFileInputLine es una línea de entrada (input line) especial }
{ utilizado por TFileDialog que actualiza sus contenidos en }
{ respuesta a un comando cmFileFocused procedente de TFileList. }
PFileInputLine = ^TFileInputLine;
TFileInputLine = object(TInputLine)
  constructor Init(var Bounds: TRect; AMaxLen: Integer);
  procedure HandleEvent(var Event: TEvent); virtual;
end;
{ TFileCollection es una colección de TSearchRec's. }
PFileCollection = ^TFileCollection;
TFileCollection = object(TSortedCollection)
  function Compare(Key1, Key2: Pointer): Integer; virtual;
  procedure FreeItem(Item: Pointer); virtual;
  function GetItem(var S: TStream): Pointer; virtual;
  procedure PutItem(var S: TStream; Item: Pointer); virtual;
end;
{ TSortedListBox es un TListBox que contiene una TStoredCollection }
{ en lugar de una simple TCollection. Permite realizar una }
{ búsqueda incremental sobre los contenidos. }
PSortedListBox = ^TSortedListBox;
TSortedListBox = object(TListBox)
  SearchPos: Word;
  ShiftState: Byte;
  constructor Init(var Bounds: TRect; ANumCols: Word;
    AScrollBar: PScrollBar);
  procedure HandleEvent(var Event: TEvent); virtual;
  function GetKey(var S: String): Pointer; virtual;
  procedure NewList(AList: PCollection); virtual;
end;
{ TFileList es una TSortedListBox que contiene una TFileCollection como }
{ tipo de colección. Comunica con mensajes de difusión (broadcast) a }
{ TFileInput y TInfoPane que fichero es seleccionado. }
PFileList = ^TFileList;
TFileList = object(TSortedListBox)
  constructor Init(var Bounds: TRect; AScrollBar: PScrollBar);
  destructor Done; virtual;
  function DataSize: Word; virtual;
  procedure FocusItem(Item: Integer); virtual;
  procedure GetData(var Rec); virtual;
  function GetText(Item: Integer; MaxLen: Integer): String; virtual;
  function GetKey(var S: String): Pointer; virtual;

```

UNIT StdDlg

```

    procedure HandleEvent(var Event: TEvent); virtual;
    procedure ReadDirectory(AWildcard: PathStr);
    procedure SetData(var Rec); virtual;
end;
{ TFileInfoPane es una TView que visualiza la información sobre el fichero }
{ seleccionado en cada momento en la TFileList de un TFileDialog. }
PFileInfoPane = ^TFileInfoPane;
TFileInfoPane = object(TView)
    S: TSearchRec;
    constructor Init(var Bounds: TRect);
    procedure Draw; virtual;
    function GetPalette: PPalette; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
end;
{ TFileDialog es un diálogo de entrada de nombre de fichero estándar }
TWildStr = PathStr;
const
    fdOkButton      = $0001;      { Poner un botón OK en el dialogo }
    fdOpenButton    = $0002;      { Poner un botón Open en el dialogo }
    fdReplaceButton = $0004;      { Poner un botón Replace en el dialogo }
    fdClearButton   = $0008;      { Poner un botón Clear en el dialogo }
    fdHelpButton    = $0010;      { Poner un botón en el dialogo }
    fdNoLoadDir     = $0100;      { No cargar el contenido del directorio }
                                { activo en el diálogo en Init. Se supone }
                                { que se pretende cambiar los caracteres }
                                { comodín usando SetData o almacenando }
                                { el diálogo en un stream. }
type
    PFileDialog = ^TFileDialog;
    TFileDialog = object(TDialog)
        FileName: PFileInputLine;
        FileList: PFileList;
        WildCard: TWildStr;
        Directory: PString;
        constructor Init(AWildcard: TWildStr; const ATitle,
            InputName: String; AOptions: Word; HistoryId: Byte);
        constructor Load(var S: TStream);
        destructor Done; virtual;
        procedure GetData(var Rec); virtual;
        procedure GetFileName(var S: PathStr);
        procedure HandleEvent(var Event: TEvent); virtual;
        procedure SetData(var Rec); virtual;
        procedure Store(var S: TStream);
        function Valid(Command: Word): Boolean; virtual;
    private
        procedure ReadDirectory;
    end;
    { TDirEntry }
    PDirEntry = ^TDirEntry;
    TDirEntry = record
        DisplayText: PString;
        Directory: PString;
    end;
    { TDirCollection es una colección de TDirEntry's usada por TDirListBox. }
    PDirCollection = ^TDirCollection;
    TDirCollection = object(TCollection)
        function GetItem(var S: TStream): Pointer; virtual;
        procedure FreeItem(Item: Pointer); virtual;
        procedure PutItem(var S: TStream; Item: Pointer); virtual;
    end;
    { TDirListBox visualiza un árbol de directorios dentro de un TChDirDialog. }

```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

PDirListBox = ^TDirListBox;
TDirListBox = object(TListBox)
  Dir: DirStr;
  Cur: Word;
  constructor Init(var Bounds: TRect; AScrollBar: PScrollBar);
  destructor Done; virtual;
  function GetText(Item: Integer; MaxLen: Integer): String; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  function IsSelected(Item: Integer): Boolean; virtual;
  procedure NewDirectory(var ADir: DirStr);
  procedure SetState(AState: Word; Enable: Boolean); virtual;
end;
{ TChDirDialog es un diálogo de cambio de directorio estándar. }
const
  cdNormal      = $0000; { Opción para usar un diálogo inmediatamente }
  cdNoLoadDir   = $0001; { Opción para inicializar el dialogo sin el }
                        { contenido de directorio, usado cuando este }
                        { se va almacenar en un stream. }
  cdHelpButton  = $0002; { Para poner un botón de help en el diálogo. }
type
  PChDirDialog = ^TChDirDialog;
  TChDirDialog = object(TDialog)
    DirInput: PInputLine;
    DirList: PDirListBox;
    OkButton: PButton;
    ChDirButton: PButton;
    constructor Init(AOptions: Word; HistoryId: Word);
    constructor Load(var S: TStream);
    function DataSize: Word; virtual;
    procedure GetData(var Rec); virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure SetData(var Rec); virtual;
    procedure Store(var S: TStream);
    function Valid(Command: Word): Boolean; virtual;
  private
    procedure SetUpDialog;
  end;
const
  CInfoPane = #30;
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RFileInputLine: TStreamRec = (
    ObjType: 60;
    VmtLink: Ofs(KindOf(TFileInputLine)^);
    Load:   @TFileInputLine.Load;
    Store:   @TFileInputLine.Store
  );
const
  RFileCollection: TStreamRec = (
    ObjType: 61;
    VmtLink: Ofs(KindOf(TFileCollection)^);
    Load:   @TFileCollection.Load;
    Store:   @TFileCollection.Store
  );
const
  RFileList: TStreamRec = (
    ObjType: 62;
    VmtLink: Ofs(KindOf(TFileList)^);
    Load:   @TFileList.Load;
    Store:   @TFileList.Store
  );

```

UNIT *TextView*

```
const
  RFileInfoPane: TStreamRec = (
    ObjType: 63;
    VmtLink: Ofs(KindOf(TFileInfoPane)^);
    Load:   @TFileInfoPane.Load;
    Store:   @TFileInfoPane.Store
  );
const
  RFileDialog: TStreamRec = (
    ObjType: 64;
    VmtLink: Ofs(KindOf(TFileDialog)^);
    Load:   @TFileDialog.Load;
    Store:   @TFileDialog.Store
  );
const
  RDirCollection: TStreamRec = (
    ObjType: 65;
    VmtLink: Ofs(KindOf(TDirCollection)^);
    Load:   @TDirCollection.Load;
    Store:   @TDirCollection.Store
  );
const
  RDirListBox: TStreamRec = (
    ObjType: 66;
    VmtLink: Ofs(KindOf(TDirListBox)^);
    Load:   @TDirListBox.Load;
    Store:   @TDirListBox.Store
  );
const
  RChDirDialog: TStreamRec = (
    ObjType: 67;
    VmtLink: Ofs(KindOf(TChDirDialog)^);
    Load:   @TChDirDialog.Load;
    Store:   @TChDirDialog.Store
  );
const
  RSortedListBox: TStreamRec = (
    ObjType: 68;
    VmtLink: Ofs(KindOf(TSortedListBox)^);
    Load:   @TSortedListBox.Load;
    Store:   @TSortedListBox.Store
  );
procedure RegisterStdDlg;
```

IV1.1 UNIT *TextView*

```
uses Objects, Drivers, Views, Dos;
type
  { TTextDevice }
  PTextDevice = ^TTextDevice;
  TTextDevice = object(TScroller)
    function StrRead(var S: TextBuf): Byte; virtual;
    procedure StrWrite(var S: TextBuf; Count: Byte); virtual;
  end;
  { TTerminal }
  PTerminalBuffer = ^TTerminalBuffer;
  TTerminalBuffer = array[0..65534] of Char;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```
PTerminal = ^TTerminal;
TTerminal = object(TTextDevice)
  BufSize: Word;
  Buffer: PTerminalBuffer;
  QueFront, QueBack: Word;
  constructor Init(var Bounds:TRect; AHScrollBar, AVScrollBar: PScrollBar;
    ABufSize: Word);
  destructor Done; virtual;
  procedure BufDec(var Val: Word);
  procedure BufInc(var Val: Word);
  function CalcWidth: Integer;
  function CanInsert(Amount: Word): Boolean;
  procedure Draw; virtual;
  function NextLine(Pos:Word): Word;
  function PrevLines(Pos:Word; Lines: Word): Word;
  function StrRead(var S: TextBuf): Byte; virtual;
  procedure StrWrite(var S: TextBuf; Count: Byte); virtual;
  function QueEmpty: Boolean;
end;
procedure AssignDevice(var T: Text; Screen: PTextDevice);
```

IV1.1 UNIT *Validate*

```
uses Objects;
const
{ Constantes de estado de TValidator }
vsOk      = 0;
vsSyntax  = 1;      { Error en la sintaxis en un TPXPictureValidator
                    o en un TDBPictureValidator }
{ Flags de opciones de Validator }
voFill    = $0001;
voTransfer = $0002;
voOnAppend = $0004;
voReserved = $00F8;
{ Constantes de TVTransfer }
type
TVTransfer = (vtDataSize, vtSetData, vtGetData);
{ Objecto abstracto TValidator }
PValidator = ^TValidator;
TValidator = object(TObject)
  Status: Word;
  Options: Word;
  constructor Init;
  constructor Load(var S: TStream);
  procedure Error; virtual;
  function IsValidInput(var S: string;
    SuppressFill: Boolean): Boolean; virtual;
  function IsValid(const S: string): Boolean; virtual;
  procedure Store(var S: TStream);
  function Transfer(var S: String; Buffer: Pointer;
    Flag: TVTransfer): Word; virtual;
  function Valid(const S: string): Boolean;
end;
{ Tipo de resultados de TPXPictureValidator }
TPicResult = (prComplete, prIncomplete, prEmpty, prError, prSyntax,
  prAmbiguous, prIncompNoFill);
{ TPXPictureValidator }
PPXPictureValidator = ^TPXPictureValidator;
TPXPictureValidator = object(TValidator)
  Pic: PString;
  constructor Init(const APic: string; AutoFill: Boolean);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Error; virtual;
```

UNIT Validate

```

function IsValidInput(var S: string;
  SuppressFill: Boolean): Boolean; virtual;
function IsValid(const S: string): Boolean; virtual;
function Picture(var Input: string;
  AutoFill: Boolean): TPicResult; virtual;
procedure Store(var S: TStream);
end;
{ TFilterValidator }
PFilterValidator = ^TFilterValidator;
TFilterValidator = object(TValidator)
  ValidChars: TCharSet;
  constructor Init(AValidChars: TCharSet);
  constructor Load(var S: TStream);
  procedure Error; virtual;
  function IsValid(const S: string): Boolean; virtual;
  function IsValidInput(var S: string;
    SuppressFill: Boolean): Boolean; virtual;
  procedure Store(var S: TStream);
end;
{ TRangeValidator }
PRangeValidator = ^TRangeValidator;
TRangeValidator = object(TFilterValidator)
  Min, Max: LongInt;
  constructor Init(AMin, AMax: LongInt);
  constructor Load(var S: TStream);
  procedure Error; virtual;
  function IsValid(const S: string): Boolean; virtual;
  procedure Store(var S: TStream);
  function Transfer(var S: String; Buffer: Pointer;
    Flag: TVTransfer): Word; virtual;
end;
{ TLookupValidator }
PLookupValidator = ^TLookupValidator;
TLookupValidator = object(TValidator)
  function IsValid(const S: string): Boolean; virtual;
  function Lookup(const S: string): Boolean; virtual;
end;
{ TStringLookupValidator }
PStringLookupValidator = ^TStringLookupValidator;
TStringLookupValidator = object(TLookupValidator)
  Strings: PStringCollection;
  constructor Init(AStrings: PStringCollection);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Error; virtual;
  function Lookup(const S: string): Boolean; virtual;
  procedure NewStringList(AStrings: PStringCollection);
  procedure Store(var S: TStream);
end;
{ Procedure de registro de Validate }
procedure RegisterValidate;
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RPXPictureValidator: TStreamRec = (
    ObjType: 80;
    VmtLink: ofs(TypeOf(TPXPictureValidator)^);
    Load: @TPXPictureValidator.Load;
    Store: @TPXPictureValidator.Store
  );
const
  RFilterValidator: TStreamRec = (
    ObjType: 81;
    VmtLink: ofs(TypeOf(TFilterValidator)^);
    Load: @TFilterValidator.Load;
    Store: @TFilterValidator.Store
  );

```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```
const
  RRangeValidator: TStreamRec = (
    ObjType: 82;
    VmtLink: Ofs(KindOf(TRangeValidator)^);
    Load: @TRangeValidator.Load;
    Store: @TRangeValidator.Store
  );
const
  RStringLookupValidator: TStreamRec = (
    ObjType: 83;
    VmtLink: Ofs(KindOf(TStringLookupValidator)^);
    Load: @TStringLookupValidator.Load;
    Store: @TStringLookupValidator.Store
  );
```

IV1.1 UNIT Views

```
uses Objects, Drivers, Memory;
const
{ Máscaras de estado de TView }
  sfVisible      = $0001;
  sfCursorVis    = $0002;
  sfCursorIns    = $0004;
  sfShadow       = $0008;
  sfActive       = $0010;
  sfSelected     = $0020;
  sfFocused      = $0040;
  sfDragging     = $0080;
  sfDisabled     = $0100;
  sfModal        = $0200;
  sfDefault      = $0400;
  sfExposed      = $0800;
{ Máscaras de opciones de TView }
  ofSelectable   = $0001;
  ofTopSelect    = $0002;
  ofFirstClick   = $0004;
  ofFramed       = $0008;
  ofPreProcess   = $0010;
  ofPostProcess  = $0020;
  ofBuffered     = $0040;
  ofTileable     = $0080;
  ofCenterX     = $0100;
  ofCenterY     = $0200;
  ofCentered    = $0300;
  ofValidate     = $0400;
  ofVersion      = $3000;
  ofVersion10   = $0000;
  ofVersion20   = $1000;
{ Máscaras de GrowMode de TView }
  gfGrowLoX     = $01;
  gfGrowLoY     = $02;
  gfGrowHiX     = $04;
  gfGrowHiY     = $08;
  gfGrowAll     = $0F;
  gfGrowRel     = $10;
{ Máscaras de DragMode de TView }
  dmDragMove    = $01;
  dmDragGrow    = $02;
  dmLimitLoX    = $10;
  dmLimitLoY    = $20;
  dmLimitHiX    = $40;
  dmLimitHiY    = $80;
  dmLimitAll    = $F0;
```


UNIT Views

```
{ Códigos de contexto de TView }
hcNoContext = 0;
hcDragging = 1;
{ Códigos de componentes de TScrollBar }
sbLeftArrow = 0;
sbRightArrow = 1;
sbPageLeft = 2;
sbPageRight = 3;
sbUpArrow = 4;
sbDownArrow = 5;
sbPageUp = 6;
sbPageDown = 7;
sbIndicator = 8;
{ Opciones de TScrollBar para TWindow.StandardScrollBar }
sbHorizontal = $0000;
sbVertical = $0001;
sbHandleKeyboard = $0002;
{ Máscaras de Flags de TWindow }
wfMove = $01;
wfGrow = $02;
wfClose = $04;
wfZoom = $08;
{ Constantes de números de TWindow }
wnNoNumber = 0;
{ Entradas de la paleta de TWindow }
wpBlueWindow = 0;
wpCyanWindow = 1;
wpGrayWindow = 2;
{ Códigos de comandos estándar }
cmValid = 0;
cmQuit = 1;
cmError = 2;
cmMenu = 3;
cmClose = 4;
cmZoom = 5;
cmResize = 6;
cmNext = 7;
cmPrev = 8;
cmHelp = 9;
{ Códigos de comandos de Application }
cmCut = 20;
cmCopy = 21;
cmPaste = 22;
cmUndo = 23;
cmClear = 24;
cmTile = 25;
cmCascade = 26;
{ Comandos estándar de TDialog }
cmOK = 10;
cmCancel = 11;
cmYes = 12;
cmNo = 13;
cmDefault = 14;
{ Mensajes estándar }
cmReceivedFocus = 50;
cmReleasedFocus = 51;
cmCommandSetChanged = 52;
{ Mensajes de TScrollBar }
cmScrollBarChanged = 53;
cmScrollBarClicked = 54;
{ Mensajes de selección de TWindow }
cmSelectWindowNum = 55;
{ Mensajes de TListViewer }
cmListItemSelected = 56;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

{ Paletas de color }
CFrame      = #1#1#2#2#3;
CScrollBar  = #4#5#5;
CScroller   = #6#7;
CListViewer = #26#26#27#28#29;
CBlueWindow = #8#9#10#11#12#13#14#15;
CCyanWindow = #16#17#18#19#20#21#22#23;
CGrayWindow = #24#25#26#27#28#29#30#31;
{ Ancho máximo del buffer de dibujo de vista TDrawBuffer }
MaxViewWidth = 132;
type
{ Conjunto de comandos }
PCommandSet = ^TCommandSet;
TCommandSet = set of Byte;
{ Tipo paleta de color }
PPalette = ^TPalette;
TPalette = String;
{ TDrawBuffer, buffer usado por los métodos de dibujo (Draw) }
TDrawBuffer = array[0..MaxViewWidth - 1] of Word;
{ Puntero de objetos TView }
PView = ^TView;
{ Puntero a objetos TGroup }
PGroup = ^TGroup;
{ Tipo objeto TView }
TView = object(TObject)
  Owner: PGroup;
  Next: PView;
  Origin: TPoint;
  Size: TPoint;
  Cursor: TPoint;
  GrowMode: Byte;
  DragMode: Byte;
  HelpCtx: Word;
  State: Word;
  Options: Word;
  EventMask: Word;
  constructor Init(var Bounds: TRect);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Awaken; virtual;
  procedure BlockCursor;
  procedure CalcBounds(var Bounds: TRect; Delta: TPoint); virtual;
  procedure ChangeBounds(var Bounds: TRect); virtual;
  procedure ClearEvent(var Event: TEvent);
  function CommandEnabled(Command: Word): Boolean;
  function DataSize: Word; virtual;
  procedure DisableCommands(Commands: TCommandSet);
  procedure DragView(Event: TEvent; Mode: Byte;
    var Limits: TRect; MinSize, MaxSize: TPoint);
  procedure Draw; virtual;
  procedure DrawView;
  procedure EnableCommands(Commands: TCommandSet);
  procedure EndModal(Command: Word); virtual;
  function EventAvail: Boolean;
  function Execute: Word; virtual;
  function Exposed: Boolean;
  function Focus: Boolean;
  procedure GetBounds(var Bounds: TRect);
  procedure GetClipRect(var Clip: TRect);
  function GetColor(Color: Word): Word;
  procedure GetCommands(var Commands: TCommandSet);
  procedure GetData(var Rec); virtual;
  procedure GetEvent(var Event: TEvent); virtual;
  procedure GetExtent(var Extent: TRect);
  function GetHelpCtx: Word; virtual;
  function GetPalette: PPalette; virtual;
  procedure GetPeerViewPtr(var S: TStream; var P);

```

UNIT Views

```
function GetState(AState: Word): Boolean;
procedure GrowTo(X, Y: Integer);
procedure HandleEvent(var Event: TEvent); virtual;
procedure Hide;
procedure HideCursor;
procedure KeyEvent(var Event: TEvent);
procedure Locate(var Bounds: TRect);
procedure MakeFirst;
procedure MakeGlobal(Source: TPoint; var Dest: TPoint);
procedure MakeLocal(Source: TPoint; var Dest: TPoint);
function MouseEvent(var Event: TEvent; Mask: Word): Boolean;
function MouseInView(Mouse: TPoint): Boolean;
procedure MoveTo(X, Y: Integer);
function NextView: PView;
procedure NormalCursor;
function Prev: PView;
function PrevView: PView;
procedure PutEvent(var Event: TEvent); virtual;
procedure PutInFrontOf(Target: PView);
procedure PutPeerViewPtr(var S: TStream; P: PView);
procedure Select;
procedure SetBounds(var Bounds: TRect);
procedure SetCommands(Commands: TCommandSet);
procedure SetCmdState(Commands: TCommandSet; Enable: Boolean);
procedure SetCursor(X, Y: Integer);
procedure SetData(var Rec); virtual;
procedure SetState(AState: Word; Enable: Boolean); virtual;
procedure Show;
procedure ShowCursor;
procedure SizeLimits(var Min, Max: TPoint); virtual;
procedure Store(var S: TStream);
function TopView: PView;
function Valid(Command: Word): Boolean; virtual;
procedure WriteBuf(X, Y, W, H: Integer; var Buf);
procedure WriteChar(X, Y: Integer; C: Char; Color: Byte;
  Count: Integer);
procedure WriteLine(X, Y, W, H: Integer; var Buf);
procedure WriteStr(X, Y: Integer; Str: String; Color: Byte);
private
  procedure DrawCursor;
  procedure DrawHide>LastView: PView);
  procedure DrawShow>LastView: PView);
  procedure DrawUnderRect(var R: TRect; LastView: PView);
  procedure DrawUnderView(DoShadow: Boolean; LastView: PView);
  procedure ResetCursor; virtual;
end;
{ Tipos TFrame }
TTitleStr = string[80];
{ Tipo objeto TFrame }
{ Distribución de la Paleta }
{ 1 = Marco pasivo }
{ 2 = Título pasivo }
{ 3 = Marco activo }
{ 4 = Título activo }
{ 5 = Iconos }
PFrame = ^TFrame;
TFrame = object(TView)
  constructor Init(var Bounds: TRect);
  procedure Draw; virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure SetState(AState: Word; Enable: Boolean); virtual;
private
  FrameMode: Word;
  procedure FrameLine(var FrameBuf; Y, N: Integer; Color: Byte);
end;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

{ Caracteres de Barra de desplazamiento }
TScrollChars = array[0..4] of Char;
{ Tipo objeto TScrollBar }
{ Distribucción de Paleta }
{ 1 = Areas de página }
{ 2 = Flechas }
{ 3 = Indicador }
PScrollBar = ^TScrollBar;
TScrollBar = object(TView)
  Value: Integer;
  Min: Integer;
  Max: Integer;
  PgStep: Integer;
  ArStep: Integer;
  constructor Init(var Bounds: TRect);
  constructor Load(var S: TStream);
  procedure Draw; virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure ScrollDraw; virtual;
  function ScrollStep(Part: Integer): Integer; virtual;
  procedure SetParams(AValue, AMin, AMax, APgStep, AArStep: Integer);
  procedure SetRange(AMin, AMax: Integer);
  procedure SetStep(APgStep, AArStep: Integer);
  procedure SetValue(AValue: Integer);
  procedure Store(var S: TStream);
private
  Chars: TScrollChars;
  procedure DrawPos(Pos: Integer);
  function GetPos: Integer;
  function GetSize: Integer;
end;
{ Tipo objeto TScroller }
{ Distribucción de la Paleta }
{ 1 = Texto normal }
{ 2 = Texto seleccionado }
PScroller = ^TScroller;
TScroller = object(TView)
  HScrollBar: PScrollBar;
  VScrollBar: PScrollBar;
  Delta: TPoint;
  Limit: TPoint;
  constructor Init(var Bounds: TRect; AHScrollBar, AVScrollBar: PScrollBar);
  constructor Load(var S: TStream);
  procedure ChangeBounds(var Bounds: TRect); virtual;
  function GetPalette: PPalette; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure ScrollDraw; virtual;
  procedure ScrollTo(X, Y: Integer);
  procedure SetLimit(X, Y: Integer);
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure Store(var S: TStream);
private
  DrawLock: Byte;
  DrawFlag: Boolean;
  procedure CheckDraw;
end;
{ TListViewer }
{ Distribucción de la Paleta }
{ 1 = Activo }
{ 2 = Inactivo }
{ 3 = Con foco (focused) }
{ 4 = Seleccionado }
{ 5 = Divisor }
PListViewer = ^TListViewer;

```

UNIT Views

```
TListViewer = object(TView)
  HScrollBar: PScrollBar;
  VScrollBar: PScrollBar;
  NumCols: Integer;
  TopItem: Integer;
  Focused: Integer;
  Range: Integer;
  constructor Init(var Bounds: TRect; ANumCols: Word;
    AHScrollBar, AVScrollBar: PScrollBar);
  constructor Load(var S: TStream);
  procedure ChangeBounds(var Bounds: TRect); virtual;
  procedure Draw; virtual;
  procedure FocusItem(Item: Integer); virtual;
  function GetPalette: PPalette; virtual;
  function GetText(Item: Integer; MaxLen: Integer): String; virtual;
  function IsSelected(Item: Integer): Boolean; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure SelectItem(Item: Integer); virtual;
  procedure SetRange(ARange: Integer);
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure Store(var S: TStream);
private
  procedure FocusItemNum(Item: Integer); virtual;
end;
{ Buffer de video }
PVideoBuf = ^TVideoBuf;
TVideoBuf = array[0..3999] of Word;
{ Modos de selección }
SelectMode = (NormalSelect, EnterSelect, LeaveSelect);
{ Tipo objeto TGroup }
TGroup = object(TView)
  Last: PView;
  Current: PView;
  Phase: (phFocused, phPreProcess, phPostProcess);
  Buffer: PVideoBuf;
  EndState: Word;
  constructor Init(var Bounds: TRect);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Awaken; virtual;
  procedure ChangeBounds(var Bounds: TRect); virtual;
  function DataSize: Word; virtual;
  procedure Delete(P: PView);
  procedure Draw; virtual;
  procedure EndModal(Command: Word); virtual;
  procedure EventError(var Event: TEvent); virtual;
  function ExecView(P: PView): Word;
  function Execute: Word; virtual;
  function First: PView;
  function FirstThat(P: Pointer): PView;
  function FocusNext(Forwards: Boolean): Boolean;
  procedure ForEach(P: Pointer);
  procedure GetData(var Rec); virtual;
  function GetHelpCtx: Word; virtual;
  procedure GetSubViewPtr(var S: TStream; var P);
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure Insert(P: PView);
  procedure InsertBefore(P, Target: PView);
  procedure Lock;
  procedure PutSubViewPtr(var S: TStream; P: PView);
  procedure Redraw;
  procedure SelectNext(Forwards: Boolean);
  procedure SetData(var Rec); virtual;
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure Store(var S: TStream);
  procedure Unlock;
  function Valid(Command: Word): Boolean; virtual;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

```

private
  Clip: TRect;
  LockFlag: Byte;
  function At(Index: Integer): PView;
  procedure DrawSubViews(P, Bottom: PView);
  function FirstMatch(AState: Word; AOptions: Word): PView;
  function FindNext(Forwards: Boolean): PView;
  procedure FreeBuffer;
  procedure GetBuffer;
  function IndexOf(P: PView): Integer;
  procedure InsertView(P, Target: PView);
  procedure RemoveView(P: PView);
  procedure ResetCurrent;
  procedure ResetCursor; virtual;
  procedure SetCurrent(P: PView; Mode: SelectMode);
end;
{ Tipo objeto TWindow }
{ Distribución de la Paleta }
{ 1 = Marco pasivo }
{ 2 = Marco activo }
{ 3 = Icono de marco }
{ 4 = Area de página de la barra de scroll }
{ 5 = Controles de la barra de scroll }
{ 6 = Texto normal del desplazador }
{ 7 = Texto seleccionado del desplazador }
{ 8 = Reservado }
PWindow = ^TWindow;
TWindow = object(TGroup)
  Flags: Byte;
  ZoomRect: TRect;
  Number: Integer;
  Palette: Integer;
  Frame: PFrame;
  Title: PString;
  constructor Init(var Bounds: TRect; ATitle: TTitleStr; ANumber: Integer);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  procedure Close; virtual;
  function GetPalette: PPalette; virtual;
  function GetTitle(MaxSize: Integer): TTitleStr; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure InitFrame; virtual;
  procedure SetState(AState: Word; Enable: Boolean); virtual;
  procedure SizeLimits(var Min, Max: TPoint); virtual;
  function StandardScrollBar(AOptions: Word): PScrollBar;
  procedure Store(var S: TStream);
  procedure Zoom; virtual;
end;
{ Función de encaminamiento (dispatch) de mensajes }
function Message(Receiver: PView; What, Command: Word;
  InfoPtr: Pointer): Pointer;
{ Procedure de registro de Views }
procedure RegisterViews;
const
{ Máscaras de eventos }
  PositionalEvents: Word = evMouse;
  FocusedEvents: Word = evKeyboard + evCommand;
{ Tamaño de ventana mínimo }
  MinWinSize: TPoint = (X: 16; Y: 6);
{ Definiciones de sombras }
  ShadowSize: TPoint = (X: 2; Y: 1);
  ShadowAttr: Byte = $08;
{ Control de marcas }
  ShowMarkers: Boolean = False;
{ Valor de retorno de error de MapColor }
  ErrorAttr: Byte = $CF;

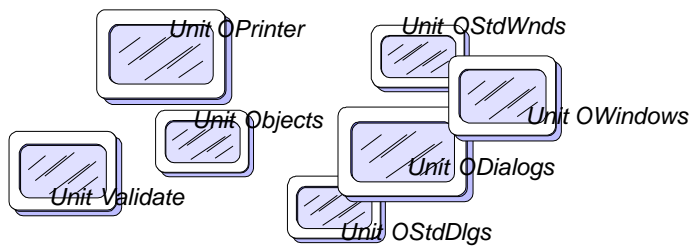
```

UNIT Views

```
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RView: TStreamRec = (
    ObjType: 1;
    VmtLink: Ofs(KindOf(TView)^);
    Load:   @TView.Load;
    Store:   @TView.Store
  );
const
  RFrame: TStreamRec = (
    ObjType: 2;
    VmtLink: Ofs(KindOf(TFrame)^);
    Load:   @TFrame.Load;
    Store:   @TFrame.Store
  );
const
  RScrollBar: TStreamRec = (
    ObjType: 3;
    VmtLink: Ofs(KindOf(TScrollBar)^);
    Load:   @TScrollBar.Load;
    Store:   @TScrollBar.Store
  );
const
  RScroller: TStreamRec = (
    ObjType: 4;
    VmtLink: Ofs(KindOf(TScroller)^);
    Load:   @TScroller.Load;
    Store:   @TScroller.Store
  );
const
  RListViewer: TStreamRec = (
    ObjType: 5;
    VmtLink: Ofs(KindOf(TListViewer)^);
    Load:   @TListViewer.Load;
    Store:   @TListViewer.Store
  );
const
  RGroup: TStreamRec = (
    ObjType: 6;
    VmtLink: Ofs(KindOf(TGroup)^);
    Load:   @TGroup.Load;
    Store:   @TGroup.Store
  );
const
  RWindow: TStreamRec = (
    ObjType: 7;
    VmtLink: Ofs(KindOf(TWindow)^);
    Load:   @TWindow.Load;
    Store:   @TWindow.Store
  );
{ Caracteres utilizados para dibujar elementos seleccionados y }
{ por defecto en conjuntos en color monocromo. }
SpecialChars: array[0..5] of Char = (#175, #174, #26, #27, ' ', ' ');
{ True si el conjunto de comandos ha cambiado desde que se puso a false }
CommandSetChanged: Boolean = False;
```

ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION

INTRODUCCION



ANEXO

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

CONTENIDOS

V.1	Introducción
V.2	Unit <i>BWCC</i>
V.3	Unit <i>ODialogs</i>
V.4	Unit <i>OMemory</i>
V.5	Unit <i>OPrinter</i>
V.6	Unit <i>OStdDlgs</i>
V.7	Unit <i>OStdWnds</i>
V.8	Unit <i>OWindows</i>

V1.1 INTRODUCCION

En el anexo anterior se presentó el interfaz de la units de *Turbo Vision*. De manera similar en este anexo se presenta el interfaz de las units de *Object Windows*. Las Units *Objects* y *Validate*, como ya se ha dicho, son comunes a las de **Turbo Vision** y ya se listaron en el anexo anterior por

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

lo que no se repiten en éste. Algunas de estas units utilizan a su vez otras units que no son propiamente de *Object Windows*, sino que son de interfaz con el API de Windows como *WinTypes*, *WinProcs* y *WinDos*. El interfaz de estas units no se lista en este anexo (contienen un elevado nº de tipos de datos, funciones y procedimientos para comunicarse con Windows).

En algunos de los listados se hace uso de las directivas *\$R* y *\$I*, que sirven para incluir ficheros de recursos en una aplicación. Los ficheros de recursos se encuentran en los directorios del compilador donde están almacenadas esas units.

V1.1 UNIT *BWCC*

```
uses WinTypes, OWindows, ODialogs;
const
  BwccVersion      = $0103;
{ Desde la versión 1.02 en adelante BWCCGetversion devuelve un
  Longint. La palabra(tipo Word) de menor orden contiene el nº
  de versión y la de mayor orden el código local (de país).  }
const
  Bwcc_Locale_US   = 1;
  Bwcc_Locale_JAPAN = 2;
{ Clase Diálogo propia de Borland }
  BorDlgClass      = 'BorDlg';
{ Propiedad usada por las ventanas de diálogo de Borland.
  ; No se debería usar una propiedad definida por el usuario
  con este nombre !. }
  BorDlgProp       = 'FB';
  IdHelp           = 998;          { Id del botón de ayuda }
{ Definiciones de los estilos de los botones:
  los botones de Borland usan los estilos de los botones de Windows
  para el tipo de botón: e.j. bs_PushButton/bs_DefPushButton }
  Button_Class     = 'BorBtn';     { Botones Bitmap de Borland }
  Radio_Class      = 'BorRadio';   { Botones de radio de Borland }
  Check_Class      = 'BorCheck';   { Botones Checkboxes de Borland }
{ Estilos }
  bbs_Bitmap:Longint = $8000;      { bitmap estático }
  bbs_DlgPaint:Longint = $4000;    { usado en tiempo de ejecución por
                                     la clase diálogo(Dialog) }
  bbs_ParentNotify:Longint=$2000;  { Notificación al padre de teclas TAB
                                     o foco. }
  bbs_OwnerDraw:Longint = $1000;   { Permitir al padre pintar por medio
                                     de wm_DrawItem }
{ Mensajes }
  bbm_SetBits      = ( BM_SETSTYLE + 10);
{ Notificaciones }
  bbn_SetFocus     = ( bn_DoubleClicked + 10);
  bbn_SetFocusmouse = ( bn_DoubleClicked + 11);
  bbn_GotaTab      = ( bn_DoubleClicked + 12);
  bbn_GotaBTab     = ( bn_DoubleClicked + 13);
  Shade_Class      = 'BorShade';
{ Nombre del mensaje de ventana pasado a RegisterWindowMessage
  durante el proceso de CtlColor para las sombras de caja del
  grupo }
  BWCC_CtlColor_Shade = 'BWCC_CtlColor_Shade';
```

UNIT ODialogs

```

bss_Group      = 1; { caja de grupo }
bss_Hdip       = 2; { borde horizontal }
bss_Vdip       = 3; { borde vertical }
bss_Hbump      = 4; { velocidad horizontal de bump }
bss_Vbump      = 5; { velocidad vertical de bump }
bss_RGroup     = 6; { caja de grupo en relieve }
bss_Caption    = $8000; { Cabecera(Caption) de grupo superior }
bss_CtlColor   = $4000; { Enviar mensaje wm_CtlColor al padre }
bss_NoPrefix   = $2000; { & en cabecera no subraya la letra siguiente }
bss_Left       = $0000; { Cabecera justificada a la izquierda }
bss_Center     = $0100; { Cabecera centrada }
bss_Right      = $0200; { Cabecera justificada a la derecha }
bss_AlignMask  = $0300;
Static_Class = 'BorStatic'; { Estáticos de Borland }
{ Declaraciones de funciones }
function DialogBox(Instance: THandle; Templatename: PChar;
  WndParent: HWnd; DialogFunc: TFarProc): Integer;
function DialogBoxParam(Instance: THandle; TemplateName: PChar;
  WndParent: HWnd; DialogFunc: TFarProc; InitParam: LongInt): Integer;
function CreateDialog(Instance: THandle; TemplateName: PChar;
  WndParent: HWnd; DialogFunc: TFarProc): HWnd;
function CreateDialogParam(Instance: THandle; TemplateName: PChar;
  WndParent: HWnd; DialogFunc: TFarProc; InitParam: LongInt): HWnd;
function BWCCMessageBox(WndParent: HWnd; Txt, Caption: PChar;
  TextType: Word): Integer;
function BWCCDefDlgProc(Dlg: HWnd; Msg, wParam: Word; lParam:
  LongInt): LongInt;
function BWCCGetPattern: HBrush;
function BWCCGetVersion: Longint;
function SpecialLoadDialog(hResMod: THandle; Templatename: PChar;
  DialogFunc: TFarProc): THandle;
function MangleDialog(hDlg: THandle; hResMod: THandle;
  DialogFunc: TFarProc): THandle;
function BWCCDefWindowProc(hWindow: HWnd; Message, wParam: Word;
  lParam: LongInt): LongInt;
function BWCCDefMDIChildProc(hWindow: HWnd; Message, wParam: Word;
  lParam: LongInt): LongInt;
{ Objetos ObjectWindows específicos para BWCC }
type
  PBDivider = ^TBDivider;
  TBDivider = object(TControl)
    constructor Init(AParent: PWindowsObject; AnId: Integer; AText: PChar;
      X, Y, W, H: Integer; IsVertical, IsBump: Boolean);
    constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
    function GetClassName: PChar; virtual;
  end;
  PBStaticBmp = ^TBStaticBmp;
  TBStaticBmp = object(TControl)
    constructor Init(AParent: PWindowsObject; AnId: Integer; AText: PChar;
      X, Y, W, H: Integer);
    constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
    function GetClassName: PChar; virtual;
  end;

```

V1.1 UNIT ODialogs

```

uses WinProcs, WinTypes, Objects, OWindows, Validate;
const
{ Estados de comprobación de TCheckBox }
  bf_Unchecked = 0;
  bf_Checked   = 1;
  bf_Grayed    = 2;
{ N° de mensaje utilizado para la validación de las entradas }
  wm_PostInvalid = wm_User + 400;

```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

type
{ Atributos de creación de TDialog }
  TDialogAttr = record
    Name: PChar;
    Param: LongInt;
  end;
{ Tipo Objeto TDialog }
  PDialog = ^TDialog;
  TDialog = object(TWindowsObject)
    Attr: TDialogAttr;
    IsModal: Boolean;
    constructor Init(AParent: PWindowsObject; AName: PChar);
    constructor Load(var S: TStream);
    destructor Done; virtual;
    procedure Store(var S: TStream);
    function Create: Boolean; virtual;
    function Execute: Integer; virtual;
    procedure EndDlg(ARetValue: Integer); virtual;
    function GetItemHandle(DlgItemID: Integer): HWnd;
    function SendDlgItemMsg(DlgItemID: Integer; AMsg, WParam: Word;
      LParam: LongInt): LongInt;
    procedure Ok(var Msg: TMessage); virtual id_First + id_Ok;
    procedure Cancel(var Msg: TMessage); virtual id_First + id_Cancel;
    procedure WMInitDialog(var Msg: TMessage);
      virtual wm_First + wm_InitDialog;
    procedure WMQueryEndSession(var Msg: TMessage);
      virtual wm_First + wm_QueryEndSession;
    procedure WMClose(var Msg: TMessage);
      virtual wm_First + wm_Close;
    procedure WMPostInvalid(var Msg: TMessage);
      virtual wm_First + wm_PostInvalid;
    procedure DefWndProc(var Msg: TMessage); virtual;
  end;
{ Tipo Objeto TDlgWindow }
  PDlgWindow = ^TDlgWindow;
  TDlgWindow = object(TDialog)
    constructor Init(AParent: PWindowsObject; AName: PChar);
    procedure GetWindowClass(var AWndClass: TWndClass); virtual;
    function Create: Boolean; virtual;
  end;
{ Tipo Objeto TControl }
  PControl = ^TControl;
  TControl = object(TWindow)
    constructor Init(AParent: PWindowsObject; AnId: Integer;
      ATitle: PChar; X, Y, W, H: Integer);
    constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
    function Register: Boolean; virtual;
    function GetClassName: PChar; virtual;
    procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
  end;
{ Tipo Objeto TGroupBox }
  PGroupBox = ^TGroupBox;
  TGroupBox = object(TControl)
    NotifyParent: Boolean;
    constructor Init(AParent: PWindowsObject; AnID: Integer;
      AText: PChar; X, Y, W, H: Integer);
    constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    function GetClassName: PChar; virtual;
    procedure SelectionChanged(ControlId: Integer); virtual;
  end;

```

UNIT ODialogs

```

{ Tipo Objeto TButton }
PButton = ^TButton;
TButton = object(TControl)
  constructor Init(AParent: PWindowsObject; AnId: Integer;
    AText: PChar; X, Y, W, H: Integer; IsDefault: Boolean);
  constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
  function GetClassName: PChar; virtual;
end;
{ Tipo Objeto TCheckBox }
PCheckBox = ^TCheckBox;
TCheckBox = object(TButton)
  Group: PGroupBox;
  constructor Init(AParent: PWindowsObject; AnID: Integer;
    ATitle: PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
  constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
  constructor Load(var S: TStream);
  procedure Store(var S: TStream);
  procedure Check;
  procedure Uncheck;
  procedure Toggle;
  function GetClassName: PChar; virtual;
  function GetCheck: Word;
  procedure SetCheck(CheckFlag: Word);
  function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
  procedure BNClicked(var Msg: TMessage);
  virtual nf_First + bn_Clicked;
end;
{ Tipo Objeto TRadioButton }
PRadioButton = ^TRadioButton;
TRadioButton = object(TCheckBox)
  constructor Init(AParent: PWindowsObject; AnID: Integer;
    ATitle: PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
  function GetClassName: PChar; virtual;
end;
{ Tipo Objeto TStatic }
PStatic = ^TStatic;
TStatic = object(TControl)
  TextLen: Word;
  constructor Init(AParent: PWindowsObject; AnId: Integer;
    ATitle: PChar; X, Y, W, H: Integer; ATextLen: Word);
  constructor InitResource(AParent: PWindowsObject; ResourceID: Word;
    ATextLen: Word);
  constructor Load(var S: TStream);
  procedure Store(var S: TStream);
  function GetClassName: PChar; virtual;
  function GetText(ATextString: PChar; MaxChars: Integer): Integer;
  function GetTextLen: Integer;
  procedure SetText(ATextString: PChar);
  procedure Clear;
  function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
end;
{ Tipo Objeto TEdit }
PEdit = ^TEdit;
TEdit = object(TStatic)
  Validator: PValidator;
  constructor Init(AParent: PWindowsObject; AnId: Integer; ATitle: PChar;
    X, Y, W, H: Integer; ATextLen: Word; Multiline: Boolean);
  constructor InitResource(AParent: PWindowsObject; ResourceID: Word;
    ATextLen: Word);
  constructor Load(var S: TStream);
  destructor Done; virtual;
  function GetClassName: PChar; virtual;
  procedure Undo;
  function CanClose: Boolean; virtual;
  function CanUndo: Boolean;
  procedure Paste;
  procedure Copy;

```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

procedure Cut;
function GetNumLines: Integer;
function GetLineLength(LineNumber: Integer): Integer;
function GetLine(ATextString: PChar;
  StrSize, LineNumber: Integer): Boolean;
procedure GetSubText(ATextString: PChar; StartPos, EndPos: Integer);
function DeleteSubText(StartPos, EndPos: Integer): Boolean;
function DeleteLine(LineNumber: Integer): Boolean;
procedure GetSelection(var StartPos, EndPos: Integer);
function DeleteSelection: Boolean;
function IsModified: Boolean;
procedure ClearModify;
function GetLineFromPos(CharPos: Integer): Integer;
function GetLineIndex(LineNumber: Integer): Integer;
function IsValid(ReportError: Boolean): Boolean;
procedure Scroll(HorizontalUnit, VerticalUnit: Integer);
function SetSelection(StartPos, EndPos: Integer): Boolean;
procedure Insert(ATextString: PChar);
function Search(StartPos: Integer; AText: PChar; CaseSensitive: Boolean):
Integer;
procedure SetupWindow; virtual;
procedure SetValidator(AValid: PValidator);
procedure Store(var S: TStream);
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
procedure CMEditCut(var Msg: TMessage);
  virtual cm_First + cm_EditCut;
procedure CMEditCopy(var Msg: TMessage);
  virtual cm_First + cm_EditCopy;
procedure CMEditPaste(var Msg: TMessage);
  virtual cm_First + cm_EditPaste;
procedure CMEditDelete(var Msg: TMessage);
  virtual cm_First + cm_EditDelete;
procedure CMEditClear(var Msg: TMessage);
  virtual cm_First + cm_EditClear;
procedure CMEditUndo(var Msg: TMessage);
  virtual cm_First + cm_EditUndo;
procedure WMChar(var Msg: TMessage);
  virtual wm_First + wm_Char;
procedure WMKeyDown(var Msg: TMessage);
  virtual wm_First + wm_KeyDown;
procedure WMGetDlgCode(var Msg: TMessage);
  virtual wm_First + wm_GetDlgCode;
procedure WMKillFocus(var Msg: TMessage);
  virtual wm_First + wm_KillFocus;
end;
{ Tipo para los nombres de los mensajes de TListBox }
TMsgName = (
  mn_AddString, mn_InsertString, mn_DeleteString,
  mn_ResetContent, mn_GetCount, mn_GetText,
  mn_GetTextLen, mn_SelectString, mn_SetCurSel,
  mn_GetCurSel);
{ Registro de transferencia de selección múltiple }
PMultiSelRec = ^TMultiSelRec;
TMultiSelRec = record
  Count: Integer;
  Selections: array[0..32760] of Integer;
end;
{ Tipo Objeto TListBox }
PListBox = ^TListBox;
TListBox = object(TControl)
  constructor Init(AParent: PWindowsObject; AnId: Integer;
    X, Y, W, H: Integer);
  function GetClassName: PChar; virtual;
  function AddString(AStr: PChar): Integer;
  function InsertString(AStr: PChar; Index: Integer): Integer;
  function DeleteString(Index: Integer): Integer;
  procedure ClearList;

```

UNIT ODialogs

```

function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
function GetCount: Integer;
function GetString(AString: PChar; Index: Integer): Integer;
function GetStringLen(Index: Integer): Integer;
function GetSelString(AString: PChar; MaxChars: Integer): Integer;
function SetSelString(AString: PChar; Index: Integer): Integer;
function GetSelIndex: Integer;
function SetSelIndex(Index: Integer): Integer;
private
function GetMsgID(AMsg: TMsgName): Word; virtual;
end;
{ Tipo Objeto TComboBox }
PComboBox = ^TComboBox;
TComboBox = object(TListBox)
  TextLen: Word;
  constructor Init(AParent: PWindowsObject; AnID: Integer;
    X, Y, W, H: Integer; AStyle: Word; ATextLen: Word);
  constructor InitResource(AParent: PWindowsObject; ResourceID: Integer;
    ATextLen: Word);
  constructor Load(var S: TStream);
  procedure Store(var S: TStream);
  function GetClassName: PChar; virtual;
  procedure ShowList;
  procedure HideList;
  function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
  procedure SetupWindow; virtual;
  function GetTextLen: Integer;
  function GetText(Str: PChar; MaxChars: Integer): Integer;
  procedure SetText(Str: PChar);
  function SetEditSel(StartPos, EndPos: Integer): Integer;
  function GetEditSel(var StartPos, EndPos: Integer): Boolean;
  procedure Clear;
private
function GetMsgID(AMsg: TMsgName): Word; virtual;
end;
{ Registro de transferencia de TScrollBar }
TScrollBarTransferRec = record
  LowValue: Integer;
  HighValue: Integer;
  Position: Integer;
end;
{ Tipo Objeto TScrollBar }
PScrollBar = ^TScrollBar;
TScrollBar = object(TControl)
  LineMagnitude, PageMagnitude: Integer;
  constructor Init(AParent: PWindowsObject; AnID: Integer;
    X, Y, W, H: Integer; IsHScrollBar: Boolean);
  constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
  constructor Load(var S: TStream);
  procedure Store(var S: TStream);
  function GetClassName: PChar; virtual;
  procedure SetupWindow; virtual;
  procedure GetRange(var LoVal, HiVal: Integer);
  function GetPosition: Integer;
  procedure SetRange(LoVal, HiVal: Integer);
  procedure SetPosition(ThumbPos: Integer);
  function DeltaPos(Delta: Integer): Integer;
  function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
  procedure SBLineUp(var Msg: TMessage);
  virtual nf_First + sb_LineUp;
  procedure SBLineDown(var Msg: TMessage);
  virtual nf_First + sb_LineDown;
  procedure SBPageUp(var Msg: TMessage);
  virtual nf_First + sb_PageUp;
  procedure SBPageDown(var Msg: TMessage);
  virtual nf_First + sb_PageDown;
  procedure SBThumbPosition(var Msg: TMessage);

```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

    virtual nf_First + sb_ThumbPosition;
    procedure SBThumbTrack(var Msg: TMessage);
    virtual nf_First + sb_ThumbTrack;
    procedure SBTop(var Msg: TMessage);
    virtual nf_First + sb_Top;
    procedure SBBottom(var Msg: TMessage);
    virtual nf_First + sb_Bottom;
end;
{ Rutinas de apoyo para realizar la selección múltiple }
function AllocMultiSel(Size: Integer): PMultiSelRec;
procedure FreeMultiSel(P: PMultiSelRec);
{ Rutina para Stream }
procedure RegisterODialogs;
const
  RDialog: TStreamRec = (
    ObjType: 54;
    VmtLink: Ofs(KindOf(TDialog)^);
    Load: @TDialog.Load;
    Store: @TDialog.Store);
const
  RDlgWindow: TStreamRec = (
    ObjType: 55;
    VmtLink: Ofs(KindOf(TDlgWindow)^);
    Load: @TDlgWindow.Load;
    Store: @TDlgWindow.Store);
const
  RControl: TStreamRec = (
    ObjType: 56;
    VmtLink: Ofs(KindOf(TControl)^);
    Load: @TControl.Load;
    Store: @TControl.Store);
const
  RMDIClient: TStreamRec = (
    ObjType: 58;
    VmtLink: Ofs(KindOf(TMDIClient)^);
    Load: @TMDIClient.Load;
    Store: @TMDIClient.Store);
const
  RButton: TStreamRec = (
    ObjType: 59;
    VmtLink: Ofs(KindOf(TButton)^);
    Load: @TButton.Load;
    Store: @TButton.Store);
const
  RCheckBox: TStreamRec = (
    ObjType: 60;
    VmtLink: Ofs(KindOf(TCheckBox)^);
    Load: @TCheckBox.Load;
    Store: @TCheckBox.Store);
const
  RRadioButton: TStreamRec = (
    ObjType: 61;
    VmtLink: Ofs(KindOf(TRadioButton)^);
    Load: @TRadioButton.Load;
    Store: @TRadioButton.Store);
const
  RGroupBox: TStreamRec = (
    ObjType: 62;
    VmtLink: Ofs(KindOf(TGroupBox)^);
    Load: @TGroupBox.Load;
    Store: @TGroupBox.Store);
const
  RListBox: TStreamRec = (
    ObjType: 63;
    VmtLink: Ofs(KindOf(TListBox)^);
    Load: @TListBox.Load;
    Store: @TListBox.Store);

```


UNIT OMemory

```
const
  RComboBox: TStreamRec = (
    ObjType: 64;
    VmtLink: ofs(KindOf(TComboBox)^);
    Load:   @TComboBox.Load;
    Store:   @TComboBox.Store);
const
  RScrollBar: TStreamRec = (
    ObjType: 65;
    VmtLink: ofs(KindOf(TScrollBar)^);
    Load:   @TScrollBar.Load;
    Store:   @TScrollBar.Store);
const
  RStatic: TStreamRec = (
    ObjType: 66;
    VmtLink: ofs(KindOf(TStatic)^);
    Load:   @TStatic.Load;
    Store:   @TStatic.Store);
const
  REdit: TStreamRec = (
    ObjType: 67;
    VmtLink: ofs(KindOf(TEdit)^);
    Load:   @TEdit.Load;
    Store:   @TEdit.Store);
```

V1.1 UNIT OMemory

```
const
  SafetyPoolSize: Word = 8192;
procedure InitMemory;
procedure DoneMemory;
function LowMemory: Boolean;
procedure RestoreMemory;
function MemAlloc(Size: Word): Pointer;
function MemAllocSeg(Size: Word): Pointer;
```

V1.1 UNIT OPrinter

```
uses WinTypes, WinProcs, Objects, OWindows, ODialogs;
{ Estados de TPrinter }
const
  ps_Ok = 0;
  ps_InvalidDevice = -1;      { Parámetros de dispositivo (al inicializar)
                               invalidos }
  ps_Unassociated = -2;      { Objeto no asociado con ninguna impresora }
{ Flags de TPrintOut }
const
  pf_Graphics = $01;         { Banda actual sólo acepta texto }
  pf_Text = $02;             { Banda actual sólo acepta gráficos }
  pf_Both = $03;            { Banda actual acepta tanto texto como
                               gráficos }
  pf_Banding = $04;         { printout convirtiéndose en bandas }
  pf_Selection = $08;       { Imprimiendo la selección }
type
  PPrintDialogRec = ^TPrintDialogRec;
  TPrintDialogRec = record
    drStart: Integer;        { Comienzo de página }
    drStop: Integer;        { Fin de página }
    drCopies: Integer;      { N° de copias a imprimir }
```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

    drCollate: Boolean;           { Indica a la impresora cotejar copias }
    drUseSelection: Boolean;      { Uso de selección en lugar de Start, Stop }
end;
{ TPrintOut representa el documento impreso físico, el cual se
envía a una impresora. TPrintOut realiza el render del documento
sobre la impresora. Para cada documento, o tipo de documento,
se debe crear la correspondiente clase TPrintOut }
type
PPrintOut = ^TPrintOut;
TPrintOut = object(TObject)
    Title: PChar;
    Banding: Boolean;
    ForceAllBands: Boolean;
    DC: HDC;
    Size: TPoint;
    constructor Init(ATitle: PChar);
    destructor Done; virtual;
    procedure BeginDocument(StartPage, EndPage: Integer;
        Flag: Word); virtual;
    procedure BeginPrinting; virtual;
    procedure EndDocument; virtual;
    procedure EndPrinting; virtual;
    function GetDialogInfo(var Pages: Integer): Boolean; virtual;
    function GetSelection(var Start, Stop: Integer): Boolean; virtual;
    function HasNextPage(Page: Word): Boolean; virtual;
    procedure PrintPage(Page: Word; var Rect: TRect; Flags: Word); virtual;
    procedure SetPrintParams(ADC: HDC; ASize: TPoint); virtual;
end;
{ TPrinter representa el dispositivo físico de impresora. Para imprimir
un TPrintOut, se envía éste al método Print de TPrinter }
TPrinter = ^TPrinter;
TPrinter = object(TObject)
    Device, Driver, Port: PChar; { Descripción del dispositivo de Impresora }
    Status: Integer;             { Estado del dispositivo, error si <> ps_Ok }
    Error: Integer;              { < 0 si el error sucede durante la impresión }
    DeviceModule: THandle;       { Handle para módulo de driver de impresora }
    DeviceMode: TDeviceMode;     { Puntero de función a DevMode }
    ExtDeviceMode: TExtDeviceMode; { Puntero de función a ExtDevMode }
    DevSettings: PDevMode;       { Copia local de configuración de impresora }
    DevSettingSize: Integer;     { Tamaño de la configuración de impresora }
    constructor Init;
    destructor Done; virtual;
    procedure ClearDevice;
    procedure Configure(Window: PWindowsObject);
    function GetDC: HDC; virtual;
    function InitAbortDialog(Parent: PWindowsObject;
        Title: PChar): PDialog; virtual;
    function InitPrintDialog(Parent: PWindowsObject; PrnDC: HDC;
        Pages: Integer; SelAllowed: Boolean;
        var Data: TPrintDialogRec): PDialog; virtual;
    function InitSetupDialog(Parent: PWindowsObject): PDialog; virtual;
    procedure ReportError(PrintOut: PPrintOut); virtual;
    procedure SetDevice(ADevice, ADriver, APort: PChar);
    procedure Setup(Parent: PWindowsObject);
    function Print(ParentWin: PWindowsObject; PrintOut: PPrintOut): Boolean;
end;
{ TPrinterSetupDlg es un diálogo para modificar que impresora está
asignada a un objeto TPrinter. Presenta todas las impresoras
activas en el sistema permitiendo al usuario seleccionar la
impresora deseada. El diálogo también permite llamar al diálogo
de configuración de impresoras para una posterior configuración
de la impresora una vez seleccionada. }
const
    id_Combo = 100;
    id_Setup = 101;

```

UNIT OPrinter

```
type
  PPrinterSetupDlg = ^TPrinterSetupDlg;
  TPrinterSetupDlg = object(TDialog)
    Printer: PPrinter;
    constructor Init(AParent: PWindowsObject; TemplateName: PChar;
      APrinter: PPrinter);
    destructor Done; virtual;
    procedure TransferData(TransferFlag: Word); virtual;
    procedure IDSetup(var Msg: TMessage);
      virtual id_First + id_Setup;
    procedure Cancel(var Msg: TMessage);
      virtual id_First + id_Cancel;
  private
    OldDevice, OldDriver, OldPort: PChar;
    DeviceCollection: PCollection;
  end;
const
  id_Title = 101;
  id_Device = 102;
  id_Port = 103;
type
  PPrinterAbortDlg = ^TPrinterAbortDlg;
  TPrinterAbortDlg = object(TDialog)
    constructor Init(AParent: PWindowsObject; Template, Title,
      Device, Port: PChar);
    procedure SetupWindow; virtual;
    procedure WMCommand(var Msg: TMessage);
      virtual wm_First + wm_Command;
  end;
const
  id_PrinterName = 102;
  id_All = 103;
  id_Selection = 104;
  id_Pages = 105;
  id_FromText = 106;
  id_From = 107;
  id_ToText = 108;
  id_To = 109;
  id_PrintQuality = 110;
  id_Copies = 111;
  id_Collate = 112;
type
  PPrintDialog = ^TPrintDialog;
  TPrintDialog = object(TDialog)
    Printer: PPrinter;
    PData: PPrintDialogRec;
    PrinterName: PStatic;
    Pages: Integer;
    Controls: PCollection;
    AllBtn, SelectBtn, PageBtn: PRadioButton;
    FromPage, ToPage: PEdit;
    Copies: PEdit;
    Collate: PCheckBox;
    PrnDC: HDC;
    SelAllowed: Boolean;
    constructor Init(AParent: PWindowsObject; Template: PChar; APrnDC: HDC;
      APages: Integer; APrinter: PPrinter; ASelAllowed: Boolean;
      var Data: TPrintDialogRec);
    procedure SetupWindow; virtual;
    procedure TransferData(Direction: Word); virtual;
    procedure IDSetup(var Msg: TMessage);
      virtual id_First + id_Setup;
  end;
```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

type
  PEditPrintout = ^TEditPrintout;
  TEditPrintout = object(TPrintout)
    Editor: PEdit;
    NumLines: Integer;
    LinesPerPage: Integer;
    LineHeight: Integer;
    StartPos: Integer;
    StopPos: Integer;
    StartLine: Integer;
    StopLine: Integer;
    constructor Init(AEditor: PEdit; ATitle: PChar);
    procedure BeginDocument(StartPage, EndPage: Integer;
      Flags: Word); virtual;
    function GetDialogInfo(var Pages: Integer): Boolean; virtual;
    function GetSelection(var Start, Stop: Integer): Boolean; virtual;
    function HasNextPage(Page: Word): Boolean; virtual;
    procedure PrintPage(Page: Word; var Rect: TRect; Flags: Word); virtual;
    procedure SetPrintParams(ADC: HDC; ASize: TPoint); virtual;
  end;
type
  PWindowPrintout = ^TWindowPrintout;
  TWindowPrintout = object(TPrintOut)
    Window: PWindow;
    Scale: Boolean;
    constructor Init(ATitle: PChar; AWindow: PWindow);
    function GetDialogInfo(var Pages: Integer): Boolean; virtual;
    procedure PrintPage(Page: Word; var Rect: TRect; Flags: Word); virtual;
  end;

```

V1.1 UNIT *OStdDlgs*

```

uses WinTypes, WinProcs, WinDos, OWindows, ODialogs, Strings;
{$R OSTDDLGS}
{ Incluir constantes para ficheros de recursos }
{$I OSTDDLGS.INC}
const
  fsFileSpec = fsFileName + fsExtension;
type
  PFileDialog = ^TFileDialog;
  TFileDialog = object(TDialog)
    Caption: PChar;
    FilePath: PChar;
    PathName: array[0..fsPathName] of Char;
    Extension: array[0..fsExtension] of Char;
    FileSpec: array[0..fsFileSpec] of Char;
    constructor Init(AParent: PWindowsObject; AName, AFilePath: PChar);
    function CanClose: Boolean; virtual;
    procedure SetupWindow; virtual;
    procedure HandleFName(var Msg: TMessage); virtual id_First + id_FName;
    procedure HandleFList(var Msg: TMessage); virtual id_First + id_FList;
    procedure HandleDList(var Msg: TMessage); virtual id_First + id_DList;
  private
    procedure SelectFileName;
    procedure UpdateFileName;
    function UpdateListBoxes: Boolean;
  end;
const
  sd_WNInputDialog = $7F02;      { Plantilla para diálogo de entrada normal }
  sd_BCInputDialog = $7F05;      { Plantilla para diálogo de entrada BWCC }
const
  id_Prompt = 100;
  id_Input = 101;

```

UNIT *OStdWnds*

```
type
  PInputDialog = ^TInputDialog;
  TInputDialog = object(TDialog)
    Caption: PChar;
    Prompt: PChar;
    Buffer: PChar;
    BufferSize: Word;
    constructor Init(AParent: PWindowsObject;
      ACaption, APrompt, ABuffer: PChar; ABufferSize: Word);
    function CanClose: Boolean; virtual;
    procedure SetupWindow; virtual;
  end;
```

V1.1 UNIT *OStdWnds*

uses WinTypes, WinProcs, WinDos, Objects, OWindows, ODialogs,
OMemory, OStdDlgs, Strings;

```
type
  { TSearchRec }
  TSearchRec = record
    SearchText: array[0..80] of Char;
    CaseSensitive: Bool;
    ReplaceText: array[0..80] of Char;
    ReplaceAll: Bool;
    PromptOnReplace: Bool;
    IsReplace: Boolean;
  end;
  { TEditWindow }
  PEditWindow = ^TEditWindow;
  TEditWindow = object(TWindow)
    Editor: PEdit;
    SearchRec: TSearchRec;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    procedure WMSize(var Msg: TMessage);
      virtual wm_First + wm_Size;
    procedure WMSetFocus(var Msg: TMessage);
      virtual wm_First + wm_SetFocus;
    procedure CMEditFind(var Msg: TMessage);
      virtual cm_First + cm_EditFind;
    procedure CMEditFindNext(var Msg: TMessage);
      virtual cm_First + cm_EditFindNext;
    procedure CMEditReplace(var Msg: TMessage);
      virtual cm_First + cm_EditReplace;
  private
    procedure DoSearch;
  end;
  { TFileWindow }
  PFileWindow = ^TFileWindow;
  TFileWindow = object(TEditWindow)
    FileName: PChar;
    IsNewFile: Boolean;
    constructor Init(AParent: PWindowsObject; ATitle, AFileName: PChar);
    destructor Done; virtual;
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    function CanClear: Boolean; virtual;
    function CanClose: Boolean; virtual;
    procedure NewFile;
    procedure Open;
    procedure Read;
    procedure SetFileName(AFileName: PChar);
    procedure ReplaceWith(AFileName: PChar);
```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```
function Save: Boolean;
function SaveAs: Boolean;
procedure SetupWindow; virtual;
procedure Write;
procedure CMFileNew(var Msg: TMessage);
  virtual cm_First + cm_FileNew;
procedure CMFileOpen(var Msg: TMessage);
  virtual cm_First + cm_FileOpen;
procedure CMFileSave(var Msg: TMessage);
  virtual cm_First + cm_FileSave;
procedure CMFileSaveAs(var Msg: TMessage);
  virtual cm_First + cm_FileSaveAs;
end;
const
  REditWindow: TStreamRec = (
    ObjType: 80;
    VmtLink: ofs(TypeOf(TEditWindow)^);
    Load: @TEditWindow.Load;
    Store: @TEditWindow.Store);
const
  RFileWindow: TStreamRec = (
    ObjType: 81;
    VmtLink: ofs(TypeOf(TFileWindow)^);
    Load: @TFileWindow.Load;
    Store: @TFileWindow.Store);
procedure RegisterStdWnds;
```

V1.1 UNIT *OWindows*

```
uses WinTypes, WinProcs, Objects;
{ Incluir constantes para ficheros de recursos }
{$I OWINDOWS.INC}
const
{ Máscaras de flags de TWindowsObject }
wb_KBHandler = $01;
wb_FromResource = $02;
wb_AutoCreate = $04;
wb_MDICChild = $08;
wb_Transfer = $10;
{ Códigos de estado de TWindowsObject }
em_InvalidWindow = -1;
em_OutOfMemory = -2;
em_InvalidClient = -3;
em_InvalidChild = -4;
em_InvalidMainWindow = -5;
{ Códigos de transferencia de TWindowsObject }
tf_SizeData = 0;
tf_GetData = 1;
tf_SetData = 2;
type
{ Registro de mensajes de ventana de TMessage }
PMessage = ^TMessage;
TMessage = record
  Receiver: HWnd;
  Message: Word;
  case Integer of
    0: (
      WParam: Word;
      LParam: Longint;
      Result: Longint);
    1: (
      WParamLo: Byte;
      WParamHi: Byte;
      LParamLo: Word;
```

UNIT OWindows

```

        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
end;
{ Usado por TWindowsObject }
PMDIClient = ^TMDIClient;
PScroller = ^TScroller;
{ Tipo Objeto TWindowsObject }
PWindowsObject = ^TWindowsObject;
TWindowsObject = object(TObject)
    Status: Integer;
    HWindow: HWND;
    Parent, ChildList: PWindowsObject;
    TransferBuffer: Pointer;
    Instance: TFarProc;
    Flags: Byte;
    constructor Init(AParent: PWindowsObject);
    constructor Load(var S: TStream);
    destructor Done; virtual;
    procedure Store(var S: TStream);
    procedure DefWndProc(var Msg: TMessage); virtual {indice 8};
    procedure DefCommandProc(var Msg: TMessage); virtual {indice 12};
    procedure DefChildProc(var Msg: TMessage); virtual {indice 16};
    procedure DefNotificationProc(var Msg: TMessage); virtual {indice 20};
    procedure SetFlags(Mask: Byte; OnOff: Boolean);
    function IsFlagSet(Mask: Byte): Boolean;
    function FirstThat(Test: Pointer): PWindowsObject;
    procedure ForEach(Action: Pointer);
    function Next: PWindowsObject;
    function Previous: PWindowsObject;
    procedure Focus;
    function Enable: Boolean;
    function Disable: Boolean;
    procedure EnableKBHandler;
    procedure EnableAutoCreate;
    procedure DisableAutoCreate;
    procedure EnableTransfer;
    procedure DisableTransfer;
    function Register: Boolean; virtual;
    function Create: Boolean; virtual;
    procedure Destroy; virtual;
    function GetID: Integer; virtual;
    function ChildWithId(Id: Integer): PWindowsObject;
    function GetClassName: PChar; virtual;
    function GetClient: PMDIClient; virtual;
    procedure GetChildPtr(var S: TStream; var P);
    procedure PutChildPtr(var S: TStream; P: PWindowsObject);
    procedure GetSiblingPtr(var S: TStream; var P);
    procedure PutSiblingPtr(var S: TStream; P: PWindowsObject);
    procedure GetWindowClass(var AWndClass: TWndClass); virtual;
    procedure SetupWindow; virtual;
    procedure Show(ShowCmd: Integer);
    function CanClose: Boolean; virtual;
    function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
    procedure TransferData(Direction: Word); virtual;
    procedure DispatchScroll(var Msg: TMessage); virtual;
    procedure CloseWindow;
    procedure GetChildren(var S: TStream);
    procedure PutChildren(var S: TStream);
    procedure AddChild(AChild: PWindowsObject);
    procedure RemoveChild(AChild: PWindowsObject);
    function IndexOf(P: PWindowsObject): Integer;
    function At(I: Integer): PWindowsObject;
    function CreateChildren: Boolean;
    function CreateMemoryDC: HDC;
    procedure WMVScroll(var Msg: TMessage); virtual wm_First + wm_VScroll;
    procedure WMHScroll(var Msg: TMessage); virtual wm_First + wm_HScroll;

```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

procedure WMCommand(var Msg: TMessage); virtual wm_First + wm_Command;
procedure WMClose(var Msg: TMessage); virtual wm_First + wm_Close;
procedure WMDestroy(var Msg: TMessage); virtual wm_First + wm_Destroy;
procedure WMNCDestroy(var Msg: TMessage); virtual wm_First + wm_NCDestroy;
procedure WMActivate(var Msg: TMessage); virtual wm_First + wm_Activate;
procedure WMQueryEndSession(var Msg: TMessage);
    virtual wm_First + wm_QueryEndSession;
procedure CMExit(var Msg: TMessage); virtual cm_First + cm_Exit;
private
    CreateOrder: Word;
    SiblingList: PWindowsObject;
end;
{ Atributos de creación de TWindow }
TWindowAttr = record
    Title: PChar;
    Style: LongInt;
    ExStyle: LongInt;
    X, Y, W, H: Integer;
    Param: Pointer;
    case Integer of
        0: (Menu: HMenu);           { Handle de Menu }
        1: (Id: Integer);          { Identificador de hijo(Child) }
    end;
end;
{ Tipo Objeto TWindow }
PWindow = ^TWindow;
TWindow = object(TWindowsObject)
    Attr: TWindowAttr;
    DefaultProc: TFarProc;
    Scroller: PScroller;
    FocusChildHandle: THandle;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
    constructor Load(var S: TStream);
    destructor Done; virtual;
    procedure Store(var S: TStream);
    procedure SetCaption(ATitle: PChar);
    procedure GetWindowClass(var AWndClass: TWndClass); virtual;
    procedure FocusChild;
    procedure UpdateFocusChild;
    function GetId: Integer; virtual;
    function Create: Boolean; virtual;
    procedure DefWndProc(var Msg: TMessage); virtual;
    procedure WMActivate(var Msg: TMessage);
        virtual wm_First + wm_Activate;
    procedure WMMDIActivate(var Msg: TMessage);
        virtual wm_First + wm_MDIActivate;
    procedure SetupWindow; virtual;
    procedure WMCreate(var Msg: TMessage);
        virtual wm_First + wm_Create;
    procedure WMHScroll(var Msg: TMessage);
        virtual wm_First + wm_HScroll;
    procedure WMVScroll(var Msg: TMessage);
        virtual wm_First + wm_VScroll;
    procedure WMPaint(var Msg: TMessage);
        virtual wm_First + wm_Paint;
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure WMSize(var Msg: TMessage);
        virtual wm_First + wm_Size;
    procedure WMMove(var Msg: TMessage);
        virtual wm_First + wm_Move;
    procedure WMLButtonDown(var Msg: TMessage);
        virtual wm_First + wm_LButtonDown;
    procedure WMSysCommand(var Msg: TMessage);
        virtual wm_First + wm_SysCommand;
private
    procedure UpdateWindowRect;
end;

```


UNIT OWindows

```

{ Tipo Objeto TMDIWindow }
PMDIWindow = ^TMDIWindow;
TMDIWindow = object(TWindow)
  ClientWnd: PMDIClient;
  ChildMenuPos: Integer;
  constructor Init(ATitle: PChar; AMenu: HMenu);
  destructor Done; virtual;
  constructor Load(var S: TStream);
  procedure Store(var S: TStream);
  procedure SetupWindow; virtual;
  procedure InitClientWindow; virtual;
  function GetClassName: PChar; virtual;
  function GetClient: PMDIClient; virtual;
  procedure GetWindowClass(var AWndClass: TWndClass); virtual;
  procedure DefWndProc(var Msg: TMessage); virtual;
  function InitChild: PWindowsObject; virtual;
  function CreateChild: PWindowsObject; virtual;
  procedure CMCreateChild(var Msg: TMessage);
    virtual cm_First + cm_CreateChild;
  procedure TileChildren; virtual;
  procedure CascadeChildren; virtual;
  procedure ArrangeIcons; virtual;
  procedure CloseChildren; virtual;
  procedure CMTileChildren(var Msg: TMessage);
    virtual cm_First + cm_TileChildren;
  procedure CMCascadeChildren(var Msg: TMessage);
    virtual cm_First + cm_CascadeChildren;
  procedure CMArrangeIcons(var Msg: TMessage);
    virtual cm_First + cm_ArrangeIcons;
  procedure CMCloseChildren(var Msg: TMessage);
    virtual cm_First + cm_CloseChildren;
end;
{ Tipo Objeto TMDIClient }
TMDIClient = object(TWindow)
  ClientAttr: TClientCreateStruct;
  constructor Init(AParent: PMDIWindow);
  constructor Load(var S: TStream);
  procedure Store(var S: TStream);
  function GetClassName: PChar; virtual;
  function Register: Boolean; virtual;
  procedure TileChildren; virtual;
  procedure CascadeChildren; virtual;
  procedure ArrangeIcons; virtual;
  procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
end;
{ Tipo Objeto TScroller }
TScroller = object(TObject)
  Window: PWindow;
  XPos: LongInt; { posición horizontal actual en unidades de scroll
                  horizontal }
  YPos: LongInt; { posición vertical actual en unidades de scroll
                  vertical }
  XUnit: Integer; { unidades de dispositivo lógicas por unidad de
                  scroll horizontal }
  YUnit: Integer; { unidades de dispositivo lógicas por unidad de
                  scroll vertical }
  XRange: LongInt; { n° de unidades de scroll horizontal con
                    posibilidad de scroll }
  YRange: LongInt; { n° de unidades de scroll vertical con
                    posibilidad de scroll }
  XLine: Integer; { n° de unidades de scroll horizontal por línea }
  YLine: Integer; { n° de unidades de scroll vertical por línea }
  XPage: Integer; { n° de unidades de scroll horizontal por página }
  YPage: Integer; { n° de unidades de scroll vertical por página }
  AutoMode: Boolean; { Modo auto-scroll }
  TrackMode: Boolean; { Modo track scroll }
  AutoOrg: Boolean; { Origen de los desplazamientos de Scroller }

```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

```

HasHScrollBar: Boolean;
HasVScrollBar: Boolean;
constructor Init(TheWindow: PWindow; TheXUnit, TheYUnit: Integer;
  TheXRange, TheYRange: LongInt);
constructor Load(var S: TStream);
destructor Done; virtual;
procedure Store(var S: TStream);
procedure SetUnits(TheXUnit, TheYUnit: LongInt);
procedure SetPageSize; virtual;
procedure SetSBarRange; virtual;
procedure SetRange(TheXRange, TheYRange: LongInt);
procedure BeginView(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
procedure EndView; virtual;
procedure VScroll(ScrollRequest: Word; ThumbPos: Integer); virtual;
procedure HScroll(ScrollRequest: Word; ThumbPos: Integer); virtual;
procedure ScrollTo(X, Y: LongInt);
procedure ScrollBy(Dx, Dy: LongInt);
procedure AutoScroll; virtual;
function IsVisibleRect(X, Y: LongInt; XExt, YExt: Integer): Boolean;
private
  function XScrollValue(ARangeUnit: Longint): Integer;
  function YScrollValue(ARangeUnit: Longint): Integer;
  function XRangeValue(AScrollUnit: Integer): Longint;
  function YRangeValue(AScrollUnit: Integer): Longint;
end;
{ Tipo Objeto TApplication }
PApplication = ^TApplication;
TApplication = object(TObject)
  Status: Integer;
  Name: PChar;
  MainWindow: PWindowsObject;
  HAccTable: THandle;
  KBHandlerWnd: PWindowsObject;
  constructor Init(AName: PChar);
  destructor Done; virtual;
  function IdleAction: Boolean; virtual;
  procedure InitApplication; virtual;
  procedure InitInstance; virtual;
  procedure InitMainWindow; virtual;
  procedure Run; virtual;
  procedure SetKBHandler(AWindowsObject: PWindowsObject);
  procedure MessageLoop; virtual;
  function ProcessAppMsg(var Message: TMsg): Boolean; virtual;
  function ProcessDlgMsg(var Message: TMsg): Boolean; virtual;
  function ProcessAccels(var Message: TMsg): Boolean; virtual;
  function ProcessMDIAccels(var Message: TMsg): Boolean; virtual;
  function MakeWindow(AWindowsObject: PWindowsObject): PWindowsObject; vir-
tual;
  function ExecDialog(ADialog: PWindowsObject): Integer; virtual;
  function ValidWindow(AWindowsObject: PWindowsObject): PWindowsObject; vir-
tual;
  procedure Error(ErrorCode: Integer); virtual;
  function CanClose: Boolean; virtual;
end;
{ Funciones de utilidad }
function GetObjectPtr(HWindow: HWnd): PWindowsObject;
{ Rutinas para Stream }
procedure RegisterOWindows;
procedure RegisterWObjects;
{ Rutinas inline para Longint }
function LongMul(X, Y: Integer): Longint;
inline($5A/$58/$F7/$EA);
function LongDiv(X: Longint; Y: Integer): Integer;
inline($59/$58/$5A/$F7/$F9);
{ Puntero a objetos aplicación }
const
  Application: PApplication = nil;

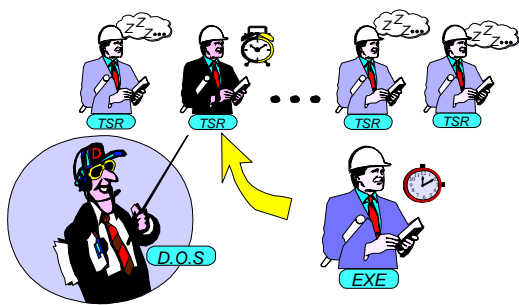
```

UNIT OWindows

```
{ Estructuras Record para registrar objetos para su uso con Stream }
const
  RWindowsObject: TStreamRec = (
    ObjType: 52;
    VmtLink: Ofs(KindOf(TWindowsObject)^);
    Load:   @TWindowsObject.Load;
    Store:   @TWindowsObject.Store);
const
  RWindow: TStreamRec = (
    ObjType: 53;
    VmtLink: Ofs(KindOf(TWindow)^);
    Load:   @TWindow.Load;
    Store:   @TWindow.Store);
const
  RMDIWindow: TStreamRec = (
    ObjType: 57;
    VmtLink: Ofs(KindOf(TMDIWindow)^);
    Load:   @TMDIWindow.Load;
    Store:   @TMDIWindow.Store);
const
  RScroller: TStreamRec = (
    ObjType: 68;
    VmtLink: Ofs(KindOf(TScroller)^);
    Load:   @TScroller.Load;
    Store:   @TScroller.Store);
type
  TCreateDialogParam = function (HInstance: THandle; TemplateName: PChar;
    WndParent: HWnd; DialogFunc: TFarProc; InitParam: LongInt): HWnd;
  TDialogBoxParam = function (HInstance: THandle; TemplateName: PChar;
    WndParent: HWnd; DialogFunc: TFarProc; InitParam: LongInt): Integer;
  TDefaultProc = function (Wnd: HWnd; Msg, wParam: Word;
    lParam: LongInt): LongInt;
  TMessageBox = function (WndParent: HWnd; Txt, Caption: PChar;
    TextType: Word): Integer;
const
  CreateDialogParam: TCreateDialogParam = WinProcs.CreateDialogParam;
  DialogBoxParam: TDialogBoxParam = WinProcs.DialogBoxParam;
  DefWndDlgProc: TDefaultProc = WinProcs.DefWindowProc;
  DefMDIDlgProc: TDefaultProc = WinProcs.DefMDIChildProc;
  DefDlgProc: TDefaultProc = WinProcs.DefDlgProc;
  MessageBox: TMessageBox = WinProcs.MessageBox;
  BWCCClassNames: Boolean = False;
```

ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS

INTRODUCCION



ANEXO VI

ANEXO VI: PROGRAMAS RESIDENTES

CONTENIDOS

- VI.1 Introducción
- VI.2 Vector de interrupciones
- VI.3 Recursos de Turbo Pascal para crear programas residentes
- VI.4 Un ejemplo
- VI.5 Ampliaciones y notas bibliográficas

VII.1 INTRODUCCION

Los primeros sistemas operativos para microordenadores eran poco más que un *shell* BASIC, Pascal o PL/I que ofertaba el entorno del lenguaje y un simple manejador de ficheros. Estos sistemas estaban orientados normalmente a ordenadores específicos, sin compatibilidad entre diferentes sistemas. A medida que el hardware evolucionó, los sistemas operativos (S.O.) comenzaron a estabilizarse. Entre estos se encontraban *Apple DOS*, *North Star DOS*, *TRSDOS* y *CP/M*. Estos sistemas compartían las siguientes características:

- Ejecución de un sólo programa por un sólo usuario al mismo tiempo.
- Soporte de un sistemas de ficheros estructurado en directorios.
- Soporte de interpretes de lenguajes, ensambladores y compiladores.
- Incompatibilidad de programas y ficheros de datos entre sistemas.

ANEXO VI: PROGRAMAS RESIDENTES

Cada S.O. tenía sus propias características, no compartidas por otros. *Gary Kildall* diseñó CP/M como un S.O. genérico. Aunque inicialmente fue ideado como un entorno de desarrollo de software para el procesador *Intel MDS*, CP/M no estaba ligado a un modelo de computador específico. CP/M consta de un sistema operativo de disco básico (BDOS), de un procesador de comandos de consola (CCP) y de un sistema básico de E/S (BIOS). La BIOS es el código específico del hardware (el código que maneja la consola, la impresora y el disco del sistema). La idea consistía en que escribiendo una BIOS a medida, un ordenador aceptaría miles de programas disponibles. Dado que es adaptable, CP/M llegó a ser el S.O. estándar en la industria para ordenadores personales que disponían de un procesador **8080** o **Z80**.

Los procesadores 8080 y Z80 pueden direccionar sólo 64 Kb de memoria, por lo que CP/M fue diseñado como un sistema monousuario y monoprogamación. Una versión multiusuario se desarrolló a partir del CP/M y se denominó MP/M, aunque nunca llegó a ser tan extendido como lo fue CP/M, principalmente porque era lento⁹⁷ y por que los ordenadores personales no eran muy adecuados para tareas multiusuario.

El DOS es una adaptación desarrollada para correr en el procesador 8086, con una apariencia distinta al CP/M. Microsoft lo compró a su creador, *Seattle Computers Products*, e IBM obtuvo la licencia de Microsoft. DOS recuerda todavía a CP/M en muchos aspectos, aunque con cada nueva versión se distancia cada vez más. DOS consta de tres módulos similares en funcionalidad al BDOS, CCP y BIOS de CP/M, y su interface de usuario en línea de comandos es idéntico al del CP/M. DOS tiene características adicionales que no se encuentran en CP/M, como tuberías (*pipes*), filtros, redirección de entrada/salida, marca de fecha/tiempo de fichero y una estructura de directorios de ficheros jerárquica. DOS, no obstante, sigue siendo un sistema operativo monousuario y monoprogamación como el CP/M.

Los primeros PC's de IBM eran muy similares a sus antecesores basados en 8080 y Z80: tenían 64 Kb de memoria, lectoras de discos flexibles y un procesador no mucho más rápido que un Z80. La monoprogamación del DOS era adecuada para este tipo de máquinas, aunque el PC tenía tres características en su arquitectura que le predestinaban al crecimiento que le llevó a lo que es hoy en día: el microprocesador 8088 puede direccionar hasta 1 Mb de memoria, tiene una estructura de *vector de interrupciones* y el teclado de un PC y el monitor son partes integradas en el ordenador en lugar de utilizarse una terminal conectada al ordenador por una puerta serie. Estas características permiten, en cierta medida, solventar la carencia de la multitarea al permitir la utilización de programas conocidos como **programas residentes** o **programas Terminar-y-Quedar-Residente** (*Terminated-and-Stay-Resident* o *TSR*), que son aquellos que una vez finalizada su ejecución permanecen en la memoria del ordenador sin que esta sea ocupada por otros programas ejecutados posteriormente.

⁹⁷ El 8080 no es un procesador rápido.

VECTOR DE INTERRUPCIONES

Los programas residentes, con memoria suficiente, una estructura con soporte para interrupciones y un monitor de vídeo con mapeo sobre memoria, mejoran notablemente las prestaciones del DOS frente a otros entornos de monousuario y monotarea.

El DOS incluye dos funciones que permiten a los programas declararse a sí mismos como residentes en memoria. La función 0x31 de la interrupción⁹⁸ 0x21 de DOS finaliza el programa que está ejecutando en ese momento, pero permitiéndole que quede residente en memoria. DOS no intentará usar la memoria declarada por el programa residente como propia. La interrupción de DOS 0x27 realiza la misma función pero restringe el tamaño del programa residente a 64 Kb.

Las dos funciones TSR que aporta DOS no soportan programas de utilidad residentes de memoria, ya que una vez que finalizaron su ejecución no volverán a tener el control de ordenador. Simplemente se está desperdiciando el espacio ocupado por los programas que quedaron residentes. Para que estos programas tengan utilidad hay que utilizar un mecanismo que los active una vez que hayan quedado residentes en la memoria. Este mecanismo es el que DOS concibió para el desarrollo de rutinas de servicio de interrupción (*Interrupt Service Routines ISR's*) para manejar hardware de entrada/salida específico como ratones, tabletas digitalizadoras o *joysticks*. Estos dispositivos no son partes de la arquitectura del PC y por lo tanto no tienen un software de interface estándar en DOS.

El entorno de ISR puede ser utilizado para soportar otro tipo de programas que no tienen por que estar necesariamente asociados con un dispositivo, pero que si pueden ampliar el interface del ordenador. Estos programas son los programas residentes. Típicos programas residentes son aquellos que permiten incrementar el rendimiento del teclado (*keyboard enhancers*) mediante definiciones de macros asociadas con ciertas combinaciones de teclas o aquellos otros que permiten añadir accesorios al entorno (calculadoras, editores activables en cualquier instante y desde cualquier programa). Ejemplos de *keyboard enhancers* son *Prokey* y *SuperKey* que permiten al usuario asignar secuencias de caracteres a una tecla de función (Fn), una combinación con (Alt) o cualquier otra tecla. Un ejemplo muy conocido de accesorios de entorno es el programa *Sidekick* y *Homebase* que aportan blocks de notas, calculadoras, marcadores de teléfono y otras ayudas que se activan con cierta combinación de teclas.

VII.1 VECTOR DE INTERRUPCIONES

Una interrupción es una suspensión momentánea de un procedimiento secuencial dentro de un programa que permite ejecutarse otro programa. Cuando se completa la interrupción, el programa interrumpido se reanuda. El proceso interrumpido puede que no sea afectado en modo alguno por el que interrumpe, o puede que sí (caso del ejemplo de este anexo). La interrupción puede ser causada por un evento externo al programa en ejecución o puede ser generada por el propio programa, es decir, por un evento de hardware o por una instrucción software del programa.

⁹⁸ El concepto de interrupción se vió en el capítulo 10.

ANEXO VI: PROGRAMAS RESIDENTES

Hay 256 interrupciones en la arquitectura de un PC, numeradas de 0 a 0xff. Algunas de estas interrupciones son definidas por el procesador⁹⁹. Otras son definidas por la propia arquitectura del PC para llamar a funciones de la ROM-BIOS¹⁰⁰. Otras son definidas por el DOS para realizar las tareas propias del sistema operativo. Es decir, cada una de las tres capas (microprocesador, arquitectura del PC y sistemas operativo) tiene su propio conjunto de interrupciones reservado. El resto de las interrupciones están disponibles para los programas de usuario y las rutinas de servicio de interrupción de dispositivos.

Cada interrupción está representada en un vector compuesto de elementos de 4 bytes almacenado en la memoria del ordenador en posiciones absolutas: desde la 0 a la 0x3ff. Cuando ocurre una interrupción, el registro de flags del procesador y las direcciones de 4 bytes presentes son almacenadas en el *stack*, se inhabilitan posteriores interrupciones y se pasa el control a la dirección indicada en la posición del vector de interrupciones asociada con la interrupción producida. El programa que toma el control deberá guardar los registros de la CPU que vaya a utilizar para devolverlos a su estado cuando finalice de forma que el programa interrumpido pueda reanudar su ejecución en el estado que estaba cuando fue interrumpido.

Este mecanismo será el que utilicen los programas residentes para ser activados posteriormente una vez finalizada su ejecución. Para ello habrá que modificar el vector de interrupciones almacenando la dirección de entrada al TSR en alguna posición del vector asociada con el evento que se desee que active al programa residente.

El construir un programa residente no es tan sencillo como ésto. Se deberá *hilar fino* para no alterar a otros programas residentes, ni al programa en ejecución ni al propio sistema operativo. Para desarrollar TSR que funcionen correctamente hay que tener un profundo conocimiento de la forma de trabajar el sistema operativo y los distintos programas que bajo su tutela se ejecutan en el ordenador.

La figura VI.1 muestra el mapa de memoria del DOS en dos estados distintos: sin ningún programa residente y con dos TSR cargados en memoria. Esta estructura deberá ser tenida en cuenta para implementar un programa residente, así como la forma que tiene el DOS de ejecutar un programa: cada programa que se ejecuta (sólo puede ejecutarse un programa bajo DOS) tiene su propio contexto (ficheros a manejar, su segmento de datos y código, etc) y el DOS sólo permite, como es obvio, tener un único contexto activo. Una tarea, que puede ser necesaria pero no obligatoria, en un TSR es recuperar su propio contexto almacenado en su **prefijo de segmento de**

⁹⁹ Por ejemplo, la interrupción 0 es la interrupción de división por cero.

¹⁰⁰ La ROM-BIOS contiene funciones para soporte la arquitectura propia del PC: acceso a memoria, video, teclado, disco,

RECURSOS DE TURBO PASCAL PARA CREAR PROGRAMAS RESIDENTES

programa (*Program Segment Prefix, PSP*). Esta operación requiere un profundo conocimiento de la estructura de un programa y su ubicación en memoria. La figura VI.2 muestra el mapa de memoria de un programa generado con el compilador *Turbo C* con el modelo de memoria *Tiny*¹⁰¹.

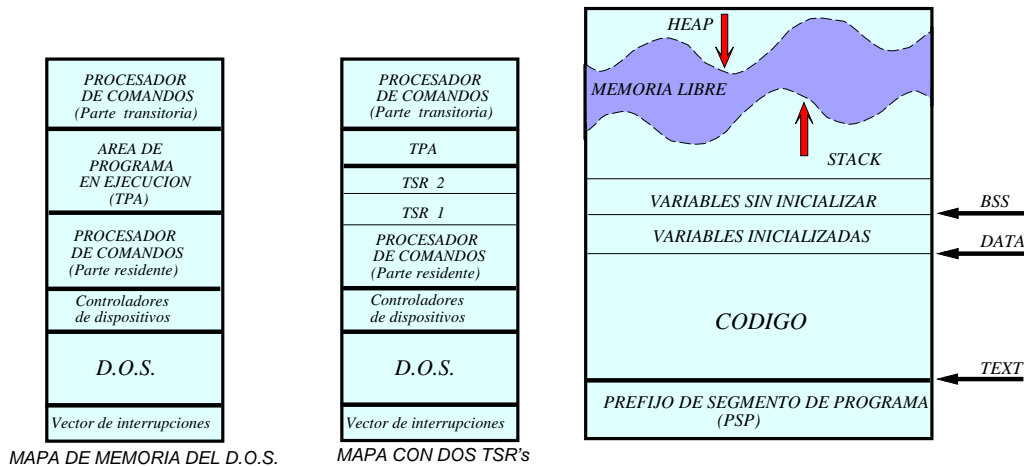


Fig. VI.1 Mapa de memoria del D.O.S.

Fig. VI.2. Mapa de memoria de un programa

No es el propósito de este anexo el entrar en detalles sobre estas peculiaridades, o alguna otra como: la forma de detectar que un programa ya está en memoria para que no vuelva a quedarse residente con la correspondiente duplicación de código, o cómo descargar y desactivar un TSR. Nuestro objetivo es presentar las facilidades que ofrece Turbo Pascal para la construcción de programas residentes. Esto será lo que hagamos en el siguiente apartado.

Para concluir este apartado citar otro concepto a tener en cuenta en los programas residentes: el **encadenamiento de interrupciones**. Se pueden crear o utilizar TSR's que se enlacen en la misma posición del vector de interrupciones. Cuando ésto sucede, el que tiene el control ante el evento asociado a esa posición del vector es el último programa instalado en memoria. Este decidirá si llama al TSR que tenía el control antes de su *toma de posesión en el vector* o si se limita exclusivamente a realizar su tarea.

VII.1 RECURSOS DE TURBO PASCAL PARA CREAR PROGRAMAS RESIDENTES

Los recursos que Turbo Pascal pone en manos del programador para crear TSR's son muy limitados. Se ciñen únicamente a tres:

- Reservar la memoria necesaria para el correcto funcionamiento del programa
- Manipulación del vector de interrupciones.

¹⁰¹ El modelo de memoria *Tiny* genera programas cuya ocupación, entre datos y código, no supera las 64 Kb's de memoria.

ANEXO VI: PROGRAMAS RESIDENTES

- Permitir la finalización de un programa quedando residente en memoria.

El programador deberá encargarse de otras tareas como de que el programa tenga un mecanismo de activación, descargar el programa residente, reconocer la posibilidad de que el programa ya este en memoria, recuperar su propio contexto por ejemplo para manipular ficheros inicializados durante la ejecución que le permite permanecer posteriormente en memoria, etc.

Un programa que desee residir en memoria deberá indicar al S.O. cuánta memoria precisará. Esto se puede realizar mediante la directiva **\$M**. Con esta directiva se indica cuanto espacio de *stack* y de *heap* se precisa para el programa.

Sintaxis:

```
{ $M tamaño de stack, heap mínima, heap máxima }
```

Para DOS *Heap Mínima* y *Heap Máxima* especifican los tamaños de mínimo y máximo, respectivamente, de memoria *heap*. *Heap Mínima* debe estar en el rango desde 0 a 655360 y *Heap Máxima* en el rango desde *Heap Mínima* a 655360.

Para poder instalar una rutina en una posición del vector de interrupciones se dispone del procedimiento `SetIntVec`:

Declaración:

```
procedure SetIntVec(NumeroInterrupcion: Byte; var Vector: Pointer);
```

Ejemplo:

```
SetIntVec($9, Addr102(RESI));
```

y para poder obtener la dirección de una rutina de interrupción instalada se utiliza `GetIntVec`:

Declaración:

```
procedure GetIntVec(NumeroInterrupcion: Byte; var Vector: Pointer);
```

Ejemplo:

```
GetIntVec($9, @KbdIntVec);
```

que almacena en la dirección de una variable puntero a procedimiento la dirección de la rutina de interrupción situada en la posición del vector de interrupciones pasada como primer argumento.

En la declaración de los procedimientos que se vayan a utilizar como rutinas de interrupción se debe añadir la directiva de procedimiento **interrupt** después de la cabecera:

Ejemplo:

```
PROCEDURE RESI; interrupt;
```

102 *Addr* devuelve la dirección del objeto pasado como argumento.

UN EJEMPLO

Para finalizar un programa no es necesario generar la interrupción software del DOS que realiza esta operación. Esto se podría hacer con los procedimientos *Intr* o *MsDos*¹⁰³ de la *Unit Dos*. Para llevar a cabo esta misión se utiliza el procedimiento **Keep** que finaliza el programa y lo deja residente en memoria.

Declaración:

```
procedure Keep(CodigoExit: Word);
```

Ejemplo:

```
Keep(0);
```

El programa entero permanece en memoria incluyendo el segmento de datos, el segmento de stack y la memoria heap. `CodigoExit` corresponde al valor pasado al procedimiento estándar *Halt* llamado por *keep*.

El byte bajo de `CodigoExit` es el código enviado por el proceso que finaliza. El byte alto se codifica de acuerdo a la tabla VI.1.

Tipo de terminación	Byte Alto
<i>Normal</i>	<i>0</i>
<i>Ctrl-C</i>	<i>1</i>
<i>Error de dispositivo</i>	<i>2</i>
<i>Procedure Keep</i>	<i>3</i>

Tabla VI.1 Valores del byte alto del código de finalización de un programa.

VII.1 UN EJEMPLO

Como demostración de programa residente en este apartado se presenta un ejemplo de TSR, que implementa una utilidad para calcular el **nif** (nº de identificación fiscal) a partir de un **dni** dado. Este programa manipula las páginas de video para poder guardar la pantalla en el estado que estaba cuando se activo el programa residente. Almacena también la posición del cursor para que una vez cumplida su tarea lo deje en el lugar que estaba.

La activación del programa residente se realiza con la tecla **F4**. Por ese motivo el TSR se encadena a la posición 9 del vector de interrupción, que corresponde a la rutina que da servicio al teclado, para almacenar las pulsaciones de usuario en el buffer circular gestionado por el DOS. La comprobación de que la tecla pulsada es la que activa el programa se efectúa leyendo directamente el puerto de teclado (\$60) y comparando con el código de scan de la tecla **F4** (62). Si no

103 *Intr* ejecuta una interrupción software específica. *MsDos* ejecuta una llamada a una función DOS.

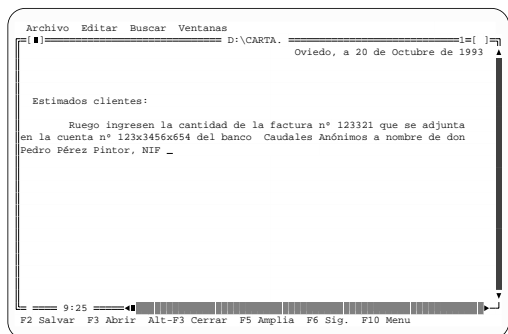
es esa tecla el programa no realiza ningún trabajo adicional. Si es la tecla de activación salva la pantalla actual mediante la utilización de las páginas de video de acuerdo al modo de vídeo activo y presenta la pantalla para calcular el nif.

El último *nif* calculado (antes de pulsar **Esc**) es introducido en el buffer de teclado del DOS (función `resultado_a_buffer`), de manera que si el programa que quedó suspendido temporalmente, estaba esperando por lectura de pulsaciones de teclado tomará los del buffer del DOS y los introducirá en la posición donde se estaba situado¹⁰⁴.

En este ejemplo se utilizan datos del tipo **regs** estudiado en el capítulo 10, así como la generación de interrupciones software, por ejemplo para obtener el modo de vídeo activo.

En este ejemplo no han implementado algunas de las características que debe tener un buen TSR, como son:

- la posibilidad de desactivar el programa para que temporalmente no responda al evento de activación que se le halla definido y su posterior activación
- detección de que este programa ya está cargado en memoria para que una posterior ejecución, una vez dejado residente, no duplique el código en memoria. Si el programa residente se ejecuta repetidas veces se puede observar con el comando **mem** del DOS como disminuye la memoria disponible. La última copia cargada en memoria será la que se active cada vez que se produzca la pulsación de **F4**



La figura VI.3 muestra la ejecución del programa residente mientras se está escribiendo una carta dentro de un editor creado con *Turbo Vision*. En la posición donde se estaba situado en el texto de la ventana activa del editor, se inserta el último *nif* calculado por el programa residente.

104 La lectura de los caracteres del buffer del DOS funcionará correctamente si el programa lee los caracteres a través de funciones de DOS. Si realiza lectura directa al dispositivo los caracteres permanecerán en el buffer hasta que algún programa los consuma o elimine el contenido de dicho buffer.

UN EJEMPLO

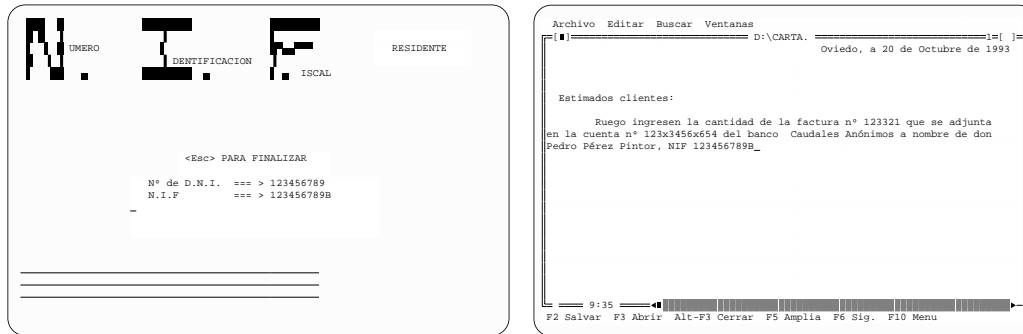


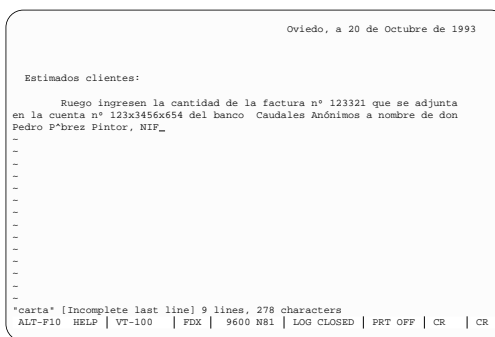
Fig. VI.3 Ejecución del programa residente desde un editor bajo DOS

Este TSR puede que no funcione correctamente con algunos programas, con lo que normalmente quedará el ordenador bloqueado teniéndose que realizar un *reboot* (reiniciar la máquina). Si se carga un TSR posterior al del ejemplo, y este también utiliza la interrupción de teclado para activarse muy probablemente dejará inutilizado nuestro TSR (y viceversa). En este caso, en principio, no tiene por que producirse un bloqueo de la máquina.

El programa se ha probado con algunos editores de texto, procesadores y programas de comunicaciones (*Turbo Pascal 7.0 (turbo)* , *Lotus Manuscript* , ...) funcionando correctamente. Por ejemplo, si se activa desde el compilador **tpx** que se distribuye dentro del compilador *Turbo Pascal 7.0* el ordenador se bloquea.

En la figura VI.4 se ejecuta el TSR desde un programa de comunicaciones que permite conectar un PC como terminal de un miniordenador con sistema operativo UNIX.

Una vez en sesión dentro del sistema UNIX, se ejecuta el editor estándar de este S.O. (*vi*). Si una vez en edición se lanza el programa residente, el nif calculado se inserta en el fichero que se está creando. Hay que resaltar que el programa *vi* se está ejecutando en otra máquina de arquitectura y características muy diferentes a las del PC donde se encuentra instalado el TSR. El fichero donde se guarda el nif calculado, es un fichero UNIX.



ANEXO VI: PROGRAMAS RESIDENTES

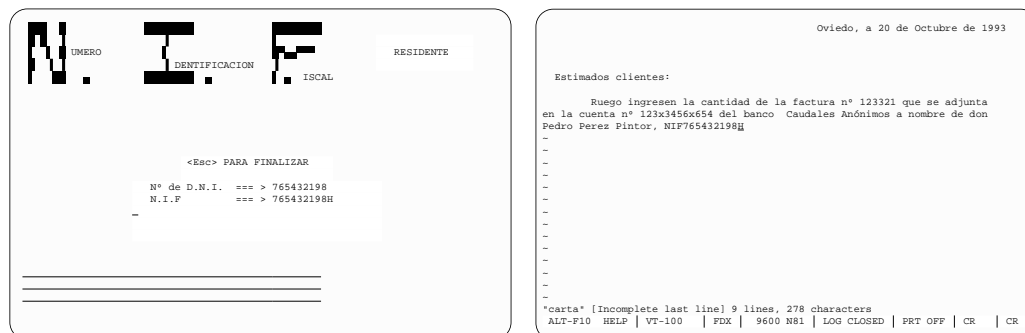


Fig. VI.4 Ejecución del programa residente desde el editor vi de UNIX por medio de un programa de comunicaciones ejecutado en el PC.

Programa residente NIFR.PAS

```
{
  Programa residente que guarda pantalla, calcula un nif y devuelve ultimo
  nif calculado en el buffer de teclado, volviendo a restaurar la pantalla
  original.
}
```

```
{ $M $1600,0,0 } { 4K stack, sin memoria heap }
```

USES

```
Crt, Dos;
```

VAR

```
KbdIntVec : Procedure;
```

```
{ $F+ }
```

```
{-----}
```

PROCEDURE RESI; interrupt;

VAR

```
c:char;
```

PROCEDURE NIF;

```
{ Programa que queda residente para calcular el NIF . Se activa con F4 }
```

CONST

```
long = 8;
```

```
{ tabla letras Nif }
```

```
letra : ARRAY[0..22] of char = ( 'T','R','W','A','G','M','Y','F','P','D',
                                'X','B','N','J','Z','S','Q','V','H','L',
                                'C','K','E');
```

VAR

```
dni,temp : longint;
```

```
cadena : ARRAY[1..long+1] OF char; { NIF en forma de string }
```

```
fin,correcto : boolean;
```

```
cursor_x,cursor_y:byte;
```

```
mode_video,colnum,page_disp,n_pag:byte;
```

```
n:integer; { N° de cifras del dni }
```

```
FUNCTION leer(VAR dn:longint) : boolean;
```

VAR

```
ch:char;
```

```
cx:integer;
```

```
{ Columna donde se esta leyendo
  el dni }
```

UN EJEMPLO

```
PROCEDURE n_error;
{ Limpia la ventana de errores }
BEGIN
  Window(2, 21, 50,23);
  TextBackground(Black);
  clrscr;
  Window(20, 14, 60,18);
  TextBackground(White);
  TextColor(Blue);
  gotoxy(cx,1);
END;
PROCEDURE error(e:INTEGER);
BEGIN
  Window(2, 21, 50,23);
  TextBackground(Red);
  TextColor(128+White);
  write('ERROR. ');
  CASE e OF
    1 :   writeln('  CARACTER NO NUMERICO.  ');
    2 :   writeln('  N° EXCESIVO DE CIFRAS.  ');
    3 :   writeln('  NO HAY DIGITOS PARA Borrar. ');
  END;
  Window(20, 14, 60,18);
  TextBackground(White);
  TextColor(Blue);
  gotoxy(cx,1);
END;

BEGIN
dn:=0;
n:=0;
cx:=24;
REPEAT
  ch:=READKEY;
  IF ch IN ['0'..'9'] THEN
    BEGIN
      IF n>long THEN
        error(2)
      ELSE
        BEGIN
          n_error;
          cadena[cx-23]:=ch;
          dn:=dn*10+(ord(ch)-ord('0'));
          n:=n+1;
          cadena[n]:=ch;
          gotoxy(cx,1);
          write(ch);
          cx:=cx+1;
        END;
      END
    ELSE
      IF(ord(ch) = 27) THEN
        fin := TRUE
      ELSE
        IF (ord(ch) = 8) THEN
          BEGIN
            IF cx > 24 THEN
              BEGIN
                cx:=cx-1;
                n:=n-1;
                n_error;
                dn:=dn DIV 10;
              END
            ELSE
              error(3);
              gotoxy(cx,1);
              write(' ');
            END
          END
        END
      END
    END
```

ANEXO VI: PROGRAMAS RESIDENTES

```

                gotoxy(cx,1);
            END
        ELSE
            IF (ord(ch) <> 13) THEN
                BEGIN
                    IF ord(ch)=0 THEN
                        ch:=ReadKey;
                    error(1);
                END;
            UNTIL (ORD(CH) = 13) OR fin;
            n_error;
            IF (ord(ch) = 13) THEN
                leer:=TRUE
            ELSE
                leer:=FALSE;
            END;
        END;

PROCEDURE Get_video; { Obtiene modo vídeo }
VAR
    regs : registers;
BEGIN
    regs.AX:=$0F00;
    Intr(16,regs);
    mode_video:=Lo(regs.AX);
    colnum     :=Hi(regs.AX);
    page_disp  :=Hi(regs.BX);
END;

PROCEDURE switch_video_off; { Desactiva vídeo }
VAR
    regs : registers;
BEGIN
    CASE mode_video OF
        0 : Port[$3D8]:=$24;
        1 : Port[$3D8]:=$20;
        2 : Port[$3D8]:=$25;
        3 : Port[$3D8]:=$21;
    END;
END;

PROCEDURE switch_video_on; { Activa vídeo }
VAR
    regs : registers;
BEGIN
    CASE mode_video OF
        0 : Port[$3D8]:=$2C;
        1 : Port[$3D8]:=$28;
        2 : Port[$3D8]:=$2D;
        3 : Port[$3D8]:=$29;
    END;
END;

{ Copia la página "n1" en la página "n2" }
PROCEDURE Copia_pag(n1,n2:byte);
TYPE
    { En modo 40 caracteres hay 8 páginas de pantalla.
      En modo 80 caracteres hay 4 páginas de pantalla. }
    bufferPant40 = ARRAY[0..7,1..2048] OF byte;
    bufferPant80 = ARRAY[0..3,1..4096] OF byte;
VAR
    display40 : bufferPant40 ABSOLUTE $B800:0;
    display80 : bufferPant80 ABSOLUTE $B800:0;
BEGIN
    IF mode_video IN [0..3]
        THEN
            CASE colnum OF
                40 : BEGIN

```


UN EJEMPLO

```

        switch_video_off;
        display40[n2]:=display40[n1];
        switch_video_on;
    END;
80 : BEGIN
        switch_video_off;
        display80[n2]:=display80[n1];
        switch_video_on;
    END;
END;

{ Envía resultado a buffer de teclado }
PROCEDURE resultado_a_buffer;
VAR
    cabbufkbd : word    ABSOLUTE 0:$41A; { apunta a la cabecera del buffer }
    colbufkbd : word    ABSOLUTE 0:$41C;
    bufkbd    : array [1..16] OF RECORD
        car : char;
        ext : byte;
    END ABSOLUTE 0:$41E;
    i: byte;
BEGIN
    cabbufkbd:=30;
    FOR i:=1 TO n DO
        bufkbd[i].car:= cadena[i];
    IF n>0 THEN
        { Se ha leído algun dígito del dni }
        BEGIN
            bufkbd[n+1].car:= cadena[n+1];
            colbufkbd:=cabbufkbd+(n+1)*2;
        END
    ELSE
        colbufkbd:=30;
    END;
END;

BEGIN
    { Guardar pantalla y posición del cursor }
    cursor_x:=WhereX;
    cursor_y:=WhereY;
    get_video;
    IF page_disp = 2 THEN
        n_pag:=3
    ELSE
        n_pag:=2;
    copia_pag(page_disp,n_pag);
    { Portada del programa }
    Window(1, 1, 80,25);
    TextBackground(Black);
    TextColor(Green);
    CLRSCR;
    writeln(' *** *                *                *                * ');
    writeln(' * * *                *                *                * ');
    writeln(' * * * UMER0                *                *                * ');
    writeln(' * * *                * DENTIFICACION                * ');
    writeln(' * ** *                *                *                * ISCAL ');
    Window(60, 2, 75,4);
    TextBackground(11);
    TextColor(128+8);
    Clrscr;
    GOTOXY(4,2);
    writeln('RESIDENTE');
    Window(28, 12, 52,13);
    TextBackground(Blue);
    TextColor(White);
    Clrscr;
    { Se lee una tecla porque el Keypressed en programa residente no

```

ANEXO VI: PROGRAMAS RESIDENTES

```

funciona bien la primera vez que se lee tras utilizarlo por
vez primera en el programa}
writeln('    PULSE UN TECLA  ');
REPEAT
UNTIL Keypressed;
c:=readkey;
Clrscr;
writeln(' <Esc> PARA FINALIZAR ');
Window(20, 14, 60,18);
TextBackground(White);
TextColor(Blue);
fin:= FALSE;
{Leemos dni y calculamos su NIF, hasta pulsar ESC}
REPEAT
  clrscr;
  write('    Nº de D.N.I.  === > ');
  correcto:=leer(dni);
  IF correcto THEN
    BEGIN
      writeln;
      write('    N.I.F          === > ');
      temp := dni MOD 23;
      writeln(dni,letra[temp]);
      cadena[n+1]:=letra[temp];
      REPEAT
        UNTIL Keypressed;
        c:=readkey;
        IF (ord(c) = 27) THEN
          fin:= TRUE;
      END;
    UNTIL fin;
    Window(1, 1, 80,25);
    TextBackground(Black);
    TextColor(White);
    IF correcto THEN
      { meter resultado en buffer }
      resultado_a_buffer;
    clrscr;
    { restaurar pantalla,posicionar cursor }
    copia_pag(n_pag,page_disp);
    gotoxy(cursor_x,cursor_y);
  END; { NIF }

BEGIN
{ Llamada a la ISR antigua para control de teclado usando el vector salvado}
  KbdIntVec;
{ Comprobar si tecla pulsada es F4}
  IF Port[$60] = 62 THEN
    BEGIN
      { Restaura el vector de interrupciones para evitar que se
        controle si se vuelve a pulsar F4 }
      SetIntVec($9,@KbdIntVec);
      { Leer dos códigos correspondientes a F4 de buffer de teclado}
      c:=readkey;
      c:=readkey;
      { Llamada a NIF}
      nif;
      { Restaurar vector interr. con este programa }
      SetIntVec($9,Addr(RESI));
    END;

```

AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

```
inline ($9C); { PUSHF -- Pone flags }
END; { RESI }
    {$F-}

{-----}

BEGIN
    { Meter ISR en la vble. de procedimiento }
    GetIntVec($9,@KbdIntVec);
    { Poner la direccion del procedimiento residente en el vector de
      interrupciones}
    SetIntVec($9,Addr(RESI));
    Keep(0); { Terminar, y dejar el programa residente }
END.
```

VII.1 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

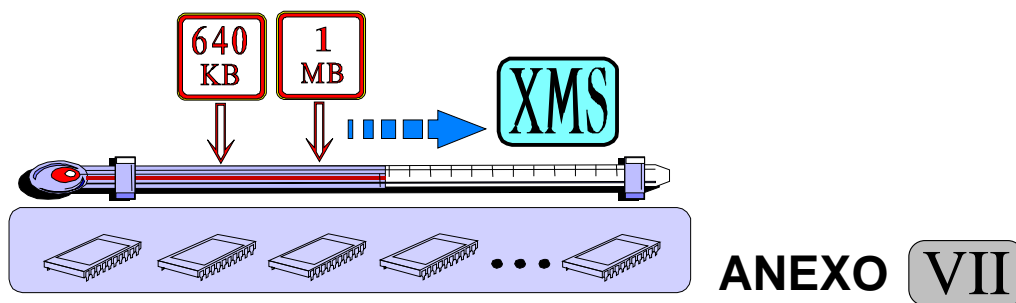
Para profundizar en el conocimiento e implementación de programas residentes se puede consultar el libro *Turbo C: Memory-Resident Utilities, screen I/O and programming techniques* (Al Stevens, MIS Press, 1987). Aunque implementa todos los ejemplos en lenguaje C dedica el capítulo 11 a exponer las características de los TSR's sin ligarlo a ningún lenguaje en concreto, para luego construir varios ejemplos en el siguiente capítulo.

Existen en el mercado bibliotecas o librerías de funciones que facilitan enormemente la construcción de programas residentes. Un ejemplo es la librería *TSRs and More* de *TechMate Inc.* y *Turbo Power Software* que ofrece un conjunto de funciones para utilizar desde Turbo C++, Borland C++, Microsoft C/C++ y Turbo Pascal en la creación de programas residentes, gestión de XMS y EMS, manipulación de datos, ...

Se pretende profundizar en los entresijos del sistema operativo DOS, los PC's o la BIOS de IBM sería recomendable consultar los libros ya recomendados en el capítulo 10 sobre éstos aspectos.

ANEXO VI: PROGRAMAS RESIDENTES

INTRODUCCION



ANEXO VII: MEMORIA EXTENDIDA

CONTENIDOS

- VII.1 Introducción
- VII.2 Unit XMS
- VII.3 Uso de la unit XMS
- VII.4 Ampliaciones y notas bibliográficas

VII.1 INTRODUCCION

Cuando se tienen varios megabytes de memoria extendida, y se necesita acceder a más memoria de la que Turbo Pascal permite con los arrays estáticos o con la memoria dinámica es preciso implementar un módulo que permita reservar memoria más allá del Mbyte que gestiona DOS (hasta 640 Kb para usuario).

ANEXO VII: MEMORIA EXTENDIDA

La *unit* que se lista a continuación (*XMS.TPU*) permite acceder a la memoria extendida. Para que funcionen correctamente las rutinas que implementa debe estar cargado el controlador de memoria extendida *HIMEN.EXE*¹⁰⁵ en el *CONFIG.SYS*. Los programas que utilicen esta Unit deben ejecutarse en un ordenador con procesador 286 o superior y con más de un megabyte de memoria.

VIII.1 UNIT XMS

```
UNIT XMS;
{----- Unit para acceso a memoria extendida -----}

INTERFACE

VAR
  XMSErrorCode: byte; {Código de error definido en especificaciones XMS}
  XMSAddr      : pointer; {Punto de entrada para el driver HIMEM.SYS}

FUNCTION XMSDriverCargado: boolean;
FUNCTION XMSMemoriaTotalDisponible: word;
FUNCTION XMSBloqueMayorDisponible: word;
FUNCTION XMSReservaBloque(KBSize: word): word;
FUNCTION XMSLiberaBloque(handle: word): boolean;
FUNCTION XMSMoverDatosA(sourceAddr: pointer; numBytes: longInt;
                        XMSHandle: word; XMSOffset: longInt): boolean;
FUNCTION XMSObtenerDatosDe(XMSHandle: word; XMSOffset: longInt;
                           numBytes: longInt; lowMemAddr: pointer): boolean;

IMPLEMENTATION
TYPE
  XMSMoveStruct = RECORD
    moveLen: longInt; { Longitud del bloque a mover en bytes }
    CASE integer OF
      {Variante 0: Memoria baja a XMS}
      0: (sHandle : word; {Para mem. convencional: Source Handler = 0}
          sPtr   : pointer; {Dirección de origen (source address)}
          XMSHdl : word; {Manejador de destino XMS }
          XMSOffset: longInt); {Desplazamiento XMS de 32 bits}
      {Variante 1: XMS a Memoria baja}
      1: (XMSH : word; {Manejador de origen XMS }
          XMSOfs : longInt; {Comienzo de desplazamiento en XMS}
          DHandle: word; {0 cuando el destino es mem. convencional}
          dPtr   : pointer); {dirección en mem. convencional}
    END;

VAR moveparms: XMSMoveStruct; {Estructura para mover desde y hacia XMS}

(*****
 * XMSDriverCargado - Devuelve TRUE si está cargado el driver de memoria
 * extendida HIMEM.SYS
 * - Establece la dirección del punto de entrada (XMSAddr)*
 *****)
FUNCTION XMSDriverCargado: boolean;
CONST
  himemSeg: word = 0;
  himemOfs: word = 0;
BEGIN
  XMSErrorCode := 0;
```

¹⁰⁵ Controlador de memoria extendida que viene con DOS o con WINDOWS.

UNIT XMS

```

ASM
  mov ax,4300h { Comprueba que HIMEM.SYS está instalado }
  int 2fh
  cmp al, 80h { Devuelve AL = 80H si está instalado }
  jne @1
  mov ax,4310h { Ahora se obtiene el punto de entrada }
  int 2fh
  mov himemOfs,bx
  mov himemSeg,es
@1:
END;
XMSDriverCargado := (himemSeg <> 0);
XMSAddr := Ptr(himemSeg, himemOfs);
END;

(*****
(* XMSMemoriaTotalDisponible - Devuelve el total de memoria XMS disponible*)
*****
)
FUNCTION XMSMemoriaTotalDisponible: word;
BEGIN
  XMSErrorCode := 0;
  XMSMemoriaTotalDisponible := 0;
  IF XMSAddr = NIL
    THEN
      IF NOT XMSDriverCargado
        THEN
          Exit;
ASM
  mov ah,8
  call XMSAddr
  or ax,ax
  jnz @1
  mov XMSErrorCode,b1 { Establece el código de error }
  xor dx,dx
  @1:
  mov @Result,dx { DX = total de memoria extendida libre }
END;
END;

(*****
(* XMSBloqueMayorDisponible - Devuelve el mayor bloque de memoria XMS *)
(* contigua disponible *)
*****
)
FUNCTION XMSBloqueMayorDisponible: word;
BEGIN
  XMSErrorCode := 0;
  XMSBloqueMayorDisponible := 0;
  IF XMSAddr = NIL { Comprueba que HIMEM.SYS está cargado }
    THEN
      IF NOT XMSDriverCargado
        THEN
          Exit;
ASM
  mov ah,8
  call XMSAddr
  or ax,ax
  jnz @1
  mov XMSErrorCode,b1 { Establece el código de error, en caso de error }
  @1:
  mov @Result,ax { AX = mayor bloque de XMS libre }
END;
END;

```

ANEXO VII: MEMORIA EXTENDIDA

```

(*****
(* XMSReservaBloque - Direcciona un bloque de memoria XMS *)
(* - Entrada: KBSize (nº de Kbytes requeridos) *)
(* - Devuelve Handle de memoria, si tiene éxito *)
(*****)
FUNCTION XMSReservaBloque(KBSize: word): word;
BEGIN
  XMSReservaBloque := 0;
  XMSErrorCode := 0;
  IF XMSAddr = NIL { Comprueba que HIMEM.SYS está cargado }
  THEN
    THEN
      IF NOT XMSDriverCargado
      THEN
        Exit;
  ASM
    mov ah,9
    mov dx,KBSize
    call XMSAddr
    or ax,ax
    jnz @1
    mov XMSErrorCode,b1 { Establece el código de error, en caso de error }
    xor dx,dx
    @1:
    mov @Result,dx { DX = Manejador(handle) de memoria extendida }
  END;
END;

(*****
(* XMSLiberaBloque - Libera un bloque de la memoria XMS *)
(* - Entrada: handle identifying memory a liberar *)
(* - Devuelve TRUE si tiene éxito *)
(*****)
FUNCTION XMSLiberaBloque(handle: word): boolean;
VAR
  Ok: word;
BEGIN
  XMSErrorCode := 0;
  XMSLiberaBloque := false;
  IF XMSAddr = NIL { Comprueba que HIMEM.SYS está cargado }
  THEN
    THEN
      IF NOT XMSDriverCargado
      THEN
        Exit;
  ASM
    mov ah,0Ah
    mov dx,handle
    call XMSAddr
    or ax,ax
    jnz @1
    mov XMSErrorCode,b1 { Establece el código de error, en caso de error }
    @1:
    mov Ok,ax
  END;
  XMSLiberaBloque := (Ok <> 0);
END;

(*****
(* XMSMoverDatosA - Mueve bloques de datos de memoria convencional a XMS *)
(* - Los datos tienen que estar direccionados previamente *)
(* - Entrada - sourceAddr: dirección de los datos en *)
(* - memoria convencional *)
(* - numBytes: número de bytes a mover *)
(* - XMSHandle: handle para bloques XMS *)
(* - XMSOffset: Desplazamiento de destino de 32 *)
(* - bits en bloque XMS *)
(* - Devuelve TRUE si se completa con éxito *)
(*****)

```


UNIT XMS

```

FUNCTION XMSMoverDatosA(sourceAddr: pointer; numBytes: longInt;
                        XMSHandle: word; XMSOffset: longInt): boolean;
VAR status: word;
BEGIN
  XMSErrorCode := 0;
  XMSMoverDatosA := FALSE;
  IF XMSAddr = NIL { Comprueba que HIMEM.SYS está cargado }
  THEN
    IF NOT XMSDriverCargado
    THEN
      Exit;
    moveParams.moveLen := numBytes;
    moveParams.sHandle := 0; { Handle de origen = 0 para }
    moveParams.sPtr := sourceAddr; { memoria convencional }
    moveParams.XMSHdl := XMSHandle;
    moveParams.XMSOffset := XMSOffset;
  ASM
    mov ah,0Bh
    mov si,offset MoveParams
    call XMSAddr
    mov status,ax { Estado de finalización de operación }
    or ax,ax
    jnz @l
    mov XMSErrorCode,b1 { Salva el código de error }
    @l:
  END;
  XMSMoverDatosA := (Status <> 0);
END;

(*****
(* XMSObtenerDatosDe - Mueve bloques de XMS a memoria convencional *)
(* - Los datos tienen que estar direccionados previamente*)
(* y movidos a XMS *)
(* - Entrada - XMSHandle: handle para bloques XMS de *)
(* origen. *)
(* - XMSOffset: Desplazamiento de destino de *)
(* 32 bits en bloque XMS *)
(* - numBytes: número de bytes a mover *)
(* - LowMemAddr: dirección de destino en *)
(* memoria convencional *)
(* - Devuelve TRUE si se completa con éxito *)
(*****
FUNCTION XMSObtenerDatosDe(XMSHandle: word; XMSOffset: longInt;
                          numBytes: longInt; lowMemAddr: pointer): boolean;
VAR status: word;
BEGIN
  XMSErrorCode := 0;
  XMSObtenerDatosDe := FALSE;
  IF XMSAddr = NIL { Comprueba que HIMEM.SYS está cargado }
  THEN
    IF NOT XMSDriverCargado
    THEN
      Exit;
    moveParams.moveLen := numBytes;
    moveParams.XMSH := XMSHandle;
    moveParams.XMSOfs := XMSOffset;
    moveParams.DHandle := 0; { Handle Destino = 0 para }
    moveParams.dPtr := lowMemAddr; { memoria convencional }
  ASM
    mov ah,0Bh
    mov si,offset MoveParams
    call XMSAddr
    mov status,ax { Estado de finalización de operación }
    or ax,ax
    jnz @l
    mov XMSErrorCode,b1 { Establece el código de error }
    @l:

```

ANEXO VII: MEMORIA EXTENDIDA

```
END;  
XMSObtenerDatosDe := (Status <> 0);  
END;  
  
BEGIN  
  XMSAddr := NIL;    { Se inicializa XMSAddr }  
  XMSErrorCode := 0;  
END.
```

```
Driver HIMEM.SYS cargado = TRUE  
Memoria extendida total: 1728 KB  
Mayor bloque de memoria extendida libre: 1728 KB  
512 KB handle = 43282  
Total de memoria extendida disponible después del direccionamiento: 1216 KB  
Llenando bloques de memoria  
131072  
Datos en XMS [1]=1  
Datos en XMS [131072]=131072  
Liberación de bloque (TRUE = Correcta, FALSE = Incorrecta) TRUE  
Pulse una tecla  
-
```

Fig. VII.1 Ejecución del programa TestXMS

VIII.1 USO DE LA UNIT XMS

El programa que se presenta a continuación sirve como banco de pruebas de la Units *XMS* y como ejemplo de su utilización. Este programa realiza lo siguiente:

- Verifica que el controlador de memoria extendida está cargado.
- Indica el tamaño del mayor bloque de memoria extendida que esté libre.
- Reserva un bloque de memoria extendida de 512 Kb para almacenar en él 131.072 enteros largos (*LongInt*) de 4 bytes cada uno.
- Indica el manejador de memoria devuelto.
- Informa de la memoria extendida disponible después de la reserva de 512 Kb efectuada.
- Inicializa el bloque reservado con los números de 1 a 131.072.

USO DE LA UNIT XMS

- Lee y presenta el valor del primer y último elemento en el área de 512 Kb de memoria extendida.
- Libera el bloque de memoria reservado utilizando el manejador devuelto cuando se efectuó la solicitud del bloque.

La figura VII.1 contiene una ejecución del programa TestXMS en la que se pueden apreciar los puntos antes citados.

Programa de prueba de la unit XMS

```
{ Programa para el test de la Unit XMS }
{$X+}

PROGRAM TestXMS;

USES crt, XMS;

CONST
  numVars = 131072;   {Nº total de variables en array      }
  bytesPerVar = 4;   {p.e. 2 para integer, 4 para longInt...}
VAR
  i: longInt;
  result: longInt;
  hdl: word;         {Handle para direccionar memoria extendida}
  hiMemOk: boolean;
BEGIN
  ClrScr;
  hiMemOk := XMSDriverCargado;
  Writeln(' Driver HIMEM.SYS cargado = ', hiMemOk);
  IF NOT hiMemOk
  THEN
    Halt;
  Writeln(' Memoria extendida total: ', XMSMemoriaTotalDisponible, ' KB');
  Writeln(' Mayor bloque de memoria extendida libre: ',
    XMSBloqueMayorDisponible, ' KB');
  {Direccionamiento de memoria - hdl es el handle de bloques de memoria}
  {
  o identificador
  }
  hdl := XMSReservaBloque((numVars * bytesPerVar + 1023) DIV 1024);
  { 1023 para redondear al siguiente KB }
  Writeln(' ',(numVars * bytesPerVar + 1023) DIV 1024, ' KB handle = ', hdl);
  Write(' Total de memoria extendida disponible después del');
  Writeln(' direccionamiento: ',XMSMemoriaTotalDisponible, ' KB');
  { Se dan los valores desde 1 hasta NumVars a las variables para
  el ejemplo }
  Writeln(' Llenando bloques de memoria ');
  FOR I := 1 TO NumVars DO
    BEGIN
      { Los parámetros en Move Data son:
      - Direcciones de Data a Move
      - Número de Bytes
      - Memoria Handle
      - Desplazamiento en el área XMS }
      IF NOT XMSMoverDatosA(@I, BytesPerVar, Hdl, (I - 1) * BytesPerVar)
      THEN
        Writeln(' Error sobre Move a XMS: ',I,' Error: ', XMSErrorCode);
      IF I MOD 1024 = 0 THEN Write(I:7,^M);
    END;
  Writeln;
  { Lectura de localizaciones }
  I := 1; { Primer elemento }
  IF NOT XMSObtenerDatosDe(Hdl, (I - 1) * BytesPerVar, BytesPerVar, @Result)
  THEN
    Writeln(' Error sobre XMSObtenerDatosDe')
```

ANEXO VII: MEMORIA EXTENDIDA

```
ELSE
  Writeln(' Datos en XMS [' ,I,']=', Result); { Escribirlo }
I := NumVars; { Ultimo elemento }
IF NOT XMSObtenerDatosDe(Hdl, (I - 1) * BytesPerVar, BytesPerVAR, @Result)
THEN
  Writeln (' Error sobre XMSObtenerDatosDe')
ELSE
  Writeln(' Datos en XMS [' ,I,']=',Result); { Escribirlo }
Writeln(' Liberación de bloque (TRUE = Correcta, FALSE = Incorrecta) ',
  XMSLiberaBloque(Hdl));
Writeln (' Pulse una tecla ');
ReadKey;
END.
```

La rapidez del movimiento de datos en memoria extendida con *HIMEN.EXE* deja mucho que desear. Cargando *EMM386.EXE*, la gestión de memoria se hace más rápida. Se pueden utilizar otros gestores de memoria como *QEMM*. Trabajando bajo este controlador el programa va incluso más rápido que con la combinación de *HIMEN.EXE* con *EMM386.EXE*

Los programas Turbo Pascal no pueden almacenar estructuras de datos en memoria XMS y acceder a ellas como a otras estructuras dinámicas. Es necesario mover los datos a la memoria convencional para poder usarlas.

Quizá el uso más práctico que se puede hacer de la Unit *XMS* sea para almacenar datos temporales de un programa, que de otra forma se deberían guardar en un fichero.

VIII.1 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Puede consultarse el libro de *Ray Duncan* titulado *MS-DOS avanzado* (Ed. Anaya, 1988), así como los libros mencionados en el anexo anterior y en el capítulo 10.

INDICE ALFABETICO

- \$DEFINE, 1033
- \$ELSE, 1033
- \$ENDIF, 1033
- \$IFDEF, 1033
- \$IFNDEF, 1033
- \$IFOPT, 1033
- \$M, 1105
- \$R
 - Incluir fichero de recursos, 959
- \$UNDEF, 1033
- \$X, 830
- \$X+
 - sintaxis extendida, 566

- {I+}, 476
- {I-}, 476

- Absolute, 571
- Activación de métodos, 659
- Addr, 563
- Algebra de Boole, 124
- Algoritmo, 2
- Ambito
 - subprogramas, 260
 - variables, 246
- Análisis, 58
- AND, 124
- Antepasado, 673
- AOO
 - Análisis Orientado a Objetos, 999
 - definición de Booch, 1001
- API, 570, 963, 965
- Append, 112
- Apuntadores, 528
- Arbol, 556
 - binario, 556
 - nodo padre, 556
 - nodo raíz, 556
 - nodos hijos, 556
 - nodos hojas, 556
- Arbol binario
 - algoritmos de tratamiento, 557
 - búsqueda, 560
 - creación, 558
 - recorrido, 559
- Arc, 275
- Argumentos en línea de comandos, 480
- Array
 - de registros, 413
- ARRAY
 - asignación de un array completo, 317
 - bidimensional, 311
 - empaquetado, 322
 - estructura de datos, 300
 - multidimensional, 311, 319
 - n-dimensional, 320
- Arrays
 - de caracteres con base 0, 566
- Ascendente, 673
- ASCII, 3, 324
- Asignación de memoria
 - dinámica, 575
 - dinámica heap, 576
 - dinámica stack, 575
 - estática, 575
- Asíncrono, 47
- Asm, 432
- Assign, 112
- AssignCrt, 113
- Assigned, 563
- Ayuda interactiva
 - ayuda sensible al contexto, 873
 - fichero de ayuda, 877
 - hint, 874
 - referencias cruzadas, 878

- Bases de datos, 45
- Baudio, 29, 47
- Biblioteca, 265, 269, 272, 273, 276, 279
- Bibliotecas de clases, 60, 63
- BIOS, 11, 435
- Bit, 2
- BlockRead, 112, 478
- BlockWrite, 112, 478
- BMP, 28
- Böhm, 67
- Booch, 61
- Booleano, 6
- Bpi, 21
- Browser, 676
- Buffer, 48, 465
- Búsqueda asociativa, 183
- Byte, 2

- C++, 652
- CAD, 49
- CADD, 49
- Cadenas
 - terminadas en caracter nulo, 564
- Cadenas de caracteres, 323
- CAE, 49
- CAI, 53
- CAL, 53
- CAM, 49
- Campo
 - de un registro, 406
- Campo selector, 415
- Campos bitmap
 - definición, 790
 - flags, 790

- máscara, 791
- máscaras de eventos, 810
- operaciones sobre, 791
- Caracteres, 3
 - alfabéticos, 3
 - de control, 3
 - especiales, 3
 - expandidos, 3
 - numéricos, 3
- Caracteres por segundo (cps), 22
- CASE
 - estructura, 160
 - registros variantes, 415
 - sentencia, 186
- CD-ROM, 21
- Chapin, 69
- ChDir, 112
- Chip, 9
- Ciclo de vida de una aplicación, 76
- Cilindros, 19
- CIM, 49
- Circle, 275
- CISC, 12
- Clascal, 652
- Clase, 61, 654, 1002
- Clases, 60
 - librería de, 730
- ClearViewPort, 275
- Clipboard, 889, 984
- Close, 112
- CloseGraph, 275
- ClrEol, 114
- ClrScr, 114
- Códigos de barras, 29
- Colas, 553
- Colores, 271
- Comentarios, 106
- Compatibilidad de tipos
 - object, 681
- Compilador, 39
- Componentes de un programa
 - barras de desplazamiento, 743
 - botón, 737
 - botones de radio, 740
 - casillas de verificación, 740
 - cuadro de diálogo, 739
 - editores de línea, 746
 - lista de selección, 744
 - menú, 738
 - objetos de control, 742
 - ventana, 737
 - vista, 737
 - vista de control, 741
 - vista principal, 738
 - vistas de datos, 737, 746
 - vistas de texto, 740, 745
 - vistas gráficas, 740
- Computación, 1
- Computadora, 2
- Comunicación
 - dirección, 249, 252
 - valor, 249, 251
- Concat, 351
- Concatenación
 - operador, 351
- Conceptos avanzados
 - persistencia, 885
- Conjunto
 - desigualdad, 395
 - diferencia, 394
 - igualdad, 395
 - inclusión, 395
 - intersección, 394
 - subconjunto, 392
 - unión, 394
 - universal, 392
 - vacio, 392
- Conjunto de caracteres
 - ASCII, 129
 - EBCDIC, 130
- CONST, 89
- CONSTRUCTOR, 683, 687
 - inicializar, 683
- Contexto de representación (DC), 965
- Control numérico, 50
- Controladores de memoria extendida
 - EMM386.EXE, 1123
 - HIMEN.EXE, 1117, 1123
 - QEMM, 1123
- Coordenadas
 - del periférico, 268
 - del usuario, 269
 - normalizadas, 269
- Coprocador matemático, 9
- Copy, 351
- COWS, 732
- CPU, 7
- Crt, 113
- Cseg, 563
- Cuadratura de Gauss, 342
- Cuadros de diálogo, 955
 - amodales, 956
 - modales, 956
- Cursor, 950
- Datos, 2
 - escalares, 146
 - ordinales, 146
- DC, Display Context, 965
- DDE, 889, 900, 971, 224
 - cliente, 992
 - servidor, 992
- Declaración
 - global, 246, 258

- local, 246, 258
- Declaraciones, 89
 - etiquetas, 249
 - parámetros formales, 249
- Definiciones, 89
- Delay, 114
- Delete, 351
- DelLine, 114
- Demodulación, 28
- Densidad de grabación, 18
- Descendiente, 673
- Descriptor de fichero, 456
- DESTRUCTOR, 697
- Destruyores, 697
- DetectGraph, 275
- Diagramas de flujo, 65
- Diagramas sintácticos, 1018
- Dijkstra, 59
- Disco duro, 19
- Diseño asistido por ordenador, 49
- Diseño de la jerarquía de clases, 62
- Diseño descendente, 59
- Diseño modular, 298
- Diseño orientado a objetos, 60
- Disk Array, 20
- Dispose
 - Ampliación del procedimiento, 698
- Disquetes, 17
- DLL, 889, 946, 987
 - funciones obligatorias, 988
 - ventajas de su uso, 990
 - y drivers de dispositivo, 988
- DLLs
 - fundamentales de Windows, 988
- Documentación de programas, 76
- Done
 - destructor, 904
- DOO
 - definición de Booch, 1001
 - Diseño Orientado a Objetos, 999
- Dynamic Link Libraries (DLL), 987

- E/S, 7
- EAO, 53
- Early binding
 - ligadura temprana, 682
- EBCDIC, 3
- EBNF, 87
- Echo, 48
- Editores, 43
- Efectos laterales, 261
 - Alias, 263
- ELSE, 168
 - ambigüedad de la cláusula ELSE, 170
- EMS, 11
- Encapsulación, 331, 654
- Enlace
 - dinámico (dynamic linking), 988
 - estático (static linking), 988
- Ensamblador, 39
- Ensamblador 80x86, 265
- Entornos operativos, 48
- Entrada / Salida
 - unidades de, 7
- Eoln, 100
- EPROM, 11
- Erase, 112
- Error
 - de heap, 702
- Error representacional, 5
- Errores
 - de compilación, 75
 - de ejecución, 75
- Errores de entrada/salida, 476
- Estructura
 - alternativa, 158
 - repetitiva, 161
 - secuencial, 158
- Estructura de datos, 300
- Estructura dinámica de datos, 319
- Estructura estática de datos, 319
- Estructuras de control de flujo, 65
 - alternativas, 66
 - repetitivas, 66
 - secuencial, 66
- Estructuras de datos
 - genéricas, 705
- Estructuras dinámicas de datos, 526
 - lineales, 534
 - no lineales, 555
- Etiquetas CASE, 186
- etiquetas de GOTO, 191
- Evento, 891
 - anulación de eventos, 821
 - bucle principal de eventos, 748
 - categorías, 747
 - definición, 731
 - enmascarar eventos, 815
 - evento abandonado, 758, 814, 825
 - evento de exposición, 750
 - fase, 815
 - programación dirigida por eventos, 747
- Eventos en Turbo Vision
 - anulación de eventos, 821
 - bucle principal de procesamiento, 812
 - clasificación, 809
 - comandos, 817
 - comunicación entre vistas, 828
 - enmascarar eventos, 815
 - evento abandonado, 814, 825
 - eventos de emisión, 814
 - eventos de mensaje, 811
 - eventos de ratón, 810
 - eventos de teclado, 811

- eventos definidos por el programador, 815
- eventos enfocados, 813
- eventos nulos, 811
- eventos posicionales, 813
- máscaras de eventos, 810
- mensajes, 809
- procesamiento concurrente, 827
- registro de evento, 810, 824
- tiempo muerto entre eventos, 827, 835
- Exportar
 - tipos object, 668
- Expresiones
 - absolutas, 571
 - reubicables, 571
- Extensibilidad, 670
- External, 432
- EXTERNAL, 265

- Fail (procedimiento), 703
- Ficheros
 - definición, 457
 - escritura, 458
 - lectura, 460
 - por dirección, 457
- Ficheros directos, 455
- Ficheros en redes, 487
- Ficheros estándar
 - input, 471
 - output, 471
- Ficheros externos, 458
- Ficheros homogéneos, 487
- Ficheros internos, 458
- Ficheros secuenciales, 454
- Ficheros secuenciales indexados, 456
 - área de desbordamiento, 456
 - área de índices, 456
 - área principal, 456
- Ficheros sin tipo, 477
- FIFO (First In First Out), 553
- File, 454
- FilePos, 112
- FileRec, 481
- FileSize, 112
- Filmadora, 31
- Flujo o Stream
 - acceso aleatorio, 871
 - definición, 864
 - inicialización, 865
 - introducción de objetos, 865
 - manejo de errores, 867
 - recuperación de objetos, 866
 - registro de declaración, 870
 - streams de Turbo Vision, 865
- Flush, 112
- FOR
 - estructura, 162
 - sentencia, 173
- Formatos raster, 28
- Formatos vectoriales, 28
- Forward, 266
- Fragmentación
 - de la memoria heap, 562
- Frameworks, 61, 64
- Frecuencia de reloj, 10
- FreeList, 579
- FreeMem, 562
- Fuentes de letra, 953
 - laser, 24
- Funciones API, 963, 965
 - Ellipse, 910
 - MessageBox, 913
 - TextOut, 911
- Funciones callback, 967

- Gauss, 342
- Gauss-Jordan
 - método de, 336
- GDI, 963
- GDI, Graphics Device Interface, 965
- GetAspectRatio, 275
- GetColor, 275
- GetDir, 112
- GetImage, 275
- GetMaxColor, 275
- GetMaxX, 275
- GetMaxY, 275
- GetMem, 562
- GetPalette, 275
- GetPaletteSize, 275
- GetPixel, 275
- GIF, 28
- Gigabyte, 3
- GIS, 52
- GOTO
 - sentencia, 191
- GotoXY, 114
- Grafo, 555
 - aristas, 555
 - vértices, 555
- GraphResult, 276
- GUI, 732, 888

- Handle, 482
 - WinTypes, 897
- Hardware, 6
- Hash, 633
 - abierta, 634
 - cerrada, 634
- Heap, 576, 702
- HeapError, 579
- HeapOrg, 577
- HeapPtr, 577
- Herencia, 62, 670
 - antepasado, 673

- ascendiente, 673
- descendiente, 673
- dominio, 673
- ejemplos, 675, 678
- INHERITED, 677
- múltiple, 674
- propiedad transitiva, 673
- redefinir, 677
- simple, 674
- sintaxis, 674
- High, 347
- HighVideo, 115
- Hojas de cálculo, 43
- Host, 48
- I/O, 7
- IBM SAA/CUA, 889, 919, 955, 229
- Iconos, 952
- IDE, Entorno Integrado de Desarrollo, 75, 92, 275, 164
- IDE, Integrated Drive Electronics, 19
- IF THEN
 - anidada o jerarquizada, 170
 - estructura, 159
 - sentencia, 166
- IF THEN ELSE
 - estructura, 159
 - sentencia, 168
- ImageSize, 275
- Implementación
 - de métodos, 658
- Impresora
 - salida a impresora, 117
- Impresoras, 22
 - de cadena, 24
 - de caracteres, 22
 - de chorro de tinta, 23
 - de líneas, 23
 - de página, 23
 - laser, 24
 - margarita, 23
 - matriciales, 23
- Indice
 - de método dinámico, 689
 - de un array, 301
- Informática, 1
- Ingeniería del Software, 1002
- INHERITED, 677
- Init
 - constructor, 903
- InitGraph, 275
- Inline, 432
- Inorden, 559
- Insert, 351
- InsLine, 115
- Inspector de tipos objeto (browser), 676
- Instancia, 906
- Instanciación de objetos, 658
- Instancias, 61
- Integración numérica, 342
- Interface, 730
- Intérpretes, 40
- Interrupciones, 434
- IoResult, 476
- IOResult, 112
- Jacopini, 67
- Jerarquía de clases, 60, 63
- Kay, Alan, 888
- KeyPressed, 115
- Kilobyte, 3
- LAN, 487
- LAN, red local, 31
- Lápiz óptico, 28
- Late binding
 - ligadura tardía, 682
- Length, 351
- Lenguaje ensamblador, 37
- Lenguaje máquina, 37
- Lenguaje natural, 64
- Lenguajes concurrentes, 38
- Lenguajes de alto nivel, 37
- Lenguajes de medio nivel, 37
- Lenguajes de programación
 - Tipos, 36
- Lenguajes declarativos, 38
- Lenguajes funcionales, 38
- Lenguajes imperativos, 37
- Lenguajes lógicos, 38
- Lenguajes orientados a objetos, 37
 - híbridos, 653
 - puros, 653
- Librería, 265, 269, 272, 273, 276, 279
- LIFO (Last In First Out), 553
- Ligadura tardía, 682
- Ligadura temprana, 682
- Line, 275
- Líneas por minuto (lpm), 23
- LineRel, 275
- LineTo, 275
- Linker, 40
- Lista circular, 554
- Lista de exportaciones, 298
- Lista de importaciones, 298
- Lista de libres, 579
- Lista encadenada, 527
- Lista simplemente enlazada, 534
 - búsqueda, 542
 - creación, 538
 - inserción de nodos, 544
 - lista circular, 554
 - lista FIFO, 553

- lista LIFO, 553
- lista ordenada, 546
- meter (push), 553
- recorrido, 541
- sacar (pop), 553
- supresión de nodos, 548
- Low, 347
- LowVideo, 115
- lst, 117

- Macroensamblador, 265
- Magnitud
 - límite de, 5
- Mainframes, 13
- Mantenimiento de programas, 76
- Mapa de bits, 949
- Máquina de caracteres, 184
- Marco de aplicación
 - control, 731
 - definición, 730
 - eventos, 731
 - métodos, 731
 - modelo, 731
 - vista, 731
- Marcos de trabajo (frameworks), 64
- MaxAvail, 562
- MDI, 919, 971
- Megabyte, 3
- MegaHercios, 10
- MemAvail, 562
- Memoria
 - de video, 15
 - principal, 7
- Memoria dinámica, 575
- Mensajes, 968
 - de comandos, 975
 - de notificación, 975
- MessageLoop
 - método, 904
- Metalenguajes, 87
- Método, 654
 - constructor, 687
 - destructor, 697
- Método de aproximaciones sucesivas, 197
- Método de Cuadratura de Gauss, 342
- Método de Gauss-Jordan, 336
- Método de la burbuja, 308
- Método de Newton-Raphson, 200
- Métodos, 61
 - abstractos, 678
 - ámbito, 660
 - constructores, 681
 - destructores, 681
 - dinámicos, 682, 689, 909
 - estáticos, 682
 - virtuales, 681, 682
- Métodos Turbo Vision
 - abstractos, 769
 - estáticos, 768
 - pseudo-abstractos, 769
 - virtuales, 768
- MFLOPS, 13
- MHz, 10
- Microordenador, 9
- Microprocesador, 9
- MIPS, 12
- MkDir, 112
- Modem, 28
- Modo gráfico, 14
- Modo texto, 14
- Modulación, 28
- Modularización, 298
- Montador de enlaces (linker), 40
- Montón (heap), 576
- Motores de Turbo Vision
 - colecciones, 786
 - listas de cadenas, 788
 - objetos de validación, 788
 - recursos, 786
 - streams, 785
- Multimedia, 53

- Nassi-Shneiderman, 69
- New
 - extensiones al procedimiento, 693
 - función, 696
- Newton-Raphson
 - método de, 200
- NIL, 531
- NormVideo, 115
- NoSound, 115
- NOT, 124
- Notación algorítmica, 72
- Notación húngara, 964

- OBJECT, 655, 656
- Object Pascal, 652
- ObjectWindows, 900
- Objeto, 60, 61, 654
- Objetos, 658
- Objetos de control
 - botones de radio, 742
 - botones individuales, 742
 - casillas de verificación, 742
- Objetos polimórficos, 682
- OCR, 28
- Ocultación de información, 331
 - parámetro, 665
- OEM, 985
- Ofs, 563
- OLE, 889, 900, 971, 224
- OOA
 - Análisis Orientado a Objetos, 999
- OOD

- Diseño Orientado a Objetos, 999
- OOP
 - Programación Orientada a Objetos, 999
- Open parameters, 347
- OpenString, 348
- Operador @, 561
 - Aplicado a una variable, 561
 - Aplicado al nombre de un subprograma, 561
- Operador concatenación, 351
- Operadores
 - lógicos de manejo de bits, 572
- OR, 124
- Ordenador, 2
- Organigramas, 64
 - estructurados, 70
- OutText, 275
- Override, redefinir, 677
- PACKED
 - array, 322
 - en Turbo Pascal, 347
 - registro, 409
- Páginas por minuto (ppm), 23
- Palabras reservadas, 86
- Palanca de juegos, 30
- Paleta, 15
- Pantalla, 14
- Papel continuo, 22
- Paquetes integrados, 48
- Paradigma MVC
 - bucle principal de eventos, 735
 - control, 734
 - controlador, 733
 - modelo, 733, 734, 735
 - programa, 733
 - vista, 733, 734, 735
 - vista principal, 735
- Parámetros abiertos, 347
- Parámetros array abiertos, 348
- Parámetros string abiertos, 348
- PC
 - ordenador personal, 9
- PCHar, 561
- PCX, 28
- PE, 999
 - programación estructurada, 998
- Periféricos
 - ráster, 268
 - vectoriales, 268
- pila LIFO (Last In First Out), 553
- Pilas, 553
- Pista, 17
- Pixel, 14, 268, 275
- Plotter, 268
- Plotter, trazador gráfico, 25
 - electroestático o raster, 26
 - vectorial, 25
- Pointer, 528
 - tipo de puntero genérico, 560
- Polimorfismo, 62, 681
- POO, 651, 998, 1000
 - definición de Booch, 1001
 - Programación Orientada a Objetos, 999
- Pop (sacar), 553
- Portapapeles, 889, 984
- Pos, 351
- Postorden, 559
- Precisión
 - límite de, 5
- Prefijo de segmento de programa, 431
- Preorden, 559
- Primitivas gráficas, 272
- Printer
 - salida a impresora, 117
- PRIVATE, 665
- Procesadores
 - 8080, 1101
 - 8086, 1101
 - Intel MDS, 1101
 - Z80, 1101
- Procesadores de texto, 42
- Programa, 2, 36
 - Estructura completa, 104
 - Estructura completa en Turbo Pascal, 111
- Programación, 2
- Programación dirigida por eventos, 890, 901
 - bucle principal de eventos, 748
 - categorías de eventos, 747
 - control principal, 748
 - definición, 747
 - eventos, 747
 - vista principal, 748
- Programación estructurada, 999
 - teorema, 67
- Programación orientada a objetos, 901, 1000
 - encapsulación, 1002, 1004
 - herencia, 1002, 1006
 - polimorfismo, 1002, 1007
- Programas de comunicaciones, 46
- Programas residentes
 - definición de interrupciones, 1103
 - encadenamiento de interrupciones, 1104
 - interrupciones con Turbo Pascal, 1105
 - prefijo de segmento de programa, 1103
 - procedimientos de interrupción, 1105
 - programas residentes desde Turbo Pascal, 1106
 - rutinas de servicio de interrupción, 1102
 - TSR, 1101
 - vector de interrupciones, 1101
- PROM, 11
- Prueba de programas, 74
- Pseudocódigo, 70

PSP
 prefijo de segmento de programa, 1104
 PTAD
 PTAD versus POO, 1001
 Ptr, 563
 PUBLIC, 665
 Puntero, 6, 528
 Punteros
 genéricos, 560
 Push (meter), 553
 PutImage, 275
 PutPixel, 275

 Raíces de ecuaciones, 196
 RAM, 10
 Randell, 59
 Ranuras de expansión (slots), 10
 Raster, 28
 Ratón, 30
 Read, 98
 Turbo Pascal, 110
 ReadKey, 115
 ReadLn, 100
 RECORD, 406
 Recorte (Clipping), 271
 Rectangle, 275
 Recursividad, 246
 visión recursiva de una lista, 551
 Recursos, 899, 946, 988
 de información de versión, 958
 definidos por el usuario, 958
 Red local, 31
 Redefinición de métodos, 676
 virtuales, 683
 Registro
 empaquetado, 409
 estructura de datos, 406
 jerárquico, 410
 variante, 415
 Registros del microprocesador, 427
 generales, 428
 otros registros, 431
 punteros e índices, 429
 registros de segmento, 430
 Rename, 112
 REPEAT
 comparación de WHILE y REPEAT, 179
 REPEAT UNTIL
 estructura, 161
 sentencia, 179
 Resolución gráfica, 14
 Resource workshop, 946
 Reubicación, 571
 RIFF, 985
 RISC, 12
 RmDir, 112
 ROM, 11

 Run
 método, 904

 Scanner, 28
 SCSI, Small Computer System Interface, 19
 SDK, 997
 Sectores, 17
 Secuencia, 182
 esquemas de recorrido, 182
 Secuencias de escape, 480
 Seek, 112
 SeekEof, 112
 SeekEoln, 112
 Seg, 563
 Selector de campo, 410
 Self, 698
 parámetro, 660
 Señales analógicas, 31
 Sentencia
 asignación, 90
 SetAspectRatio, 275
 SetColor, 275
 SetLineStyle, 275
 SetPalette, 275
 SetTextBuf, 112
 SetTextJustify, 275
 SetTextStyle, 275
 SetViewPort, 275
 Shl, 573
 Shr, 573
 Símbolos
 no terminales, 87, 1018
 terminales, 87, 1018
 Simonyi, Charles, 964
 Simula 67, 652
 Síncrono, 47
 Sintaxis
 diagramas sintácticos, 87
 notación EBNF, 87
 por medio de colores, 92
 Sistema de coordenadas
 coordenadas globales, 789
 coordenadas locales, 789
 Sistema operativo, 456
 Sistemas de coordenadas, 268
 Sistemas de ventanas
 COWS, 732
 GUI, 732
 Sistemas expertos, 53
 Sistemas operativos, 33
 Apple DOS, 1100
 componentes del S.O. CP/M, 1101
 CP/M, 1100
 DOS, 1101
 North Star DOS, 1100
 TRSDOS, 1100
 SizeOf, 347

- Smalltalk, 888
- Software, 32
 - a medida, 40
 - de aplicación, 40
- Soporte físico
 - hardware, 6
- Sound, 115
- SPtr, 563
- Sseg, 563
- Stack
 - pila LIFO, 553
- Str, 351
- StrCat, 564
- StrComp, 564
- StrCopy, 565
- StrDispose, 564
- StrECopy, 565
- StrEnd, 565
- StrIComp, 565
- STRING, 329
 - en Turbo Pascal, 349
- Strings
 - unit, 564
- Strings (cadenas)
 - terminadas en caracter nulo, 564
- StrLCat, 565
- StrLComp, 565
- StrLCopy, 565
- StrLen, 565
- StrLIComp, 565
- StrLower, 565
- StrMove, 565
- StrNew, 564
- StrPas, 565
- StrPCopy, 565
- StrPos, 565
- StrRScan, 566
- StrScan, 566
- StrUpper, 566
- Subíndice
 - de un array, 301
- Subprograma
 - anidamiento, 258
 - externo, 264
 - interno, 264
- Subprogramas, 238
 - funciones, 239
 - procedimientos, 248
- Subvista
 - activa, 804
 - árbol de vistas, 801
 - cadena de foco, 804, 813
 - definición, 801
 - enfocada, 804
 - inserción, 803
 - orden-Z, 802
 - seleccionada, 804
- Super VGA, 16
- Superordenadores, 13
- Tabla
 - array de registros, 413
- Tabla de bloqueos, 488
- Tabla de interrupciones, 434
- Tabla de Métodos Dinámicos, 689, 712
- Tabla de Métodos Virtuales, 687, 689, 703, 199
- Tablas de cadenas, 957
- Tableta digitalizadora, 27
- TAD, 331
 - PTAD versus POO, 1001
- Tag-field, 415
- Taller de recursos, 923, 946
- TApplication, 902
- Tarjetas de entrada/salida de video, 31
- Tarjetas de sonido, 31
- TASM, 265
- TBufStream, 945
- TButton, 925
- TCheckBox, 926
- TCollection, 942
- TComboBox, 931
- TControl, 924
- TDialog, 934
- TDlgWindow, 935
- TDosStream, 944
- Tecla aceleradora, 735
- Teclado, 13
- Teclas aceleradoras, 948
- TEdit, 927
- TEditPrintout, 939
- TEditWindow, 914
- TEmsStream, 945
- Terabyte, 3
- TextBackGround, 116
- TextColor, 116
- TextMode, 116
- TFileDialog, 937
- TFileValidator, 941
- TFileWindow, 915
- TGroupBox, 924
- Tiempo de compilación, 40
 - ligadura temprana, 682
- Tiempo de ejecución, 40
 - ligadura tardía, 682
- TIFF, Tag Image File Format, 28
- TInputDialog, 936
- Tipo abstracto de datos, 331, 654
- Tipo Abstracto de Datos
 - lista, 535
 - registro, 423
- Tipo array, 301
- Tipo base, 301
- Tipo boolean

- expresiones booleanas, 126
- Tipo caracter
 - constantes, 130
 - variables, 131
- Tipo componente de un array, 301
- Tipo entero
 - constantes enteras, 120
 - expresiones enteras, 121
 - variables enteras, 120
- Tipo enumerado, 146
- Tipo índice de un array, 301
- Tipo objeto (object), 654
- Tipo PChar, 330
- Tipo real
 - error representacional, 136
 - errores computacionales, 138
 - precisión, 133
 - rango, 133
- Tipo selector, 415
- Tipo subíndice, 302
- Tipo subrango, 150
- Tipos objeto abstractos, 678, 704
- TListBox, 929
- TLookupValidator, 941
- TMD, 689
- TMDIClient, 923
- TMDIWindow, 919
- TMemoryStream, 944
- TMessage, 968, 973
- TMsg, 969
- TMV, 689, 698, 711
- TObject, 902
- Token, 110
- Top-down, 59
- TPrintDialog, 936
- TPrinter, 938
- TPrinterAbortDlg, 935
- TPrinterSetupDlg, 935
- TPrintOut, 939
- TPU, 277
- TPXPictureValidator, 940
- TRadioButton, 927
- Traductor, 39
- TRangeValidator, 941
- Transformación
 - escalado, 269
 - rotación, 270
 - traslación, 270
- Tratamiento secuencial de la información, 181
- Truncate, 112
- TScrollBar, 932
- TScroller, 937
- TSortedCollection, 942
- TSR, 1101
- TStatic, 927
- TStrCollection, 943
- TStream, 943
- TStringCollection, 943
- TStringLookupValidator, 941
- Turbo Vision, 64
 - barra de desplazamiento, 759
 - barra de menús, 758
 - comandos, 817
 - cuadros de diálogo, 759
 - elementos de menu, 759
 - escritorio, 758
 - evento, 758, 808
 - grupo, 757, 800
 - línea de estado, 759
 - motores, 758, 785
 - objetos abstractos, 760
 - objetos primitivos, 770
 - sistema de coordenadas, 789
 - subvista, 801
 - teclas aceleradoras, 759
 - ventanas, 759
 - vistas, 757, 773
 - vistas de grupo, 781
- TValidator, 940
- TWindow, 909
- TWindowPrintout, 939
- TWindowsObject, 907
- TYPE, 146
- Unidad central de proceso
 - CPU, 7
- Unión libre, 416
- Unit, 277, 332
 - Crt, 892
 - Dos, 895
 - Objects, 902, 942, 943, 944, 945
 - ocultación de información, 665, 668, 669, 158
 - ODialogs, 924, 925, 926, 927, 929, 931, 932, 934, 935
 - OPrinter, 935, 936, 938, 171
 - OStdDlgs, 936, 937
 - OStdWnds, 914, 915
 - OWindows, 902, 909, 919, 924, 937
 - Printer, 896
 - tipos object, 668
 - Validate, 940, 941
 - WinCrt, 892
 - WinDos, 895
 - WinPrn, 896
 - WinProcs, 963, 965
 - WinTypes, 897, 963, 965
- UpCase, 352
- Uses, 113
- Utilizar los tipos objeto Turbo Vision
 - aplicaciones, 832

- colecciones, 861
- controles, 850
- cuadros de diálogo, 849, 855
- desktop, 835
- fondo del desktop, 835
- líneas de estado, 844
- menús, 839
- modos de pantalla, 834
- procesos background, 835
- recursos, 871
- shell al DOS, 834
- streams, 864
- subsistemas de una aplicación, 833
- tiempo muerto, 827, 835
- ventanas, 846

Val, 351

VAR, 89

Variable de entorno

- COMSPEC, 834
- PATH, 881

Variable dinámica, 532

Variable referenciada, 529, 532

Vectores de interrupción, 434

Vectorial, 28

Vectorizador, 28

Ventana

- librería de, 731

Ventana marco, 919

Ventanas gráficas (viewports), 271

Ventanas hijas, 919

VGA, 16

VIRTUAL, 683

Vista

- definición, 731, 733, 734
- desktop, 738
- vista principal, 735

Vistas de grupo de Turbo Vision

- aplicaciones, 783
- cuadros de diálogo, 784
- desktop, 783
- grupo abstracto, 782
- subvistas, 800
- ventanas, 784

Vistas de Turbo Vision

- barras de desplazamiento, 779
- botones, 774
- botones de radio, 775
- campo de edición, 777
- campos de opciones de vista, 795
- casillas de verificación, 775
- clusters, 775
- como utilizarlas, 793
- cursor de una vista, 797
- dispositivos de texto, 780
- distribución en cascada, 835
- distribución en mosaico, 835
- estado de una vista, 796
- historia, 777
- inicializar una vista, 793
- límites de una vista, 794
- línea de estado, 781
- marcos, 774
- mensajes entre vistas, 830
- menús, 776
- movimiento de una vista, 794
- redimensionar una vista, 794
- texto estático, 780
- validación, 799
- visor de lista, 778
- vista con scroll, 779
- vista intermediaria, 829
- vista modal, 775, 808
- vista propietaria, 801
- visualizar una vista, 797

VMT (Tabla de Métodos Virtuales), 711

WAN, 31

WhereX, 117

WhereY, 117

WHILE

- comparación de WHILE y REPEAT, 179
- notación EBNF, 164
- sentencia, 163

WHILE DO

- estructura, 161

Win16, 899

Win32, 899

Win32s, 899

Window, 116

Windows/NT, 888, 899

Wirth, 59, 298

WITH, 411

- con objetos, 659
- multirregistro, 412

WMRA, 21

WORM, 21

Write, 102

Writeln, 102, 103

WYSIWYG, 43

XMS, 11

- unit XMS, 1117

XOR, 573

Zurcher, 59

TABLA DE CONTENIDOS

INTRODUCCION A LA INFORMATICA	1
1.1 INTRODUCCION	1
1.2 REPRESENTACION DE LA INFORMACION	2
1.3 HARDWARE O SOPORTE FISICO	6
1.4 SOFTWARE O SOPORTE LOGICO	32
1.5 CUESTIONES	54
1.6 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	54
CONSTRUCCION DE PROGRAMAS	57
2.1 INTRODUCCION	57
2.2 LAS FASES DEL PROCESO DE PROGRAMACION	58
2.3 ANALISIS DEL PROBLEMA	58
2.4 DESARROLLO DE LA SOLUCION	58
2.5 TECNICAS DE DESCRIPCION DE ALGORITMOS	64
2.6 CONSTRUCCION DE LA SOLUCION EN FORMA DE PROGRAMA	73
2.7 PRUEBA DE PROGRAMAS	74
2.8 DOCUMENTACION DE PROGRAMAS	76
2.9 MANTENIMIENTO DE PROGRAMAS	76
2.10 EJERCICIOS RESUELTOS	78
2.11 EJERCICIOS PROPUESTOS	79
2.12 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	80
INTRODUCCION AL LENGUAJE PASCAL	81
3.1 EL LENGUAJE PASCAL	82
3.2 LOS DATOS Y SUS TIPOS	82
3.3 IDENTIFICADORES, CONSTANTES Y VARIABLES	86
3.4 DEFINICIONES Y DECLARACIONES	89
3.5 LA SENTENCIA DE ASIGNACION	90
3.6 CARACTERISTICAS DE TURBO PASCAL	92
3.7 PALABRAS RESERVADAS DE TURBO PASCAL	93
3.8 DIRECTIVAS ESTANDAR DE TURBO PASCAL	93
3.9 CUESTIONES RESUELTAS	93
3.10 CUESTIONES PROPUESTAS	94
3.11 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	95
ENTRADA / SALIDA	97
4.1 ENTRADA ESTANDAR	97
4.2 SALIDA ESTANDAR	102
4.3 ESTRUCTURA COMPLETA DE UN PROGRAMA	104
4.4 EJERCICIOS RESUELTOS	106
4.5 EXTENSIONES DEL COMPILADOR TURBO PASCAL	110
4.6 EJERCICIOS PROPUESTOS	118
TIPOS DE DATOS SIMPLES	119
5.1 INTRODUCCION	119
5.2 TIPO ENTERO	120
5.3 TIPO BOOLEAN	124
5.4 TIPO CARACTER	129
5.5 TIPO REAL	133
5.6 TIPOS DEFINIDOS POR EL USUARIO	145
5.7 AMPLIACION DEL PASCAL ESTANDAR CON TURBO PASCAL	152
5.8 CUESTIONES Y EJERCICIOS RESUELTOS	153
5.9 CUESTIONES Y EJERCICIOS PROPUESTOS	154
5.10 AMPLIACIONES y NOTAS BIBLIOGRAFICAS	156

ESTRUCTURAS DE CONTROL	157
6.1 INTRODUCCION	158
6.2 LA ESTRUCTURA REPETITIVA <i>WHILE</i>	163
6.3 LA ESTRUCTURA ALTERNATIVA <i>IF-THEN</i>	166
6.4 LA ESTRUCTURA ALTERNATIVA <i>IF-THEN-ELSE</i>	168
6.5 LA ESTRUCTURA REPETITIVA <i>FOR</i>	173
6.6 LA ESTRUCTURA REPETITIVA <i>REPEAT-UNTIL</i>	179
6.7 TRATAMIENTO SECUENCIAL DE LA INFORMACION	181
6.8 LA ESTRUCTURA MULTIALTERNATIVA <i>CASE</i>	186
6.9 SENTENCIA <i>GOTO</i>	191
6.10 APLICACION AL CALCULO NUMERICO. DETERMINACION DE RAICES DE ECUACIONES	196
6.11 EXTENSIONES DEL COMPILADOR TURBO PASCAL	206
6.12 CUESTIONES Y EJERCICIOS RESUELTOS	207
6.13 CUESTIONES Y EJERCICIOS PROPUESTOS	232
6.14 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	235
SUBPROGRAMAS	237
7.1 INTRODUCCION	238
7.2 FUNCIONES	239
7.3 PROCEDIMIENTOS	248
7.4 TRANSFORMACION DE FUNCIONES EN PROCEDIMIENTOS	256
7.5 ANIDAMIENTO DE SUBPROGRAMAS	258
7.6 EFECTOS LATERALES	261
7.7 SUBPROGRAMAS INTERNOS Y EXTERNOS. BIBLIOTECAS	264
7.8 DECLARACION FORWARD	266
7.9 PROGRAMACION GRAFICA	267
7.10 CONTROL DE PANTALLA ALFANUMERICA	276
7.11 TIPOS PROCEDURALES DE TURBO PASCAL	276
7.12 UNITS DE TURBO PASCAL	277
7.13 EJERCICIOS RESUELTOS	279
7.14 CUESTIONES Y EJERCICIOS PROPUESTOS	295
7.15 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	298
ESTRUCTURA DE DATOS ARRAY	299
8.1 INTRODUCCION	300
8.2 ARRAYS BIDIMENSIONALES	311
8.3 OPERACIONES CON ARRAYS COMPLETOS	317
8.4 ARRAYS MULTIDIMENSIONALES	319
8.5 ARRAYS EMPAQUETADOS	322
8.6 CADENAS DE CARACTERES	323
8.7 EXTENSION STRING	329
8.8 CONCEPTO DE TIPO ABSTRACTO DE DATOS (TAD)	331
8.9 LOS ARRAYS COMO TADs	333
8.10 APLICACIONES AL CALCULO NUMERICO	336
8.11 EXTENSIONES DEL COMPILADOR TURBO PASCAL	347
8.12 CUESTIONES Y EJERCICIOS RESUELTOS	353
8.13 EJERCICIOS PROPUESTOS	384
8.14 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	388
CONJUNTOS	391
9.1 CONJUNTOS EN PASCAL	391
9.2 CONSTRUCCION DE CONJUNTOS	392
9.3 OPERACIONES CON CONJUNTOS	393
9.4 EXPRESIONES LOGICAS CON CONJUNTOS	395
9.5 REPRESENTACION INTERNA DE LOS CONJUNTOS	396
9.6 EJERCICIOS RESUELTOS	397

9.7 CUESTIONES Y EJERCICIOS PROPUESTOS	402
9.8 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	403
ESTRUCTURA DE DATOS REGISTRO	405
10.1 CONCEPTO DE REGISTRO	406
10.2 PROCESAMIENTO DE REGISTROS	409
10.3 REGISTROS JERARQUICOS	410
10.4 SENTENCIA WITH	411
10.5 ARRAYS DE REGISTROS: TABLAS	413
10.6 REGISTROS VARIANTES	415
10.7 LOS REGISTROS COMO TIPOS ABSTRACTOS DE DATOS	423
10.8 REPRESENTACION INTERNA DE LOS REGISTROS	426
10.9 EXTENSIONES DEL COMPILADOR TURBO PASCAL	427
10.10 REGISTROS DEL MICROPROCESADOR. INTERRUPCIONES	427
10.11 CUESTIONES Y EJERCICIOS RESUELTOS	440
10.12 CUESTIONES Y EJERCICIOS PROPUESTOS	449
10.13 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	452
FICHEROS	453
11.1 INTRODUCCION	454
11.2 DEFINICION DE FICHEROS EN PASCAL	457
11.3 FICHEROS INTERNOS Y EXTERNOS	457
11.4 PROCESO DE ESCRITURA EN FICHEROS	458
11.5 LECTURA DE FICHEROS	460
11.6 BUFFER DE FICHERO	465
11.7 FICHEROS DE TEXTO	467
11.8 LOS FICHEROS ESTANDAR INPUT Y OUTPUT	471
11.9 EXTENSIONES DEL COMPILADOR TURBO PASCAL	471
11.10 REPRESENTACION INTERNA DE LOS FICHEROS	481
11.11 LOS FICHEROS COMO TIPOS ABSTRACTOS DE DATOS	483
11.12 FICHEROS HOMOGENEOS Y NO HOMOGENEOS	487
11.13 MANEJO DE FICHEROS EN REDES	487
11.14 EJECICIOS RESUELTOS	494
11.15 EJERCICIOS PROPUESTOS	521
11.16 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	523
ESTRUCTURAS DINAMICAS DE DATOS	525
12.1 INTRODUCCION	526
12.2 EL TIPO PUNTERO	528
12.3 ESTRUCTURAS DINAMICAS DE DATOS LINEALES	534
12.4 ALGORITMOS DE TRATAMIENTO DE LISTAS SIMPLEMENTE ENLAZADAS	537
12.5 VISION RECURSIVA DE LA LISTA	551
12.6 OTROS TIPOS DE LISTAS: PILAS, COLAS, LISTAS CIRCULARES	553
12.7 ESTRUCTURAS DINAMICAS DE DATOS NO LINEALES	555
12.8 EXTENSIONES DEL COMPILADOR TURBO PASCAL	560
12.9 GESTION DE MEMORIA DINAMICA EN TURBO PASCAL	575
12.10 EJERCICIOS RESUELTOS	580
12.11 EJERCICIOS PROPUESTOS	643
12.12 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	648
PROGRAMACION ORIENTADA A OBJETOS	651
13.1. INTRODUCCION	651
13.2. LENGUAJES ORIENTADOS A OBJETOS	652
13.3. CONCEPTOS BASICOS DE POO	653
13.4. ENCAPSULACION	654
13.5. OCULTACION DE INFORMACION	665

13.6. HERENCIA	670
13.7. POLIMORFISMO	681
13.8. OBJETOS DINAMICOS	692
13.9. ABSTRACCION	703
13.10. GENERICIDAD	704
13.11. REPRESENTACION INTERNA DE LOS TIPOS OBJETO	711
13.12. EJERCICIOS RESUELTOS	712
13.13. EJERCICIOS PROPUESTOS	725
13.14. AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	727
MARCOS DE APLICACION Y PROGRAMACION DIRIGIDA POR EVEN- TOS CON TURBO VISION	729
14.1 INTRODUCCION	730
14.2 MARCOS DE APLICACION	731
14.3 PROGRAMACION DIRIGIDA POR EVENTOS	746
14.4 TURBO VISION: UN MARCO DE APLICACION EN MODO TEXTO	754
14.5 LAS VISTAS EN TURBO VISION	793
14.6 LOS GRUPOS EN TURBO VISION	800
14.7 EVENTOS EN TURBO VISION	808
14.8 UTILIZACION DE LOS TIPOS OBJETO DE TURBO VISION	832
14.9 EJERCICIOS PROPUESTOS	883
14.10 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	884
PROGRAMACION EN ENTORNO WINDOWS®	887
15.1 INTERFACES GRAFICOS DE USUARIO	888
15.2 EL ENTORNO WINDOWS	889
15.3 PROGRAMACION DIRIGIDA POR EVENTOS	890
15.4 TRANSICION RAPIDA A WINDOWS	891
15.5 TIPOS DE DATOS DE WINDOWS	897
15.6 LA BIBLIOTECA OBJECTWINDOWS®	900
15.7 LOS RECURSOS	946
15.8 LAS FUNCIONES API DE WINDOWS	963
15.9 LOS MENSAJES	968
15.10 EL PORTAPAPELES	984
15.11 LAS BIBLIOTECAS DE ENLACE DINAMICO (DLL)	987
15.12 INTERCAMBIO DINAMICO DE DATOS (DDE)	992
15.13 OBJETOS DE ENLACE E INCLUSION (OLE)	992
15.14 EJERCICIOS RESUELTOS	993
15.15 EJERCICIOS PROPUESTOS	996
15.16 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	997
PROGRAMACION ESTRUCTURADA <i>VERSUS</i> PROGRAMACION ORIEN- TADA A OBJETOS	998
16.1 INTRODUCCION	998
16.2 PROGRAMACION ESTRUCTURADA (PE)	999
16.3 PROGRAMACION ORIENTADA A OBJETOS (POO)	1000
16.4 COMPARACION	1004
16.5 CONCLUSIONES	1007
16.6 AMPLIACIONES Y NOTAS BIBLIOGRAFICAS	1008
ANEXO I: CONJUNTOS DE CARACTERES	1010
ANEXO II: DIAGRAMAS SINTACTICOS	1018
ANEXO III: NOTACION EBNF	1026
ANEXO IV: INTERFAZ DE LAS UNITS DE TURBO VISION	1032
ANEXO V: INTERFAZ DE LAS UNITS DE OBJECT WINDOWS	1080
ANEXO VI: PROGRAMAS RESIDENTES	1100
ANEXO VII: MEMORIA EXTENDIDA	1116

TABLA DE FIGURAS

1.1 El ordenador y sus periféricos	7
1.2 Microprocesador	9
1.3 Ordenador personal	10
1.4 Conexiones de un ordenador personal	10
1.5 Teclado	14
1.6. Resolución y pixels	15
1.7 Pixels y colores	15
1.8 Grabación y lectura de un soporte magnético	18
1.9 Unidad de lectura/escritura de disquetes	18
1.10 Disquete de cinco pulgadas y cuarto.	19
1.11 Disquete de tres pulgadas y media	19
1.12 Esquema de un disco duro	20
1.13 Esquema de una unidad de cinta	21
1.14 Disco CD-ROM	22
1.15 Papel continuo	22
1.16 Impresora matricial	23
1.17 Impresoras laser	24
1.18 Trazador gráfico de tambor	25
1.19 Esquema de tableta digitalizadora	26
1.20 Menú de tableta	27
1.21 Modem externo	29
1.22 Diversos códigos de barras	29
1.23 Palanca de juegos o joystick	30
1.24 Ratón	30
1.25 Red Local	32
1.26 Situación del sistema operativo	33
1.27 Diagrama en T de un traductor	39
1.28 Tiempo de compilación	39
1.29 Tiempo de ejecución	40
1.30 Fases de compilación, montaje y ejecución	41
1.31 Transmisión de datos en paralelo	46
1.32 Transmisión de datos en serie	47
1.33 Conector RS-232C de 26 pines	48
1.34 Entorno Windows	49
1.35 Diseño de viviendas	50
1.36 Reconstrucción de monumentos histórico-artísticos	51
1.37 Reconstrucción de monumentos histórico-artísticos	52
1.38 Ejemplo de consulta en un GIS	52
2.1 Esquema de análisis descendente	59
2.2 Ejemplo de análisis descendente	60
2.3 Ejemplo de jerarquía de clases	63
2.4 Símbolos de los diagramas de flujo	65
2.5 Estructuras de control	66
2.6 Estructuras de control alternativas	67
2.7 Estructuras de control repetitivas	67
2.8 Ejemplo de diagrama de flujo	68
2.9 Diagramas de Nasi/Shneiderman o de Chapin	69
2.10 Ejemplo con los diagramas de Nasi/Shneiderman	70
2.11 Prueba de programas	75
2.12 Relación entre desarrollo y mantenimiento de programas	77
2.13 Ciclo de vida de una aplicación informática	77
3.1 Diagrama sintáctico de un identificador	88
3.2 Diagrama sintáctico para definir constantes	89
3.3 Diagrama sintáctico para declarar variables	90
3.4 Diagrama sintáctico de la sentencia de asignación	91

4.1 Diagrama sintáctico de Read y Readln	98
4.2 Diagrama Sintáctico de Write y Writeln	102
4.3 Estructura de un programa en Turbo Pascal	111
6.1 Organigrama de la estructura secuencial	158
6.2 Organigrama de la estructura IF THEN	159
6.3 Organigrama de la estructura IF THEN ELSE	160
6.4 Organigrama de la estructura CASE	160
6.5 Organigrama de la estructura WHILE DO	161
6.6 Organigrama de la estructura REPEAT	162
6.7 Organigrama de la estructura FOR	163
6.8 Diagrama sintáctico de WHILE	163
6.9 Diagrama sintáctico de una sentencia compuesta	164
6.10 Diagrama sintáctico de IF THEN	166
6.11 Diagrama sintáctico de IF THEN ELSE	168
6.12 Estructura If anidada a tres niveles	170
6.13 Ambigüedad de la clausula ELSE	171
6.14 Ejemplo de correspondencia IF-THEN-ELSE	172
6.15 Diagrama sintáctico de FOR	173
6.16 Diagrama sintáctico de REPEAT UNTIL	179
6.17 Comparación de las estructuras WHILE y REPEAT	180
6.18 La máquina de caracteres	185
6.19 Diagrama sintáctico de CASE	186
6.20 Diagrama sintáctico de GOTO	191
6.21 Bifurcación (GOTO) hacia atrás	192
6.22 Bifurcación (GOTO) hacia adelante	192
6.23 Método de aproximaciones sucesivas	198
6.24 Método modificado de aproximaciones sucesivas	199
6.25 Método de Newton-Raphson	201
6.26 Ejemplo de cálculo del área de un triángulo	223
6.27 Ejemplo de cálculo de la superficie de un polígono	226
7.1 División de un problema en subproblemas	238
7.2 División de un programa en subprogramas	238
7.3 Declaración de parámetros	241
7.4 Cabecera de procedimientos y funciones	241
7.5 Llama a una función	243
7.6 Anidamiento de suprogramas	259
7.7 Declaración con la directiva forward	266
7.8 Pixels y resolución	268
7.9 Esquema de traslación de ejes	270
7.10 Esquema de rotación de ejes	271
8.1 Representación de un vector en Informática	300
8.2 ARRAY con tipo subíndice char	301
8.3 Diagrama sintáctico del tipo ARRAY	302
8.4 ARRAY tridimensional	320
8.5 Ahorro de memoria con un ARRAY empaquetado	322
8.6 Reducción de una matriz	375
8.7 Recorrido de una matriz en espiral	378
10.1 Diagrama sintáctico del tipo registro	406
10.2 Representación de un registro	408
10.3 Diagrama sintáctico de WITH	411
10.4 Ejemplo de utilización de una tabla	413
10.5 Diagrama sintáctico de la parte variante de un registro	415
10.6 Utilización de memoria en un registro variante	418
10.7 Registros del microprocesador	428
10.8 Mapa de memoria de un programa de Turbo Pascal	430
10.9 Registro de banderas (flags)	431
11.1 Ficheros y programas	454

11.2	Ficheros secuenciales	454
11.3	Acceso en ficheros secuenciales	455
11.4	Ficheros directos	455
11.5	Acceso en ficheros directos	455
11.6	Definición de ficheros	457
11.7	Esquema de una red en bus	488
12.1	Ejemplo de lista encadenada	527
12.2	Inserción en lista encadenada	528
12.3	Diagrama sintáctico del tipo puntero	528
12.4	Nodo de una lista enlazada	529
12.5	Variable referenciada	530
12.6	Punteros p y q antes de la asignación	531
12.7	Punteros p y q después de la asignación	531
12.8	Lista encadenada acabada en NIL	532
12.9	Puntero y variable referenciada	533
12.10	Diferencia entre $p:=q$ y $p^:=q^$	534
12.11	Nodo de una lista simplemente enlazada	536
12.12	Anatomía de un nodo	537
12.13	Lista vacía	538
12.14	Primer intento de creación de lista enlazada	538
12.15	Intento de añadir otro elemento	539
12.16	Inserción en lista enlazada	540
12.17	Recorrido de una lista	541
12.18	Búsqueda en una lista	542
12.19	Inserción detrás	544
12.20	Inserción delante	545
12.21	Supresión del sucesor de NODO(P)	549
12.22	Supresión del NODO(P)	550
12.23	Supresión de nodos con dos punteros auxiliares	550
12.24	Ejemplo de cola	554
12.25	Lista circular	554
12.26	Lista doblemente enlazada	555
12.27	Ejemplo de grafo	555
12.28	Ejemplo de árbol binario	557
12.29	Mapa de memoria de Turbo Pascal	576
12.30	Almacenamiento de variables dinámicas en el Heap	577
12.31	Liberación de variables dinámicas en el Heap	578
12.32	Aparición de un agujero en el Heap	578
12.33	Aumento y liberación del bloque libre en el Heap	579
12.34	Lista encadenada de números enteros. Ejercicio 12.1	580
12.35	Creación del primer nodo. Ejercicio 12.1	580
12.36	Introducción de datos en el nuevo nodo. Ejercicio 12.1	581
12.37	Reajuste de enlaces. Ejercicio 12.1	581
12.38	Inserción del primer nodo. Ejercicio 12.1	581
12.39	Otro nuevo nodo. Ejercicio 12.1	582
12.40	Reajuste de enlaces (2º nodo). Ejercicio 12.1	582
12.41	Inserción del segundo nodo. Ejercicio 12.1	583
12.42	Lista creada. Ejercicio 12.1	583
12.43	Otra representación de la lista creada. Ejercicio 12.1	584
12.44	p se sitúa al principio de la lista. Ejercicio 12.1	584
12.45	p avanza al siguiente nodo. Ejercicio 12.1	584
12.46	p ha llegado al final de la lista. Ejercicio 12.1	585
12.47	Lista circular. Ejercicio 12.11	602
12.48	Entrelazado de dos listas. Ejercicio 12.14	607
12.49	curva de Von Koch. Estado 0	615
12.50	Curva de Von Koch. Estado 1	616
12.51	Curva de Von Koch. Estado 2	616

12.52 Curva de Von Koch. Estado 5	617
12.53 Lista de Koch. Estado 0	617
12.54 Coordenadas de tres nuevos vértices	618
12.55 Segmento original en el estado n	618
12.56 Segmento anterior en el estado n+1	619
12.57 Lista de Koch. Inserción de tres nuevos nodos	619
12.58 Ejemplo de ejecución. Ejercicio 12.19	624
12.59 Tabla hash abierta	634
12.60 Ejemplo de lista ordenada	644
12.61 Ejemplo de lista circular	645
12.62 Ejemplo de sublista enlazada	645
12.63 Ejemplo de lista circular	647
13.1. Evolución de los lenguajes orientados a objetos	653
13.2. Acceso a los datos a través de los métodos	655
13.3. Diagrama sintáctico simplificado de OBJECT	656
13.4. Diagrama sintáctico de activación de métodos	659
13.5. Iceberg: parte visible (pública) y sumergida (privada)	666
13.6. Diagrama sintáctico con las secciones PUBLIC y PRIVATE	667
13.7. Acceso a los datos a través de los métodos	668
13.9. Esquema de herencia y jerarquía de tipos object	673
13.10. Esquema de herencia múltiple	673
13.11. Diagrama sintáctico del tipo OBJECT con herencia	675
13.12. Jerarquía de los tipos derivados de Tvehiculo	676
13.13. Diagrama sintáctico de uso de INHERITED	677
13.14. Diagrama sintáctico completo del tipo OBJECT	684
13.15. Esquema de herencia con polimorfismo	685
13.16. Pila genérica	705
14.1. Implementación del paradigma MVC	735
14.2. Ventanas de texto y gráficas	737
14.3. Desktop de Turbo Vision con tres subvistas	738
14.4. Menu en modo texto y en modo gráfico	739
14.5. Ejemplos de cajas de diálogo	739
14.6. Vista de texto para editar un fichero	740
14.7. Vista gráfica con texto e imágenes	741
14.8. Funcionamiento de una vista de control	741
14.9. Botones de control	743
14.10. Barras de desplazamiento	744
14.11. Listas de selección	745
14.12. Vista de un editor de texto	745
14.13. Editor de línea para campos de entrada de datos	746
14.14. Cuadro de diálogo	746
14.15. Ejemplo de una caja de diálogo en Turbo Vision	751
14.16. Caja de diálogo con títulos en castellano	752
14.17. Caja de diálogo creada con Object Windows	754
14.18. Elementos de una aplicación Turbo Vision	759
14.19. Jerarquía de tipos objeto de Turbo Vision (I)	760
14.20. Jerarquía de tipos objeto de Turbo Vision (II)	761
14.21. Jerarquía de objetos primitivos de Turbo Vision	771
14.22. Tipos de objetos descendientes directos de TView	773
14.23. Tipos objeto vista de Turbo Vision	774
14.24. Jerarquía de objetos motores de Turbo Vision	785
14.25. Sistema de coordenadas de Turbo Vision	789
14.26. Campo con formato bitmap "Options"	790
14.27. Desktop con dos subvistas visores de texto	800
14.28. Arbol de vistas	801
14.29. Vista lateral de un visor de texto	802
14.30. Vista lateral del desktop	803

14.31. Modos de vídeo	805
14.32. Mapeo de bits del campo TEvent.What	811
14.33. Encaminamiento de eventos	812
14.34. Encaminamiento de eventos ofPreProcess	816
14.35. Cuadro de diálogo con comunicación entre vistas	829
14.35. Modificación del fondo del desktop	836
14.36. Construcción de menús y submenús	841
14.37. Menús y listas de elementos de menús.	842
14.38. Menú de gestión de ventanas	844
14.39. Claves de estado en diferentes contextos de ayuda	845
14.40. Ejemplos de ventanas	847
14.41. Cuadros de diálogo.	852
14.42. Validación de una fecha	860
14.43. Ayuda sensible al contexto	874
14.44. Ventana de ayuda sensible al contexto	881
15.1 Interfaz gráfico de usuario de Windows	888
15.2 Elementos de una ventana Windows	890
15.3 Ejecución del programa del ejemplo 15.1	892
15.4 Definición de las propiedades de un programa Windows	893
15.5 Ejecución del programa del ejemplo 15.2	895
15.6 Esquema de la jerarquía ObjectWindows	901
15.7 Ejecución del programa del ejemplo 15.4	903
15.8 Esquema general de funcionamiento del ejemplo 15.4	904
15.9 Ejecución del ejemplo 15.5	905
15.10 Ejecución del ejemplo 15.6	907
15.11 Ejecución del ejemplo 15.7	911
15.12 Ejecución del ejemplo 15.8	912
15.13 Ejecución del ejemplo 15.9	913
15.13 Ejecución del ejemplo 15.10	915
15.15 Ejecución del ejemplo 15.11	917
15.16 Ejecución del ejemplo 15.12	919
15.17 Ejecución del ejemplo 15.13	922
15.18 Taller de recursos del ejemplo 15.13	923
15.19 Ejecución del ejemplo 15.14	926
15.20 Ejecución del ejemplo 15.15	931
15.21 Ejecución del ejemplo 15.16	934
15.22 Diseño de un menú de editor con el taller de recursos	947
15.23 Taller de recursos: teclas aceleradoras	949
15.24 Taller de recursos: mapas de bits	950
15.25 Taller de recursos: cursor	951
15.26 Taller de recursos: icono	952
15.27 Taller de recursos: fuentes	954
15.28 Taller de recursos: cuadros de diálogo	955
15.29 Taller de recursos: tablas de cadenas	957
15.30 Ejecución del ejemplo 15.17	974
15.31 Ejecución del ejemplo 15.18	979
VI.1. Mapa de memoria del D.O.S.	1104
VI.2. Mapa de memoria de un programa generado con Turbo C	1104
VI.3. Ejecución de un programa residente desde un editor DOS	1108
VI.4. TSR desde un programa de comunicaciones	1109
VII.1. Gestión de memoria extendida	1121

CUADRO DE TABLAS

1.1 Rango de enteros sin signo de 2 bytes	4
1.2 Tipos real en Turbo Pascal 7.	5
1.3. Relación entre nº de colores y nº de bits por pixel	16
3.1 Palabras reservadas del lenguaje Pascal	87
4.1 Proc. y funciones de Turbo Pascal para entrada/salida	112
4.2 Caracteres especiales de control de la unit Crt	113
5.1 Operadores aritméticos con operandos enteros	121
5.2 Funciones internas de tipo entero	123
5.3 Tabla de verdad de los operadores AND, OR y NOT	124
5.4 Operadores de relación	128
5.5 Funciones de tipo carácter	132
5.6 Operadores aritméticos con operandos reales	140
5.7 Funciones estándar	144
5.8 Tipos enteros predefinidos en Turbo Pascal	152
5.9 Tipos reales predefinidos en Turbo Pascla	152
7.1 Relación nº de colores y nº bits por pixel	272
7.2 Macros de dispositivos gráficos	274
7.3 Macros de modos gráficos	274
8.1 Parámetros de cuadraturas de Gauss	345
8.2 Subprogramas para manejo de strings en Turbo Pascal	351
9.1 Operaciones con conjuntos	394
9.2 Operaciones lógicas con conjuntos	395
10.1 Interrupciones principales	434
10.2 Ejemplo de uso de una tabla. Ejercicio 10.3.	444
10.3 Ordenación de la tabla 10.2. Ejercicio 10.3	445
12.1 Operadores lógicos de bits	573
12.2 Tabla de verdad del operador NOT bit a bit	573
12.3 Tabla de verdad del operador AND bit a bit	573
12.2 Tabla de verdad del operador OR bit a bit	573
12.5 Tabla de verdad del operador XOR bit a bit	574
13.1. Cambio de notación entre POO y Turbo Pascal	654
14.1. Herencia de TWindow	764
14.2. Campos de TWindow	765
14.3. Operaciones con campos bitmap	791
14.4. Métodos para el cambio de flags de estado de una vista	796
14.5. Rangos de comandos de Turbo Vision	818
14.6. Tamaño de los registros de transferencia de datos	855
14.7. Plantilla de un validador por patrón	860
15.1 Diferencias entre las units Crt y WinCrt	891
15.2 Diferencias y similitudes entre las units Dos y WinDos	894
15.3 Funciones de la unit WinPrn	896
15.4 Tipos de datos de Windows	898
15.5 Units de ObjectWindows y funciones API de Windows 3.0	899
15.6 Units de las funciones API de Windows 3.1	900
15.7 Prefijos de la notación húngara	964
15.8 Tipos y rangos de mensajes	972
16.1. Comparación de programación estructurada y POO	1008
VI.1. Código de finalización de un programa	1106