CUPAS

Curso de Pascal

Por Nacho Cabanes

Versión 4.00 preliminar,

junio 2010

Este texto se puede distribuir libremente,
siempre y cuando no se modifique

Última versión en www.nachocabanes.com

Contenido.

Contenido. 2
Curso de Pascal por Nacho Cabanes. Tema 0: Introducción. 6
Curso de Pascal por Nacho Cabanes. Tema 1: Generalidades del Pascal. 8
Curso de Pascal por Nacho Cabanes. Tema 2: Introducción a las variables. 14
Tema 2.2: Tipos básicos de datos. 16
Tema 2.3: With. 19
Curso de Pascal. Tema 2: Ejercicios de ejemplo. 20
Tema 3: Entrada/salida básica. 22
Tema 3.2: Anchura de presentación en los números. 23

¡Error! Marcador no definido.
Tema 3.3: Comentarios. 24
Tema 3.4: Leer datos del usuario. 26
Tema 4: Operaciones matemáticas. 27
Tema 4.2: Concatenar cadenas. 29
Tema 4.3: Operadores lógicos. 30
Tema 4.4: Operaciones entre bits. 30
Curso de Pascal. Tema 4.5: Precedencia de los operadores. 33
Tema 5: Condiciones. 34
Tema 5.2: Condiciones y variables boolean. 35
Tema 5.3: Condiciones y sentencias compuestas. 35
Tema 5.4: Si no se cumple la condición. 36
Tema 5.5: Sentencias "If" encadenadas. 37
Tema 5.6: Varias condiciones simultáneas. 38
Curso de Pascal. Tema 6: Bucles. 40
Tema 6.2: "For" encadenados. 41
Tema 6.3: "For" y sentencias compuestas. 42
Tema 6.4: Contar sin números. 43
Tema 6.4 (b): Ejercicios sobre "For". 44
Tema 6.5: "While". 44
Tema 6.6: "Repeat". 46
Curso de Pascal. Tema 6.7: Ejercicios sobre "While" y "Repeat". 48
Curso de Pascal. Tema 6.8: Ejemplo - Adivinar números. 48
Tema 7: Constantes y tipos. 49
Tema 7.2: Constantes "con tipo". 51
Tema 7.3: Definición de tipos. 52
Tema 8: Procedimientos y funciones. 55
Tema 8.1: Procedimientos. 55
Tema 8.2: Funciones. 57
Tema 8.3: Uso de las funciones. 58
Tema 8.4: Procedimientos con parámetros. 59

Tema 8.5: Modificación de parámetros. 60
Tema 8.6: Parámetros por referencia. 62
Tema 8.7: Parámetros con el mismo nombre que las variables locales. 63
Tema 8.8: Recursividad. 64
Tema 8.9: La sentencia "forward". 66
Curso de Pascal. Tema 8.10. Funciones matemáticas: abs, sin, cos, arctan, round, trunc, sqr, sqrt, exp, ln, odd, potencias. 67
Curso de Pascal. Tema 8.11. Cadenas de texto. 68

Tema 9: Otros tipos de datos. 76
Tema 9.2: Otros tipos de datos (2). 80
Tema 9.3: Otros tipos de datos (3). 81

Curso de Pascal. Tema 10: Pantalla en modo texto. 83
Tema 10.2: Pantalla en modo texto con Surpas. 86
Tema 10.3: Procedimientos y funciones en CRT. 88
Tema 10.4: Ejemplo: juego del ahorcado. 90
Tema 10.5: Ejemplo: entrada mejorada. 97

Curso de Pascal. Tema 11a: Manejo de ficheros. 100
Tema 11.1. Manejo de ficheros (1) - leer un fichero de texto. 100

Curso de Pascal. Tema 11: Manejo de ficheros. 102
Tema 11.1. Manejo de ficheros (2) - Escribir en un fichero de texto. 102

Curso de Pascal. Tema 11: Manejo de ficheros. 104
Tema 11.3. Manejo de ficheros (3) - Ficheros con tipo. 104

Curso de Pascal. Tema 11: Manejo de ficheros. 108
Tema 11.4. Manejo de ficheros (4) - Ficheros generales. 108
Aplicación a un fichero GIF. 110
Abrir exclusivamente para lectura. 111

Curso de Pascal. Tema 12: Creación de unidades. 112

Tema 13: Variables dinámicas. 118
Tema 13.2: Variables dinámicas (2). 121
Tema 13.3: Variables dinámicas (3). 126
Tema 13.4: Variables dinámicas (4). 131
Tema 13.5: Ejercicios. 137

Curso de Pascal. Tema 14: Creación de gráficos. 137

14.2: Nuestro primer gráfico con Turbo Pascal. 141

14.3: Más órdenes gráficas. 143

14.4: Un sencillo programa de dibujo. 147

14.5: Mejorando el programa de dibujo. 148

14.6: Un efecto vistoso utilizando círculos. 149

14.7. Jugando con la paleta de colores. 150

14.8: Dibujando líneas. 152

14.9. Otro efecto más vistoso. 153

14.10. Gráficos con FPK y TMT. 155
14.11. Incluir los BGI en el EXE. 157
Curso de Pascal. Tema 15: Servicios del DOS. 159
15.1: Espacio libre en una unidad de disco. 159
15.2. Fecha del sistema. 160
15.3. Fecha usando interrupciones. 160
15.4. Lista de ficheros. 161
15.5. Ejecutar otros programas. 162
15.6: Ejecutar órdenes del DOS. 163
15.6: Principales posibilidades de la unidad DOS. 165
Curso de Pascal. Tema 16: Programación Orientada a Objetos. 168
16.2: Herencia y polimorfismo. 172
16.3: Problemas con la herencia. 176
Curso de Pascal. Tema 16.4: Más detalles... 179
Curso de Pascal. Tema 16.5: Métodos virtuales. 181
Curso de Pascal. Tema 16.6: Programación Orientada a Objetos (6). 184
Curso de Pascal. Tema 16.7: Programación Orientada a Objetos (7). 187
Curso de Pascal. Tema 16.8: Programación Orientada a Objetos (8). 188
Curso de Pascal. Tema 17: El entorno Turbo Vision. 191
Curso de Pascal. Tema 17.2: Turbo Vision - Ventanas estándar. 200
Curso de Pascal. Tema 17.3: Turbo Vision - Ventanas de diálogo. 211
Curso de Pascal. Tema 17.4: Turbo Vision - Ventanas de texto. 221
Curso de Pascal. Tema 17.5: Turbo Vision - Ejemplo. 229
Curso de Pascal. Ampliación 1: Otras órdenes no vistas. 237
Bucles: break, continue. 238
Goto. 238
Punteros: getmem, freemem, pointer. 239
Fin del programa: exit, halt. 240
Números aleatorios: random, randomize. 241
Inc y Dec. 242
Acceso a la impresora. 242
Curso de Pascal. Ampliación 2: Gráficos sin BGI. 243
1. A través de la Bios. 243
A través de la memoria de pantalla. 247
Líneas y otras figuras simples 250
Borrar la pantalla 260
La paleta de colores 261
Sin parpadeos.. 263
Escribir texto 264
Curso de Pascal. Ampliación 3 - Ordenaciones 266
Burbuja 266
MergeSort 268

QuickSort 269
Búsqueda binaria. 271
Curso de Pascal. Ampliación 4. Overlays. 274
Curso de Pascal. Ampliación 5. Ensamblador desde Turbo Pascal. 280
Curso de Pascal. Ampliación 6. Directivas del compilador (Turbo Pascal). 290
Curso de Pascal. Fuentes de ejemplo - Rotaciones 3D. 296

Curso de Pascal por Nacho Cabanes. Tema 0: Introducción.

Hay distintos **lenguajes** que nos permiten dar instrucciones a un ordenador (un **programa** de ordenador es básicamente eso: un conjunto de órdenes para un ordenador).

El lenguaje más directo es el propio del ordenador, llamado "lenguaje de máquina" o "**código máquina**", formado por secuencias de ceros y unos. Este lenguaje es muy poco intuitivo para nosotros, y difícil de usar. Por ello se recurre a otros lenguajes más avanzados, más cercanos al propio lenguaje humano (**lenguajes de alto nivel**), y es entonces el mismo ordenador el que se encarga de convertirlo a algo que pueda manejar directamente.

Se puede distinguir dos **tipos** de lenguajes, según se realice esta conversión:

- En los **intérpretes**, cada instrucción que contiene el programa se va convirtiendo a código máquina antes de ejecutarla, lo que hace que sean más lentos (a cambio, los intérpretes suelen ser más fáciles de crear, lo que permite que sean baratos y que puedan funcionar en ordenadores con menor potencia).
- En los **compiladores**, se convierte todo el programa en bloque a código máquina y después se ejecuta. Así, hay que esperar más que en un intérprete para comenzar a ver trabajar el programa, pero después éste funciona mucho más rápido (a cambio, los compiladores son más caros y suelen requerir ordenadores más potentes).

Hay lenguajes para los que sólo hay disponibles intérpretes, otros para los que sólo existen compiladores, y otros en los que se puede elegir entre ambos. La mayoría de los lenguajes **actuales** son compilados, y el entorno de desarrollo suele incluir:

- Un **editor** para escribir o revisar los programas.
- El **compilador** propiamente dicho, que los convierte a código máquina.
- **Otros** módulos auxiliares, como enlazadores (linkers) para unir distintos subprogramas, y depuradores (debuggers) para ayudar a descubrir errores.

Es cada vez más frecuente que todos estos pasos se puedan dar desde un único “entorno integrado”. Por ejemplo, el entorno de Turbo Pascal 7 tiene la siguiente apariencia:

Algunos de los lenguajes más difundidos son:

- **BASIC**, que durante mucho tiempo se ha considerado un buen lenguaje para comenzar a aprender, por su sencillez, aunque se podía tender a crear programas poco legibles. A pesar de esta "sencillez" hay versiones muy potentes, incluso para programar en entornos gráficos como Windows.
- **COBOL**, que fue muy utilizado para negocios (para crear software de gestión, que tuviese que manipular grandes cantidades de datos), aunque últimamente está bastante en desuso.
- **FORTRAN**, concebido para ingeniería, operaciones matemáticas, etc. También va quedando desplazado.
- **Ensamblador**, muy cercano al código máquina (es un lenguaje de "bajo nivel"), pero sustituye las secuencias de ceros y unos (bits) por palabras más fáciles de recordar, como MOV, ADD, CALL o JMP.
- **C**, uno de los mejor considerados actualmente (junto con C++ y Java, que mencionaremos a continuación), porque no es demasiado difícil de aprender y permite un grado de control del ordenador muy alto, combinando características de lenguajes de alto y bajo nivel. Además, es muy transportable: existe un estándar, el ANSI C, lo que asegura que se pueden convertir programas en C de un ordenador a otro o de un sistema operativo a otro con bastante menos esfuerzo que en otros lenguajes.
- **C++**, un lenguaje desarrollado a partir de C, que permite Programación Orientada a Objetos, por lo que resulta más adecuado para proyectos de una cierta envergadura.
- **Java**, desarrollado a su vez a partir de C++, que elimina algunos de sus inconvenientes, y ha alcanzado una gran difusión gracias a su empleo en Internet.
- **PASCAL**, el lenguaje estructurado por excelencia (ya se irá viendo qué es esto más adelante), y que en algunas versiones tiene una potencia comparable a la del lenguaje C, como es el caso de Turbo Pascal en programación para DOS y de Delphi en la programación para Windows. Frente al C tiene el inconveniente de que es menos portable, y la

ventaja de que en el caso concreto de la programación para DOS, Turbo Pascal no tiene nada que envidiar la mayoría de versiones del lenguaje C en cuanto a potencia, y además resulta más fácil de aprender, es muy rápido, crea ficheros EXE más pequeños, etc., mientras que en la programación para Windows, Delphi es una muy buena herramienta para crear aplicaciones de calidad en un tiempo razonablemente breve.

Dos conceptos que se mencionan mucho al hablar de programación son "programación estructurada" y "programación orientada a objetos".

La **programación estructurada** consiste en dotar al programa de un cierto orden, dividiéndolo en bloques independientes unos de otros, que se encargan de cada una de las tareas necesarias. Esto hace un programa más fácil de leer y modificar.

La **programación orientada a objetos** se tratará más adelante, cuando ya se tenga una buena base de programación "convencional". Como simple comentario, para que vayan sonando las cosas a conocidas, diré que "**Objects Pascal**" es el nombre que se suele dar a un lenguaje Pascal que permita programación orientada a objetos (como es el caso de Turbo Pascal), y que "**C++**" es una ampliación del lenguaje C, que también soporta P.O.O.

En lo que sigue vamos a ver los **fundamentos** de la programación en Pascal, primero intentando ceñirnos al Pascal estándar, y luego ampliando con las mejoras que incluye Turbo Pascal, la versión más difundida.

Los primeros programas, los más sencillos, se podrán compilar con cualquier versión del lenguaje Pascal (o casi), aunque la mayoría de los más avanzados (los que incluyan manejo de gráficos, por ejemplo) estarán creados pensando en el que posiblemente sea en estos momentos el compilador de Pascal más usado: Free Pascal, o bien en el clásico Turbo Pascal 7.0 para Dos.

Curso de Pascal por Nacho Cabanes. Tema 1: Generalidades del Pascal.

Vamos a empezar "al revés" de lo habitual: veremos un **ejemplo** de programa básico en Pascal (que se limitará a escribir la palabra "Hola" en la pantalla) y comentaremos cada una de las cosas que aparecen en él:

```
program Saludo;
begin
  write('Hola');
end.
```

Lo primero que llama la atención es que casi todas las palabras están escritas en inglés. Esto será lo habitual: la gran mayoría de las **palabras clave** de Pascal (palabras con un significado especial dentro del lenguaje) son palabras en inglés o abreviaturas de éstas.

Los colores que se ven en este programa NO aparecerán en cualquier editor que utilicemos. Sólo los editores más modernos usarán distintos colores para ayudarnos a descubrir errores y a identificar mejor cada orden cuando tecleamos nuestros programas.

En el lenguaje Pascal no existe distinción entre **mayúsculas y minúsculas**, por lo que "BEGIN" haría el mismo efecto que "begin" o "Begin". Así, lo mejor será adoptar el convenio que a cada uno le resulte más legible: algunos autores emplean las órdenes en mayúsculas y el resto en minúsculas,otros todo en minúsculas, otros todo en minúsculas salvo las iniciales de cada palabra... Yo emplearé normalmente minúsculas, y a veces mayúsculas y minúsculas combinadas cuando esto haga más legible algún comando "más enrevesado de lo habitual" (por ejemplo, si están formados por dos o más palabras inglesas como OutText o SetFillStyle.)

Si sabemos un poco de inglés, podríamos traducir literalmente el programa anterior, y así podremos darnos cuenta de lo que hace (aunque en un instante lo veremos con más detalle):

programa saludo
comienzo
escribir 'Hola'
final

Otra cosa que puede extrañar es eso de que algunas líneas terminen con un punto y coma. Pues bien, cada sentencia (u orden) de Pascal debe terminar con **un punto y coma** (;), salvo el último "end", que lo hará con un punto.

También hay otras tres excepciones: no es necesario un punto y coma después de un "begin", ni antes de una palabra "end" o de un "until" (se verá la función de esta palabra clave más adelante), aunque no es mala técnica terminar siempre cada sentencia con un punto y coma, al menos hasta que se tenga bastante soltura.

Cuando definamos variables, tipos, constantes, etc., veremos que tampoco se usa punto y coma después de las cabeceras de las declaraciones. Pero eso ya llegará...

Pues ahora ya sí que vamos a ver con **un poco más de detalle** lo que hace este programa.

Comienza con la palabra **program**. Esta palabra no es necesaria en muchos compiladores, como Turbo Pascal o Surpas, pero sí lo era incialmente en Pascal estándar, y el formato era

```
program NombrePrograma (input, output);
```

(entre paréntesis se escribía "input, output" para indicar que el programa iba a manejar los dispositivos de entrada y salida). Por ejemplo, como este programa escribe en la pantalla, si alguien usa una de las primeras versiones del Pascal de GNU, o algún otro compilador que siga estrictamente el Pascal estándar, deberá poner:

```
program Saludo(output);
```

Aunque para nosotros no sea necesario, su empleo puede resultar cómodo si se quiere poder recordar el objetivo del programa con sólo un vistazo rápido a su cabecera.

En algunos compiladores, puede que "nos regañe" si la palabra que sigue a "program" es distinta del nombre que tiene el fichero (es el caso de las primeras versiones de Tmt Pascal Lite), pero normalmente el programa funcionará a pesar de ello.

En nuestro caso, a nuestro programa lo hemos llamado "Saludo". La palabra "Saludo" es un **identificador**. Los "identificadores" son palabras que usaremos para referirnos a una variable, una constante, el nombre de una función o de un procedimiento, etc. Ya iremos viendo todos estos conceptos, pero sí vamos a anticipar un poco uno de ellos: una **variable** equivale a la clásica incógnita "x" que todos hemos usado en matemáticas (eso espero), que puede tomar cualquier valor. Ahora nuestras "incógnitas" podrán tener cualquier valor (no sólo un número: también podremos guardar textos, fichas sobre personas o libros, etc) y podrán tener nombres más largos (y que expliquen mejor su contenido, no hará falta limitarnos a una única letra, como en el caso de "x").

Estos nombres de "identificadores" serán combinaciones de letras (sin acentos) y números, junto con algunos (pocos) símbolos especiales, como el de subrayado (_). No podrán empezar con un número, sino por un carácter alfabético (A a Z, sin Ñ ni acentos) o un subrayado, y no podrán contener espacios.

Así, serían identificadores correctos: Nombre_De_Programa, programa2, _SegundoPrograma pero no serían admisibles 2programa, 2ºprog, tal&tal, Prueba de programa, ProgramaParaMí (unos por empezar por números, otros por tener caracteres no aceptados, y otros por las dos cosas).

Las palabras "**begin**" y "**end**" marcan el principio y el final del programa, que esta vez sólo se compone de una línea. Nótese que, como se dijo, el último "end" debe terminar con un **punto**.

"**Write**" es la orden que permite escribir un texto en pantalla. El conjunto de todo lo que se desee escribir se indica entre paréntesis. Cuando se trata de un texto que queremos que aparezca "tal cual", éste se encierra entre comillas (una comilla simple para el principio y otra para el final, como aparece en el ejemplo).

El **punto y coma** que sigue a la orden "write" no es necesario (va justo antes de un "end"), pero tampoco es un error, y puede ser cómodo, porque si después añadimos otra orden entre "write" y "end", sería dicha orden la que no necesitaría el punto y coma (estaría justo antes de "end"), pero sí que pasaría a requerirlo el "write". Se entiende, ¿verdad? ;)

Para que no quede duda, probad a hacerlo: escribid ese "write" sin punto y coma al final, y vereis que no hay problema. En cambio, si ahora añadís otro "write" después, el compilador sí que protesta.

La orden "write" aparece algo más a la derecha que el resto. Esto se llama **escritura indentada**, y consiste en escribir a la misma altura todos los comandos que se encuentran a un mismo nivel, algo más a la derecha los que están en un nivel inferior, y así sucesivamente, buscando mayor legibilidad. Se irá viendo con más detalle a medida que se avanza.

En un programa en Pascal no hay necesidad de conservar una **estructura** tal que aparezca cada orden en una línea distinta. Se suele hacer así por claridad, pero realmente son los puntos y coma (cuando son necesarios) lo que indica el final de una orden, por lo que el programa anterior se podría haber escrito:

```
program Saludo; begin write('Hola'); end.
```

o bien

```
program Saludo;
  begin              write('Hola');
     end.
```

lo que desde luego no se puede hacer es "partir palabras": si escribimos
```
pro gram Saludo;
```

el ordenador creerá que "pro" y "gram" son dos órdenes distintas, y nos dirá que no sabe qué es eso.

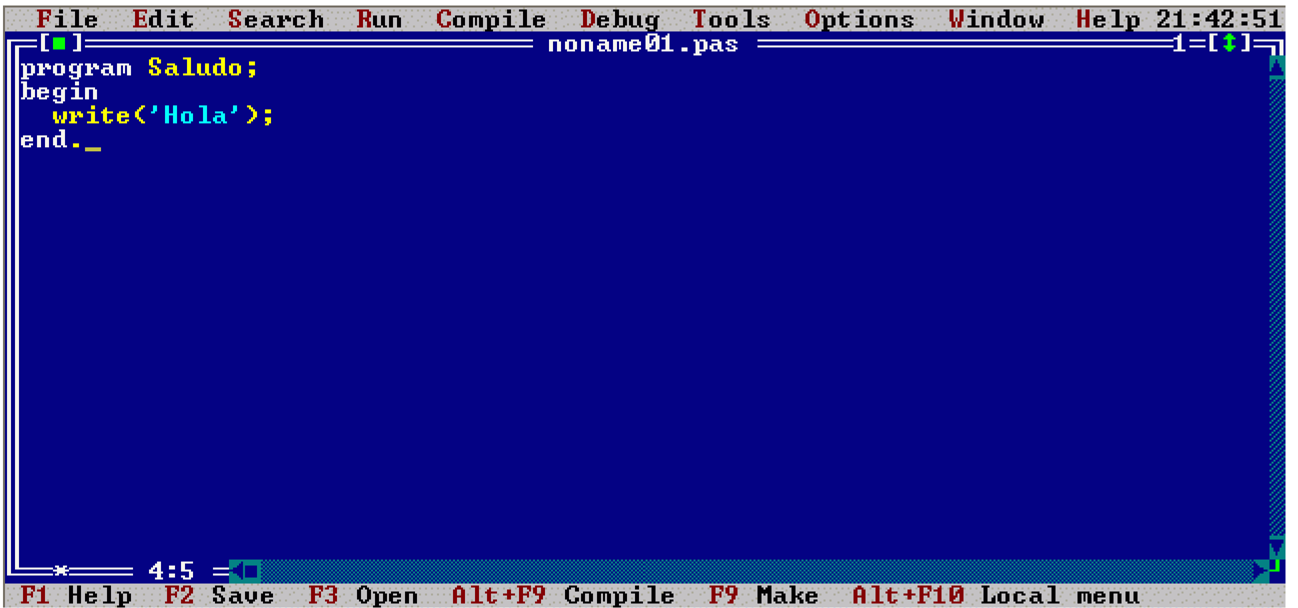
Los detalles concretos sobre **cómo probar este programa** dependerán del compilador que se esté utilizando. Unos tendrán un Entorno Integrado, desde el que escribir los programas y probarlos (como Free Pascal, Turbo Pascal y Surpas), mientras que en otros hará falta un editor para teclear los programas y el compilador para probarlos (como Tmt Lite) y otros no incluyen editor en la distribución normal, pero es fácil conseguir uno adaptado para ellos (es el caso de FPK y de Gnu Pascal). (En un apéndice de este curso tendrás enlaces a sitios donde descargar todos esos compiladores, así como instrucciones para instalarlos).

Nosotros usaremos **Free Pascal** en la mayoría de los ejemplos de este curso, porque es un compilador gratuito, que sigue bastante el estándar que marcó Turbo Pascal, que funciona bien en ordenadores modernos (Turbo Pascal da problemas en algunos) y que está disponible para distintos sistemas operativos: MsDos, Windows, Linux, etc.

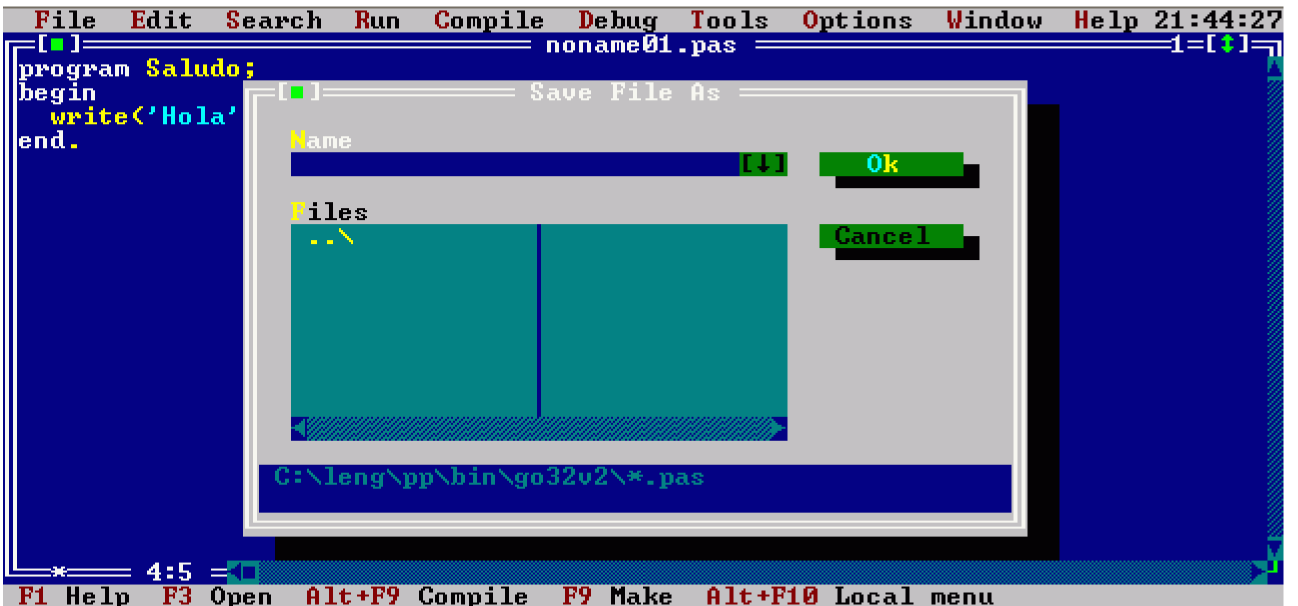
Tienes instrucciones sobre cómo [instalar Free Pascal](), pero en este momento vamos a suponer que ya está instalado y a ver cómo se teclearía y se probaría este programa bajo **Windows:**

Al entrar al compilador, nos aparece la pantalla de "bienvenida":

En el menú "File", usamos la opción "New" para comenzar a teclear un nuevo programa



Una vez que esté tecleado, escogemos la opción "Run" del menú "Run" (o pulsamos Ctrl+F9) para probarlo. Si todavía no hemos guardado el fichero, nos pedirá en primer lugar que lo hagamos.



Y ya está listo. Eso sí, posiblemente tendremos un "problema": nuestro programa realmente escribe Hola en nuestra pantalla, pero lo hace muy rápido y vuelve a la pantalla de edición del programa, así que no tenemos tiempo de leerlo. Podemos ver el resultado (la "pantalla de usuario") pulsando las teclas Alt+F5 (o la opción "User Screen" del menú "Debug"). Veremos algo como esto:

```
Running "c:\leng\pp\bin\go32v2\ej01.exe "
Hola
```

Si estamos bajo **Linux**, usaremos cualquiera de los editores del sistema para teclear nuestro programa, lo guardaremos con el nombre "ejemplo.pas", y después teclearemos

```
fpc ejemplo.pas
```

Si no hay ningún error, obtendremos un programa "ejecutable", capaz de funcionar por sí solo, y que podremos probar tecleando

```
./ejemplo
```

Ejercicio propuesto: *Crea un programa similar al primer ejemplo que hemos visto, pero que, en vez de escribir Hola, escriba tu nombre.*

Curso de Pascal por Nacho Cabanes. Tema 2: Introducción a las variables.

Las **variables** son algo que no contiene un valor predeterminado, una posición de memoria a la que nosotros asignamos un nombre y en la que podremos almacenar datos.

En el primer ejemplo que vimos, puede que no nos interese escribir siempre el mensaje "Hola", sino uno más **personalizado** según quien ejecute el programa. Podríamos preguntar su nombre al usuario, guardarlo en una variable y después escribirlo a continuación de la palabra "Hola", con lo que el programa quedaría

```
program Saludo2;

var
  nombre: string[20];

begin
  writeln('Introduce tu nombre, por favor');
  readln(nombre);
  write('Hola ',nombre);
end.
```

Aquí ya aparecen más conceptos nuevos. En primer lugar, hemos **definido una variable**, para lo que empleamos la palabra **var**, seguida del nombre que vamos a dar a la variable, y del tipo de datos que va a almacenar esa variable.

Los nombres de las variables siguen las reglas que ya habíamos mencionado para los identificadores en general, y no se indica ningún punto y coma entre la palabra "var" y el nombre de la variable (o variables que se declaran)..

Con la palabra **string** decimos que la variable nombre va a contener una cadena de caracteres (letras o números). Un poco más adelante, en esta misma lección, comentamos los principales tipos de datos que vamos a manejar. En concreto, string[20] indica que el nombre podrá estar formado hasta por 20 letras o números

Nota: en la variante del lenguaje Pascal conocida como "Extended Pascal", la longitud máxima de una cadena de texto se indica con paréntesis, de modo que si algún compilador protesta con "string[20]", habrá que probar con "string(20)".

Pasemos al cuerpo del programa. En él comenzamos escribiendo un mensaje de aviso. Esta vez se ha empleado **writeln**, que es exactamente igual que write con la única diferencia de que después de visualizar el mensaje, el cursor (la posición en la que se seguiría escribiendo, marcada normalmente por una rayita o un cuadrado que parpadea) pasa a la línea siguiente, en vez de quedarse justo después del mensaje escrito.

Después se espera a que el usuario introduzca su nombre, que le asignamos a la variable "nombre", es decir, lo guardamos en una posición de memoria cualquiera, que el compilador ha reservado para nosotros, y que nosotros no necesitamos conocer (no nos hace falta saber que está en la posición 7245 de la memoria, por ejemplo) porque siempre nos referiremos a ella llamándola "nombre". De todo esto se encarga la orden **readln**.

Si queremos **dar un valor** a la variable nosotros mismos, desde el programa, usaremos la expresión **:=** (un símbolos de "dos puntos" y otro de "igual", seguidos, sin espacios entre medias), así:

```
Edad := 17;
```

Finalmente, aparece en pantalla la palabra "Hola" seguida por el nombre que se ha introducido. Como se ve en el ejemplo, "writeln" puede escribir **varios datos**, si los separamos entre comas, pero eso lo estudiaremos con detalle un poco más adelante...

Sólo un comentario antes de seguir: lo que escribimos entre comillas en una orden "write" aparecerá tal cual en pantalla; lo que escribimos sin pantalla, es para que el ordenador intente adivinar su valor. Así, se nos pueden dar casos como estos:

```
write ( 'Hola' ) Escribe Hola
write ( nombre ) Escribe el valor de la variable nombre
write ( '3+3' ) Escribe el texto 3+3
write ( 3+3 ) Escribe el resultado de la operación 3+3 (6)
```

Está claro que + es el símbolo que usaremos para sumar. Para restar emplearemos -, para multiplicar * y para dividir /. Más adelante veremos más detalles sobre las operaciones aritméticas.

Tema 2.2: Tipos básicos de datos.

En Pascal debemos **declarar** las variables que vamos a usar, avisar a nuestro compilador para que les reserve espacio. Esto puede parecer incómodo para quien ya haya trabajado en lenguaje Basic, pero en la práctica ayuda a conseguir programas más legibles y más fáciles de corregir o ampliar. Además, evita los errores que puedan surgir al emplear variables incorrectas: si queremos usar "nombre" pero escribimos "nombe", la mayoría de las versiones del lenguaje Basic no indicarían un error, sino que considerarían que se trata de una variable nueva, que no tendría ningún valor, y normalmente se le asignaría un valor de 0 o de un texto vacío.

En Pascal disponemos de una serie de **tipos predefinidos**, y de otros que podemos crear nosotros para ampliar el lenguaje. Los primeros tipos que veremos son los siguientes:

- **Integer**. Es un número entero (sin cifras decimales) con signo, que puede valer desde -32768 hasta 32767. Ocupa 2 bytes de memoria. (*Nota*: el espacio ocupado y los valores que puede almacenar son valores para Turbo Pascal, y pueden variar para otros compiladores).

Ejercicio propuesto: *Crea un programa que, en lugar de preguntarte tu nombre, te pregunte tu edad y luego la escriba en pantalla.*

- **Byte**. Es un número entero, que puede valer entre 0 y 255. El espacio que ocupa en memoria es el de 1 byte, como su propio nombre indica. (*Nota*: es un tipo de datos definido por Turbo Pascal, y puede no estar disponible en otros compiladores, como es el caso de GNU Pascal).

Ejercicio propuesto: *Crea un programa que te pregunte dos números del 1 al 10 y escriba su suma.*

- **Char**. Representa a un carácter (letra, número o símbolo). Ocupa 1 byte.

Ejercicio propuesto: *Crea un programa que te pregunte tres letras y luego las escriba en orden inverso.*

- **String**. Es una cadena de caracteres, empleado para almacenar y representar mensajes de más de una letra (hasta 255). Ocupa 256 bytes. El formato en Pascal estándar (y en Turbo Pascal, hasta la versión 3.01) era **string[n]** (o string(n), según casos, como ya se han comentado), donde n es la anchura máxima que queremos almacenar en esa cadena de caracteres (de 0 a 255), y entonces ocupará n+1 bytes en memoria. En las últimas versiones de Turbo Pascal (y otros) podemos usar el formato "string[n]" o simplemente "string", que equivale a "string[255]". En otros compiladores, como GNU Pascal, el tamaño permitido es mucho mayor (normalmente por encima de las 32.000 letras).

Ejercicio propuesto: *Crea un programa que te pida tres palabras y las muestre separadas por espacios y en el orden contrario a como las has introducido (primero la tercera palabra, después la segunda y finalmente la primera).*

- **Real**. Es un numero real (con decimales) con signo. Puede almacenar números con valores entre 2.9e-39 y 1.7e38 (*en notación científica, e5 equivale a multiplicar por 10 elevado a 5, es decir, podremos guardar números tan grandes como un 17 seguido de 37 ceros, o tan pequeños como 0,00...029 con 38 ceros detrás de la coma*). Tendremos 11 o 12 dígitos significativos y ocupan 6 bytes en memoria.

Ejercicio propuesto: *Crea un programa que te pida dos números reales y muestre en pantalla el resultado de multiplicarlos.*

- **Boolean**. Es una variable lógica, que puede valer **TRUE** (verdadero) o **FALSE** (falso), y se usa para comprobar condiciones.

Ejercicio propuesto: *Crea un programa que cree una variable de tipo boolean, le asigne el valor TRUE y luego muestre dicho valor en pantalla.*

- **Array** (nota: algunos autores traducen esta palabra como "**arreglo**"). Se utilizan para guardar una serie de elementos, todos los cuales son del mismo tipo. Se deberá indicar el índice inferior y superior (desde dónde y hasta dónde queremos contar), separados por dos puntos (..), así como el tipo de datos de esos elementos individuales. Por ejemplo, para guardar hasta 200 números enteros, usaríamos:

```
lista: array[1..200] of integer
```

Se suele emplear para definir **vectores o matrices**. Así, una matriz de dimensiones 3x2 que debiera contener números reales sería:

```
matriz1: array[1..3,1..2] of real
```

Para mostrar en pantalla el segundo elemento de la primera lista de números (o de un vector) se usaría

```
write( lista[2] );
```

y para ver el elemento (3,1) de la matriz,

```
writeln( matriz1[3,1] );
```

Veremos ejemplos más desarrollados de cómo se usan los Arrays cuando lleguemos al tema 6, en el que trataremos órdenes como "for", que nos permitirán recorrer todos sus elementos.

Ejercicio propuesto: *Crea un programa que reserve espacio para un Array de 3 números enteros, que asigne a sus elementos los valores 3, 5 y 8, y que después muestre en pantalla la suma de los valores de sus 3 elementos.*

- **Record**. La principal limitación de un array es que todos los datos que contiene deben ser del mismo tipo. Pero a veces nos interesa agrupar datos de distinta naturaleza, como pueden ser el nombre y la edad de una persona, que serían del tipo string y byte, respectivamente. Entonces empleamos los records o **registros**, que se definen indicando el nombre y el tipo de cada **campo** (cada dato que guardamos en el registro), y se accede a estos campos indicando el nombre de la variable y el del campo separados por un punto:

```
program Record1;
var
  dato: record
    nombre: string[20];
    edad: byte;
end;

begin
```

```
  dato.nombre:='José Ignacio';
  dato.edad:=23;
  write('El nombre es ', dato.nombre );
  write(' y la edad ', dato.edad, ' años.');
end.
```

La única novedad en la definición de la variable es la aparición de una palabra **end** después de los nombres de los campos, lo que indica que hemos terminado de enumerar éstos.

Ya dentro del cuerpo del programa, vemos la forma de acceder a estos campos, tanto para darles un valor como para imprimirlo, indicando el nombre de la variable a la que pertenecen, seguido por un punto. El conjunto**:=** es, como ya hemos dicho, la sentencia de **asignación** en Pascal, y quiere decir que la variable que aparece a su izquierda va a tomar el valor que está escrito a la derecha (por ejemplo, x := 2 daría el valor 2 a la variable x).

Ejercicio propuesto: *Crea un programa que reserve espacio para un Array de 3 números enteros, que asigne a sus elementos los valores 3, 5 y 8, y que después muestre en pantalla la suma de los valores de sus 3 elementos.*

Tema 2.3: With.

Puede parecer engorroso el hecho de escribir "dato." antes de cada campo. También hay una forma de solucionarlo: cuando vamos a realizar varias operaciones sobre los campos de un mismo registro (record), empleamos la orden **with**, con la que el programa anterior quedaría

```
program Record2;
var
  dato: record
    nombre: string[20];
    edad: byte;
end;

begin
  with dato do
    begin
      nombre:='José Ignacio';
      edad:=23;
      write('El nombre es ', nombre );
      write(' y la edad ', edad, ' años.');
    end;
end.
```

En este caso tenemos un nuevo bloque en el cuerpo del programa, delimitado por el "begin" y el "end" situados más a la derecha, y equivale a decir "en toda esta parte del programa me estoy refiriendo a la variable dato". Así, podemos nombrar los campos que queremos modificar o escribir, sin necesidad de repetir a qué variable pertenecen.

Nota: aquí vuelve a aparecer la **escritura indentada**: para conseguir una mayor legibilidad, escribimos un poco más a la derecha todo lo que depende de la orden "with". No es algo obligatorio, pero sí recomendable.

Y aun hay más sobre registros. Existe una posibilidad extra, conocida como "registros variantes", que veremos más adelante.

Estos tipos básicos de datos se pueden "**relacionar**" entre sí. Por ejemplo, podemos usar un registro (record) para guardar los datos de cada uno de nuestros amigos, y guardarlos todos juntos en un array de registros. Todo esto ya lo iremos viendo.

Por cierto, si alguien ve un cierto parecido entre un **string** y un **array**, tiene razón: un string no es más que un "array de chars". De hecho, la definición original de "string[x]" en Pascal estándar era algo así como "packed array [1..x] of char", donde la palabra **packed** indicaba al compilador que tratase de compactar los datos para que ocupasen menos.

Curso de Pascal. Tema 2: Ejercicios de ejemplo.

Ejemplo 1: Cambiar el valor de una variable.
Ejemplo 2: Sumar dos números enteros.
Ejemplo 3: Media de los elementos de un vector.

Como todavía llevamos pocos conocimientos acumulados, la cosa se queda aquí, pero con la siguiente lección ya podremos realizar operaciones matemáticas algo más serias, y comparaciones lógicas.

Cambiar el valor de una variable.

```
program NuevoValor;

var
  numero: integer;

begin
  numero := 25;
```

```
  writeln('La variable vale ', numero);
  numero := 50;
  writeln('Ahora vale ', numero);
  numero := numero + 10;
  writeln('Y ahora ', numero);
  writeln('Introduce ahora tú el valor');
  readln( numero );
  writeln('Finalmente, ahora vale ', numero);
end.
```

Este programa no debería tener ninguna dificultad; primero le damos un valor (25), luego otro (50), luego modificamos este valor (50+10=60) y finalmente dejamos que sea el usuario quien dé un valor.

Sumar dos números enteros.

```
program SumaDosNumeros;

var
  numero1, numero2, suma: integer;

  begin
  writeln('Introduce el primer número');
  readln( numero1 );
  writeln('Introduce el segundo número');
  readln( numero2 );
  suma := numero1 + numero2;
  writeln('La suma de los dos números es: ', suma);
end.
```

Fácil, ¿no? Pedimos dos números, guardamos en una variable su suma, y finalmente mostramos el valor de esa suma.

Media de los elementos de un vector.

Este es un programa nada optimizado, para que se adapte a los conocimientos que tenemos por ahora y se vea cómo se manejan los Arrays. Admite muchas mejoras, que iremos viendo más adelante.

Por si alguien no ha trabajado con vectores, me salto las explicaciones matemáticas serias: la idea es simplemente que vamos a hallar la media de una serie de números.

Como novedades sobre la lección, incluye la forma de dejar una línea de pantalla en blanco (con writeln), o de definir de una sola vez varias variables

que sean del mismo tipo, separadas por comas. Las operaciones matemáticas se verán con más detalle en la próxima lección.

```
program MediadelVector;

var
  vector: array [1..5] of real;
  suma, media: real;

begin
  writeln('Media de un vector con 5 elementos.');
  writeln;
  writeln('Introduce el primer elemento');
  readln(vector[1]);
  writeln('Introduce el segundo elemento');
  readln(vector[2]);
  writeln('Introduce el tercer elemento');
  readln(vector[3]);
  writeln('Introduce el cuarto elemento');
  readln(vector[4]);
  writeln('Introduce el quinto elemento');
  readln(vector[5]);
  suma := vector[1] + vector[2] + vector[3] + vector[4]
    + vector[5];
  media := suma / 5;
  writeln('La media de sus elementos es: ', media);
end.
```

Tema 3: Entrada/salida básica.

Ya hemos visto por encima las dos formas más habituales de mostrar datos en pantalla, con "write" o "writeln", y de aceptar la introducción de datos por parte del usuario, con "readln" (o "read", que no efectúa un retorno de carro después de leer los datos). Veamos ahora su manejo y algunas de sus posibilidades con más detalle:

Para mostrar datos, tanto en pantalla como en impresora, se emplean **write** y **writeln**. La **diferencia** entre ambos es que "write" deja el cursor en la misma línea, a continuación del texto escrito, mientras que "writeln" baja a la línea inferior. Ambas órdenes pueden escribir tipos casi de cualquier clase: cadenas de texto, números enteros o reales, etc. No podremos escribir directamente arrays, records, ni muchos de los datos definidos por el usuario.

Cuando se desee escribir varias cosas **en la misma línea**, todas ellas se indican entre un mismo paréntesis, y separadas por comas.

Un comentario para quien ya haya programado en **Basic**: en la mayoría de las versiones de este lenguaje si separamos varios datos mediante comas en una sentencia PRINT, se verán separados en pantalla por un cierto número de

espacios. En ese aspecto, la "," de Pascal recuerda más al ";" de Basic, ya que escribe los datos uno a continuación del otro. De hecho, si fueran números, ni siquiera aparecerían espacios entre ellos (también al contrario que en la mayoría de versiones de Basic):

```
WriteLn (1,10,345);
```

daría como resultado

110345

Se puede especificar la **anchura** de lo escrito, mediante el símbolo de dos puntos (:) y la cifra que indique la anchura. Si se trata de un número real y queremos indicar también el número de decimales, esto se hace también después de los dos puntos, con el formato ":anchura_total:decimales". Como ejemplos:

```
program Write1;

var
  nombre: string[40];
  edad: byte;
  resultado: real;

begin
  nombre := 'Pepe';
  edad := 18;
  resultado := 13.12;
  write ('Hola, ',nombre,' ¿qué tal estás? ');
  writeln (resultado:5:2);
  writeln('Hola,',nombre:10,'.  Tu edad es: ',edad:2);
end.
```

En el caso de una cadena de texto, la anchura que se indica es la que se tomará como mínima: si el texto es mayor no se "parte", pero si es menor, se rellena con espacios por la izquierda hasta completar la anchura deseada.

Tema 3.2: Anchura de presentación en los números.

Igual ocurre con los números: si es más grande que la anchura indicada, no se "parte", sino que se escribe completo. Si es menor, se rellena con espacios por la izquierda. Los decimales sí que se redondean al número de posiciones indicado:

```
program Write2;

var num: real;
begin
  num := 1234567.89;
  writeln(num);
    (* La línea anterior lo escribe con el formato por defecto:
     exponencial *)
  writeln(num:20:3);  (* Con tres decimales *)
  writeln(num:7:2);   (* Con dos decimales *)
  writeln(num:4:1);   (* Con un decimal *)
  writeln(num:3:0);   (* Sin decimales *)
  writeln(num:5);     (* ¿Qué hará ahora? *)
end.
```

La salida por pantalla de este programa sería:

```
1.2345678900E+06
1234567.890
1234567.89
1234567.9
1234568
1.2E+06
```

Aquí se puede observar lo que ocurre en los distintos casos:

- Si no indicamos formato, se usa notación científica (exponencial).
- Si la anchura es mayor, añade espacios por la izquierda.
- Si es menor, no se trunca el número.
- Si el número de decimales es mayor, se añaden ceros.
- Si éste es menor, se redondea.
- Si indicamos formato pero no decimales, sigue usando notación exponencial, pero lo más compacta que pueda, tratando de llegar al tamaño que le indicamos.

Tema 3.3: Comentarios.

En este programa ha aparecido también otra cosa nueva: **los comentarios**:

```
writeln(num:20:3);  (* Con tres decimales *)
```

Un comentario es algo el compilador va a ignorar, como si no hubiéramos escrito nada, y que nosotros incluimos dentro del programa para que nos resulte más legible o para aclarar lo que hace una línea o un conjunto de líneas.

En Pascal, los comentarios se encierran entre (* y *). También está permitido usar { y }, tanto en Turbo Pascal como en SURPAS. Como se ve en el ejemplo, los comentarios pueden ocupar más de una línea.

A partir de ahora, yo emplearé los comentarios para ayudar a que se entienda un poco más el desarrollo del programa, y también para incluir una cabecera al principio, que indique el cometido del programa y los compiladores con los que ha sido comprobado.

En la práctica, es **muy importante** que un programa esté bien documentado. Cuando se trabaja en grupo, la razón es evidente: a veces es la única forma de que los demás entiendan nuestro trabajo. En estos casos, el tener que dar explicaciones "de palabra" es contraproducente: Se pierde tiempo, las cosas se olvidan... Tampoco es cómodo distribuir las indicaciones en ficheros aparte, que se suelen extraviar en el momento más inoportuno. Lo ideal es que los comentarios aclaratorios estén siempre en el texto de nuestro programa.

Pero es que cuando trabajamos solos también es importante, porque si releemos un programa un mes después de haberlo escrito, lo habitual es que ya no nos acordemos de lo que hacía la variable X, de por qué la habíamos definido como "Record" y no como "Array", por qué dejábamos en blanco la primera ficha o por qué empezábamos a ordenar desde atrás.

Por cierto, que de ahora en adelante, como ya entendemos eso de los comentarios, los usaré para indicar las versiones en las que está comprobado cada uno de los programas, y para explicar un poco lo que hace.

(*Nota*: según el compilador que manejemos, es habitual que un comentario que empezamos con (* se deba terminar con *), y no con }. También es frecuente que no se puedan "**anidar**" comentarios, de modo que algo como `begin {{} end` es correcto, porque la segunda llave abierta se ignora en la mayoría de los compiladores, mientras que algunos exigirán que se cierre también: `begin {{}} end` Puede incluso que algún compilador nos permita escoger cual de las dos opciones preferimos.

Una observación: si queremos **escribir las comillas** (o algún símbolo extraño que no aparezca en el teclado) como parte de un texto, podemos hacerlo de varias formas. La forma habitual es conociendo su código ASCII y usando la función CHR. Por ejemplo, el símbolo ~ corresponde al código ASCII 126, de modo que usaríamos “write(chr(126))”. En Turbo Pascal existe también una notación alternativa, que es usar # en vez de CHR, de manera que podríamos escribir “write(#126)”. Y para el caso particular de las comillas, hay una posibilidad extra: si dentro de una orden write ponemos dos comillas seguidas, el compilador entenderá que queremos escribir una de ellas.

Vamos a verlo con un ejemplo:

Ejercicio propuesto: *Crea un programa que pida al usuario dos números reales y muestre en pantalla el resultado de su división, con dos cifras decimales.*

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Como escribir las     }
{      comillas            }
{    COMILLA.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - FPK Pascal 1.0      }
{--------------------------}

program comilla;

begin
  writeln(' writeln(' + chr(39) + 'Esto es un ejemplo' +chr(39) +')');
  writeln(' writeln(' + #39 + 'Esto es un ejemplo' + #39 +')');
  writeln(' writeln(''Esto es un ejemplo'')');
end.
```

Tema 3.4: Leer datos del usuario.

Para tomar **datos del usuario**, la forma más directa es empleando **readln**, que toma un texto o un número y asigna este valor a una variable. No avisa de lo que está haciendo, así que normalmente convendrá escribir antes en pantalla un mensaje que indique al usuario qué esperamos que teclee:

```
writeln('Por favor, introduzca su nombre');
readln(nombre);
```

"Readln" tiene algunos **inconvenientes**:

- No termina hasta que pulsemos RETURN.
- La edición es incómoda: para corregir un error sólo podemos borrar todo lo que habíamos escrito desde entonces, no podemos usar las flechas o INICIO/FIN para desplazarnos por el texto.
- Si queremos dar un valor a una variable numérica y pulsamos " 23" (un espacio delante del número) le dará un valor 0.
- ...

A pesar de estos inconvenientes, es la forma estándar de leer datos del teclado, así vamos a ver un ejemplo de cómo usarla:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Introducción de datos }
{    con ReadLn            }
{    READLN1.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program Readln1;

var
  nombre: string[40];
  edad: byte;

begin
  write ('Escribe tu nombre: ');
  readln(nombre);
  write ('Y ahora tu edad: ');
  readln(edad);
  write ('Hola, ',nombre,' ¿qué tal estás?');
  writeln('Hola,',nombre:10,'.  Tu edad es:',edad:2);
end.
```

Más adelante, veremos que existen formas mucho más versátiles y cómodas de leer datos a través del teclado, en el mismo tema en el que veamos cómo se maneja la pantalla en modo texto desde Pascal...

Ejercicio propuesto: *Crea un programa que pida al usuario su nombre y su apellido y luego los escriba al revés, separados por una coma. Por ejemplo si el usuario introduce Nacho como nombre y Cabanes como apellido, se escribirá Cabanes, Nacho.*

Tema 4: Operaciones matemáticas.

En Pascal contamos con una serie de operadores para realizar sumas, restas, multiplicaciones y otras operaciones no tan habituales. Algunos de ellos ya los habíamos comentado. Vamos a verlos ahora con más detalle. Son los siguientes:

Operador	Operación	Operandos	Resultado
+	Suma	enteros reales	entero real
-	Resta	enteros reales	entero real
*	Multiplicación	enteros reales	entero real
/	División	enteros reales	real
div	División entera	enteros	entero
mod	Resto	enteros	entero

En operaciones como +, - y * supongo que no habrá ninguna duda: si sumo dos números enteros obtengo un número entero, si resto dos reales obtengo un número real, y lo mismo pasa con la multiplicación. Los problemas pueden venir con casos como el de 10/3. Si 10 y 3 son números enteros, ¿qué ocurre con su división? En otros lenguajes como C, el resultado sería 3, la parte entera de la división. En Pascal no es así: el resultado sería 3.333333, un número real. Si queremos la parte entera de la división, deberemos utilizar **div**. Finalmente, **mod** nos indica cual es el resto de la división. El signo - se puede usar también para indicar**negación**. Allá van unos ejemplillos:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Prueba de operaciones }
{    elementales           }
{    OPERAC.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program operaciones;

var
  e1, e2: integer;       (* Numeros enteros *)
  r1, r2, r3: real;      (* Números reales *)

begin
  e1:=17;
  e2:=5;
  r1:=1;
  r2:=3.2;
  writeln('Empezamos...');
```

```
  r3:=r1+r2;
  writeln('La suma de r1 y r2 es :', r3);
  writeln('  o también ', r1+r2 :5:2);     (* Indicando el formato *)
  writeln('El producto de r1 y r2 es :', r1 * r2);
  writeln('El valor de r1 dividido entre r2 es :', r1 / r2);
  writeln('La diferencia de e2 y e1 es : ', e2 - e1);
  writeln('La división de e1 entre e2 : ', e1 / e2);
  writeln('  Su división entera : ', e1 div e2);
  writeln('  Y el resto de la división : ', e1 mod e2);
  writeln('El opuesto de e2 es :', -e2);
end.
```

Supongo que no habrá ninguna duda. O:-) De todos modos, experimentad. Y ojo con el formato de "mod", porque se parece bastante poco como se diría "el resto de dividir e1 entre e2" a la notación "e1 mod e2". Aun así, todo fácil, ¿verdad?

Ejercicio propuesto: *Crea un programa que pida al usuario dos números reales y muestre cuando es el resultado de su división (sólo la parte entera) y el resto de la división. Por ejemplo, si se indican los números 21 y 5, el programa debería responder que el resultado de la división es 4 y que el resto es 1*

Tema 4.2: Concatenar cadenas.

El operador + (suma) se puede utilizar también para **concatenar** cadenas de texto, así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Concatenar cadenas    }
{    con "+"               }
{    CONCAT.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

Program Concat;

var
  texto1, texto2, texto3: string;

begin
  texto1 := 'Hola ';
  texto2 := '¿Cómo estás?';
  texto3 := texto1 + texto2;
  writeln(texto3);           (* Escribirá "Hola ¿Cómo estás?" *)
```

```
end.
```

(Más adelante se dedica un apartado al manejo de cadenas de texto).

Cuando tratemos tipos de datos más avanzados, veremos que +, - y * también se pueden utilizar para **conjuntos**, e indicarán la unión, diferencia e intersección. (Esto lo veremos más adelante, en el tema 9).

Ejercicio propuesto: *Crea un programa que pida al usuario su nombre y su apellido, cree una cadena formada por el apellido, una coma y el nombre (por ejemplo si el usuario introduce Nacho como nombre y Cabanes como apellido, la cadena contendrá "Cabanes, Nacho"), y finalmente escriba esa cadena.*

Tema 4.3: Operadores lógicos.

Vimos de pasada en el tema 2 que había unos tipos de datos llamados "boolean", y que podían valer TRUE (verdadero) o FALSE (falso). En la próxima lección veremos cómo hacer comparaciones del estilo de "si A es mayor que B y B es mayor que C", y empezaremos a utilizar variables de este tipo, pero vamos a mencionar ya eso del "y".

Podremos encadenar proposiciones de ese tipo (si A y B entonces C) con: **and** (y), **or** (ó), **not** (no) y los **operadores relacionales**,que se usan para comparar y son los siguientes:

Operador	Operación
=	Igual a
<>	No igual a (distinto de)
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

(No haremos ejercicios todavía sobre estos operadores, sino que los aplicaremos en el siguiente tema, cuando aprendamos a comprobar condiciones.

Igual que antes, algunos de ellos (>=, <=, in) los utilizaremos también en los conjuntos, más adelante.

Tema 4.4: Operaciones entre bits.

Los operadores "and", "or" y "not", junto con otros, se pueden utilizar también para **operaciones entre bits** de números enteros:

Operador	Operación
not	Negación
and	Producto lógico
or	Suma lógica
xor	Suma exclusiva
shl	Desplazamiento hacia la izquierda
shr	Desplazamiento a la derecha

Explicar para qué sirven estos operadores implica conocer qué es eso de los bits, cómo se pasa un número decimal a binario, etc. Supondré que se tienen las nociones básicas, y pondré un ejemplo, cuyo resultado comentaré después:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Operaciones entre     }
{    bits                  }
{    BITOPS.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program BitOps;   { Operaciones entre bits }

const
  a  =  67;
  b  =  33;

begin
  writeln('La variable a vale ', a);
  writeln('y b vale ', b);
  writeln('  El complemento de a es: ', not(a));
  writeln('  El producto lógico de a y b es: ', a and b);
  writeln('  Su suma lógica es: ', a or b);
  writeln('  Su suma lógica exclusiva es: ', a xor b);
  writeln('  Desplacemos a a la izquierda: ', a shl 1);
  writeln('  Desplacemos a a la derecha: ', a shr 1);
end.
```

Veamos qué ha ocurrido. La respuesta que nos da Turbo Pascal 7.0 es la siguiente:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
La variable a vale 67
y b vale 33
El complemento de a es: -68
El producto lógico de a y b es: 1
Su suma lógica es: 99
Su suma lógica exclusiva es: 98
Desplacemos a a la izquierda: 134
Desplacemos a a la derecha: 33
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Para entender esto, deberemos convertir al sistema binario esos dos números:

```
67 = 0100 0011
33 = 0010 0001
```

- En primer lugar complementamos "a", cambiando los ceros por unos:

```
1011 1100 = -68
```

- Después hacemos el producto lógico de A y B, multiplicando cada bit,
de modo que 1*1 = 1, 1*0 = 0, 0*0 = 0

```
0000 0001 = 1
```

- Después hacemos su suma lógica, sumando cada bit, de modo que

```
1+1 = 1, 1+0 = 1, 0+0 = 0
0110 0011 = 99
```

- La suma lógica exclusiva devuelve un 1 cuando los dos bits son distintos:

```
1 xor 1 = 0, 1 xor 0 = 1, 0 xor 0 = 0

0110 0010 = 98
```

- Desplazar los bits una posición a la izquierda es como multiplicar por
dos:

```
1000 0110 = 134
```

- Desplazar los bits una posición a la derecha es como dividir entre
dos:

```
0010 0001 = 33
```

¿Y qué **utilidades** puede tener todo esto? Posiblemente, más de las que parece a primera vista. Por ejemplo: desplazar a la izquierda es una forma muy rápida de multiplicar por potencias de dos; desplazar a la derecha es dividir por

potencias de dos; la suma lógica exclusiva (xor) es un método rápido y sencillo de cifrar mensajes; el producto lógico nos permite obligar a que ciertos bits sean 0; la suma lógica, por el contrario, puede servir para obligar a que ciertos bits sean 1...

Ejercicio propuesto: *Crea un programa que pida al usuario un número entero, le aplique una operación XOR 21 y muestre el resultado, aplique nuevamente una operación XOR 21 sobre dicho resultado y muestre el valor final obtenido*

Curso de Pascal. Tema 4.5: Precedencia de los operadores.

Para terminar este tema, debemos conocer la **precedencia** (o **prioridad**) de los operadores:

Operadores	Precedencia	Categoría
@ not	Mayor (1ª)	Operadores unarios
* / div mod and shl shr	2ª	Operadores de multiplicación
+ - or xor	3ª	Operadores de suma
= <> < > <= >= in	Menor (4ª)	Operadores relacionales

Esto quiere decir que si escribimos algo como 2+3*4, el ordenador primero multiplicará 3*4 (la multiplicación tiene mayor prioridad que la suma) y luego sumaría 2 a ese resultado, de modo que obtendríamos 14. Si queremos que se realice antes la suma, que tiene menor nivel de precedencia, deberíamos emplear paréntesis, así: (2+3)*4

Y queda como **ejercicio** hallar (y tratar de entender) el resultado de este programita:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Prueba de prioridad   }
{    en las operaciones    }
{    elementales.          }
{    EJT04.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
```

```
{     - Surpas 1.00          }
{--------------------------}

Program EjT04;

begin
  writeln('Allá vamos... ');
  writeln( 5+3+4*5*2 );
  writeln( (5+3)*4+3*5-8/2+7/(3-2) );
  writeln( 5 div 3 + 23 mod 4 - 4 * 5 );
end.
```

Tema 5: Condiciones.

Vamos a ver cómo podemos evaluar condiciones desde Pascal. La primera construcción que trataremos es **if ... then**. En español sería "si ... entonces", que expresa bastante bien lo que podemos hacer con ella. El formato es "**if condicion then sentencia**". Veamos un ejemplo breve antes de seguir:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primera prueba de     }
{    "if"                  }
{    IF1.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program if1;

var numero: integer;

begin
  writeln('Escriba un número');
  readln(numero);
  if numero>0 then writeln('El número es positivo');
end.
```

Todo claro, ¿verdad? La "condición" debe ser una expresión que devuelva un valor del tipo "**boolean**" (verdadero/falso). La "sentencia" se ejecutará si ese valor es "cierto" (TRUE). Este valor puede ser tanto el resultado de una comparación (como hemos hecho en el ejemplo anterior), como una propia variable booleana (forma que veremos a continuación).

Ejercicio propuesto: *Crea un programa que pida al usuario un número entero y diga si ha tecleado el número 5 o ha tecleado un número distinto.*

Ejercicio propuesto: *Crea un programa que pida al usuario un número real y diga si es mayor de 10.*
Ejercicio propuesto: *Crea un programa que pida al usuario un número entero y diga si es par (pista: habrá que comprobar el resto de la división entre 2).*

Tema 5.2: Condiciones y variables boolean.

Así, una forma más "rebuscada" (pero que a veces resultará más cómoda y más legible) de hacer lo anterior sería, usando una variable "**boolean**":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Segunda prueba de     }
{    "if"                  }
{    IF2.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program if2;

var
  numero: integer;
  esPositivo: boolean;

begin
  writeln('Escriba un número');
  readln(numero);
  esPositivo := (numero>0);
  if esPositivo then writeln('El número es positivo');
end.
```

Cuando veamos en el próximo tema las órdenes para controlar el flujo del programa, seguiremos descubriendo aplicaciones de las variables booleanas, que muchas veces uno considera "poco útiles" cuando está aprendiendo.

Tema 5.3: Condiciones y sentencias compuestas.

La "sentencia" que sigue a "if .. then" puede ser una sentencia simple o **compuesta**. Las sentencias compuestas se forman agrupando varias simples entre un "begin" y un "end":

```
{--------------------------}
{  Ejemplo en Pascal:      }
```

```
{                          }
{    Tercera prueba de     }
{    "if"                  }
{    IF3.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program if3;

var
  numero: integer;

begin
  writeln('Escriba un número');
  readln(numero);
  if numero<0 then
    begin
    writeln('El número es negativo.  Pulse INTRO para seguir.');
    readln
    end;
end.
```

En este ejemplo, si el número es negativo, se ejecutan dos acciones: escribir un mensaje en pantalla y esperar a que el usuario pulse INTRO (o ENTER, o RETURN, o <-+, según sea nuestro teclado), lo que podemos conseguir usando "readln" pero sin indicar ninguna variable en la que queremos almacenar lo que el usuario teclee.

Nota: nuevamente, hemos empleado la **escritura indentada** para intentar que el programa resulte más legible: los pasos que se dan si se cumple la condición aparecen más a la derecha que el resto, para que resulten más fáciles de identificar.

Ejercicio propuesto: *Crea un programa que pida al usuario un su nombre. Sólo si ese nombre es "Juan", deberá entonces pedirle una contraseña, y en caso de que la contraseña sea "1234", le responderá diciendo "Bienvenido!"*

Tema 5.4: Si no se cumple la condición.

Sigamos... También podemos indicar lo que queremos que se haga **si no se cumple** la condición. Para ello tenemos la construcción "if condición then sentencia1 **else** sentencia2":

```
{--------------------------}
{  Ejemplo en Pascal:      }
```

```
{                          }
{    Cuarta prueba de      }
{    "if"                  }
{    IF4.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program if4;

var
  numero: integer;

begin
  writeln('Escriba un número');
  readln(numero);
  if numero<0 then
    writeln('El número es negativo.')
  else
    writeln('El número es positivo o cero.')
end.
```

Un detalle importante que conviene tener en cuenta es que antes del "else" **no debe haber** un punto y coma, porque eso indicaría el final de la sentencia "if...", y el compilador nos avisaría con un error.

Ejercicio propuesto: *Crea un programa que pida al usuario un número real y diga si ha tecleado el número 5 o ha tecleado un número distinto, usando "else".*
Ejercicio propuesto: *Crea un programa que pida al usuario un número entero y diga si es par o es impar.*

Tema 5.5: Sentencias "If" encadenadas.

Las sentencias "if...then...else" se pueden **encadenar**:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Quinta prueba de      }
{    "if"                  }
{    IF5.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
```

```
{     - Free Pascal 2.2.0w  }
{     - Turbo Pascal 7.0    }
{     - Turbo Pascal 5.0    }
{     - Surpas 1.00         }
{---------------------------}

program if5;

var
  numero: integer;

begin
  writeln('Escriba un número');
  readln(numero);
  if numero<0 then
    writeln('El número es negativo.')
  else if numero>0 then
    writeln('El número es positivo.')
  else
    writeln('El número es cero.')
end.
```

Ejercicio propuesto: *Crea un programa que pida al usuario un número entero y diga si es mayor de 10, es menor de 10 o es exactamente 10.*
Ejercicio propuesto: *Crea un programa que pida al usuario dos números reales y que diga si son iguales, o, en caso contrario, diga cual es el mayor de los dos.*

Tema 5.6: Varias condiciones simultáneas.

Si se deben cumplir **varias condiciones** a la vez, podemos **enlazarlas** con "and" . Si se pueden cumplir varias, usaremos "or" (o). Para negar, "not" (no):

```
if ( opcion = 1 ) and ( terminado = true ) then [...]
if ( opcion = 3 ) or ( teclaPulsada = true ) then [...]
if not ( preparado ) then [...]
if ( opcion = 2 ) and not ( nivelDeAcceso < 40 ) then [...]
```

Ejercicio propuesto: *Crea un programa que pida al usuario un número entero y diga si es par y a la vez múltiplo de 3.*
Ejercicio propuesto: *Crea un programa que pida al usuario dos números reales y diga si ambos positivos.*

Pero cuando queremos comprobar entre **varios posibles valores**, sería muy pesado tener que hacerlo con muchos "if" seguidos o encadenar muchos con "and" u "or".. Hay una alternativa que resulta mucho más cómoda: la orden **case**. Su sintaxis es

```
case expresión of
caso1: sentencia1;
caso2: sentencia2;
...
casoN: sentenciaN;
end;
```

o bien, si queremos indicar lo que se debe hacer si no coincide con ninguno de los valores que hemos enumerado, usamos else:

```
case expresión of
caso1: sentencia1;
caso2: sentencia2;
...
casoN: sentenciaN;
else
otraSentencia;
end;
```

En **Pascal estándar**, esta construcción se empleaba con **otherwise** en lugar de "else" para significar "en caso contrario", así que si alguien de los que me lee no usa TP/BP, sino un compilador que protesta con el "else" (es el caso de Surpas), ya sabe dónde probar...

Con un ejemplito se verá más claro cómo usar "case":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Condiciones múltiples }
{    con "case"            }
{    CASE1.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program case1;

var
  letra: char;

begin
  WriteLn('Escriba un símbolo');
  ReadLn(letra);
  case letra of
```

```
      ' ':                  WriteLn('Un espacio');
      'A'..'Z', 'a'..'z':  WriteLn('Una letra');
      '0'..'9':             WriteLn('Un dígito');
      '+', '-', '*', '/':  WriteLn('Un operador');
    else  { otherwise en SURPAS }
      WriteLn('No es espacio, ni letra, ni dígito, ni operador');
    end;
  end.
```

Como último comentario: la "expresión" debe pertenecer a un tipo de datos con un **número finito** de elementos, como "integer" o "char", pero no "real".

Y como se ve en el ejemplo, los "**casos**" posibles pueden ser valores únicos, varios valores separados por comas, o un rango de valores separados por .. (como los puntos suspensivos, pero **sólo dos**, al igual que en los "arrays").

Ejercicio propuesto: *Crea un programa que pida al usuario un número entero del 1 al 3, y escriba en pantalla "Uno", "Dos", "Tres" o "Número incorrecto" según corresponda.*

Curso de Pascal. Tema 6: Bucles.

Vamos a ver cómo podemos crear **bucles**, es decir, partes del programa que se repitan un cierto número de veces.

Según cómo queramos que se controle ese bucle, tenemos **tres posibilidades**, que vamos a comentar en primer lugar:

- **for..to**: La orden se repite desde que una variable tiene un valor inicial hasta que alcanza otro valor final (un cierto NÚMERO de veces).
- **while..do**: Repite una sentencia MIENTRAS que sea cierta la condición que indicamos (se verá en el apartado 6.5).
- **repeat..until**: Repite un grupo de sentencias HASTA que se dé una condición (se verá en el apartado 6.6).

La **diferencia** entre estos dos últimos es que "while" comprueba la condición antes de repetir las demás sentencias, por lo que puede que estas sentencias ni siquiera se lleguen a ejecutar, si la condición de entrada es falsa. En "repeat", la condición se comprueba al final, de modo que las sentencias intermedias se ejecutarán al menos una vez.

Vamos a verlos con más detalle...

El **formato de "for"** es

```
for variable := ValorInicial to ValorFinal do
Sentencia;
```

Se podría traducir por algo como "desde que la variable valga ValorInicial hasta que valga ValorFinal" (y en cada pasada, su valor aumentará en una unidad).

Como primer ejemplo, vamos a ver un pequeño programa que escriba los números del uno al diez:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer ejemplo de     }
{    "for": contador       }
{    FOR1.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

Program For1;

var
  contador: integer;

begin
  for contador := 1 to 10 do
    writeln( contador );
end.
```

Ejercicio propuesto: *Crea un programa que muestre los números del 10 al 15, cada uno en una línea.*
Ejercicio propuesto: *Crea un programa que muestre los números del 5 al 20, todos ellos en la misma línea, separados por espacios en blanco.*
Ejercicio propuesto: *Crea un programa que escriba 10 veces "Hola" (en la misma línea).*

Tema 6.2: "For" encadenados.

Los bucles "for" se pueden **enlazar** uno dentro de otro, de modo que podríamos escribier las tablas de multiplicar del 1 al 5, dando 5 pasos, cada uno de los cuales incluye otros 10, así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
```

```
{                          }
{    Segundo ejemplo de    }
{    "for": bucles enlaza- }
{    dos -> tabla de       }
{    multiplicar           }
{    FOR2.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

Program For2;

var
  tabla, numero: integer;

begin
  for tabla := 1 to 5 do
    for numero := 1 to 10 do
      writeln( tabla, ' por ', numero ,' es ', tabla * numero );
end.
```

Ejercicio propuesto: *Crea un programa que escriba tres veces seguidas los números del 1 al 3.*
Ejercicio propuesto: *Crea un programa que escriba 3 líneas, cada una de las cuales contendrá 4 veces la palabra "Hola".*

Tema 6.3: "For" y sentencias compuestas.

Hasta ahora hemos visto sólo casos en los que después de "for" había un única sentencia. ¿Qué ocurre si queremos repetir **más de una orden**? Basta encerrarlas entre "begin" y "end" para convertirlas en una **sentencia compuesta**, como hemos hecho hasta ahora.

Así, vamos a mejorar el ejemplo anterior haciendo que deje una línea en blanco entre tabla y tabla:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Tercer ejemplo de     }
{    "for": bucles con     }
{    sentencias compuestas }
{    FOR3.PAS              }
{                          }
```

```
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{    - Turbo Pascal 5.0     }
{    - Surpas 1.00          }
{--------------------------}

Program For3;

 var
   tabla, numero: integer;

begin
  for tabla := 1 to 5 do
    begin
    for numero := 1 to 10 do
      writeln( tabla, ' por ', numero ,' es ', tabla * numero );
    writeln;                        (* Línea en blanco *)
    end;
end.
```

Recordad, como vimos, que es muy conveniente usar la**escritura indentada**, que en este caso ayuda a ver dónde empieza y termina lo que hace cada "for"... espero O:-)

Tema 6.4: Contar sin números.

Una observación: para "contar" no necesariamente hay que usar **números**, también podemos contar con letras:

```
{--------------------------}
{  Ejemplo en Pascal:       }
{                           }
{    Cuarto ejemplo de      }
{    "for": letras como     }
{    índices en un bucle    }
{    FOR4.PAS               }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{    - Turbo Pascal 5.0     }
{    - Surpas 1.00          }
{--------------------------}

Program For4;
```

```
var
  letra: char;

begin
  for letra := 'a' to 'z' do
    write( letra );
end.
```

Como último comentario: con el bucle "for", tal y como lo hemos visto, sólo se puede contar en forma creciente y de uno en uno. Para contar de forma **decreciente**, se usa "downto" en vez de "to".

Ejercicio propuesto: *Crea un programa que escriba las letras de la B a la M.*
Ejercicio propuesto: *Crea un programa que escriba las letras de la Z a la A (de forma descendente).*
Ejercicio propuesto: *Crea un programa que escriba los números del 10 al 1 (de forma descendente).*

Para contar de dos en dos (por ejemplo), hay que buscarse la vida: multiplicar por dos o sumar uno dentro del cuerpo del bucle, etc... Eso sí, sin modificar la variable que controla el bucle (usar cosas como "write(x*2)" en vez de "x := x*2", que pueden dar problemas en algunos compiladores).

Pero todo eso os dejo que lo investigueis con los ejercicios... }:-)

Tema 6.4 (b): Ejercicios sobre "For".

Ejercicio propuesto: *Un programa que escriba la secuencia de números 2, 4, 6, 8 ... 16.*
Ejercicio propuesto: *Un programa que escriba la secuencia de números 6, 5, 4,..., 1.*
Ejercicio propuesto: *Un programa que escriba la secuencia de números 3, 5, 7,..., 21.*
Ejercicio propuesto: *Un programa que escriba la secuencia de números 12, 10, 8,..., 0..*
Ejercicio propuesto: *Crea un programa que sume dos vectores, cuyos componentes indicará el usuario. Por ejemplo, la suma de (1,2,3) y (7,11,-1) sería (8,13,2).*
Ejercicio propuesto: *Crea un programa que halle el producto escalar dos vectores, cuyos componentes indicará el usuario.*
Ejercicio propuesto: *Crea un programa que multiplique dos matrices.*
Ejercicio propuesto: *Para los más osados (y que conozcan el problema), un programa de resolución de sistemas de ecuaciones por Gauss.*

Tema 6.5: "While".

While

Vimos como podíamos crear estructuras repetitivas con la orden "for", y comentamos que se podía hacer también con "while..do", comprobando una condición al principio, o con "repeat..until", comprobando la condición al final de cada repetición. Vamos a ver estas dos con más detalle:

La sintaxis de "while" es

```
while condición do
sentencia;
```

Que se podría traducir como "MIENTRAS se cumpla la condición HAZ sentencia".

Un ejemplo que nos diga la longitud de todas las frases que queramos es:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de "While":   }
{    muestra la longitud   }
{    del texto tecleado    }
{    WHILE1.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{--------------------------}

Program While1;

var
  frase: string;

begin
  writeln('Escribe frases, y deja una línea en blanco para salir');
  write( '¿Primera frase?' );
  readln( frase );
  while frase <> '' do
    begin
    writeln( 'Su longitud es ', length(frase) );
     { SURPAS 1.00 no reconoce "length" }
    write( '¿Siguiente frase?' );
    readln( frase )
    end
end.
```

En el ejemplo anterior, sólo se entra al bloque begin-end (una sentencia compuesta) si la primera palabra es correcta (no es una línea en blanco).

Entonces escribe su longitud, pide la siguiente frase y vuelve a comprobar que es correcta.

Como comentario casi innecesario, **length** es una función que nos dice cuantos caracteres componen una cadena de texto.

Si ya de principio la condición es **falsa**, entonces la sentencia no se ejecuta ninguna vez, como pasa en este ejemplo:

```
while (2<1) do
writeln('Dos es menor que uno');
```

Ejercicio propuesto: *Crea un programa vaya sumando los números que el usuario introduzca, y mostrando dicha suma, hasta que introduzca el número 0, usando "while".*
Ejercicio propuesto: *Crea un programa que pida al usuario su contraseña. Deberá terminar cuando introduzca como contraseña la palabra "acceso", pero volvérsela a pedir tantas veces como sea necesario.*
Ejercicio propuesto: *Crea un programa que escriba en pantalla los números del 1 al 10, usando "while".*

Tema 6.6: "Repeat".

Repeat..until

Para "repeat..until", la sintaxis es

```
repeat
sentencia;
...
sentencia;
sentencia
until condición;
```

Es decir, REPITE un grupo de sentencias HASTA que la condición sea cierta. Ojo con eso: es un **grupo de sentencias**, no sólo una, como ocurría en "while", de modo que ahora no necesitaremos "begin" y "end" para crear sentencias compuestas.

El conjunto de sentencias se ejecutará **al menos una vez**, porque la comprobación se realiza al final.

Como último detalle, de menor importancia, no hace falta terminar con punto y coma la sentencia que va justo antes de "until", al igual que ocurre con "end".

Un ejemplo clásico es la "clave de acceso" de un programa, que iremos mejorando cuando veamos distintas formas de "esconder" lo que se teclea, bien cambiando colores o bien escribiendo otras letras, como * (empleando métodos que veremos en el tema 10, y lo aplicaremos a un "juego del ahorcado").

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de "Repeat":  }
{    comprobación de una   }
{    clave de acceso       }
{    REPEAT.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program ClaveDeAcceso;

var
  ClaveCorrecta, Intento: String[30];

begin
  ClaveCorrecta := 'PaskalForever';
  repeat
    WriteLn( 'Introduce la clave de acceso...' );
    ReadLn( Intento )
  until Intento = ClaveCorrecta
  (* Aquí iría el resto del programa *)
end.
```

Se entiende, ¿verdad?

Ejercicio propuesto: *Crea un programa que pida números positivos al usuario, y vaya calculando la suma de todos ellos (terminará cuando se teclea un número negativo o cero), usando "repeat".*
Ejercicio propuesto: *Crea un programa que escriba en pantalla los números pares del 26 al 10 (descen diendo), usando "repeat".*
Ejercicio propuesto: *Crea un programa que pida al usuario su nombre de usuario y su contraseña, y no le permita seguir hasta que introduzca como nombre "yo" y como contraseña "acceso", usando "repeat"*

Es bastante por hoy. Ahora os toca experimentar a vosotros.

Curso de Pascal. Tema 6.7: Ejercicios sobre "While" y "Repeat".

Ejercicio propuesto: *Mejorar el programa de la clave de acceso con "while" (6.5b), para que avise de que la clave no es correcta..*
Ejercicio propuesto: *Mejorar el programa de la clave de acceso con "repeat" (6.6c), para que avise de que la clave no es correcta..*
Ejercicio propuesto: *Mejorar más todavía el programa de la clave de acceso con "while", para que sólo haya tres intentos.*
Ejercicio propuesto: *Mejorar más todavía el programa de la clave de acceso con "while", para que sólo haya tres intentos.*

Por cierto, si alguien viene de Basic puede que se pregunte "¿Y mi **goto**? ¿No existe en Pascal?" Pues sí, existe, pero no contaremos nada sobre él por ahora, porque va en contra de todos los principios de la Programación Estructurada, su uso sólo es razonable en casos muy concretos que todavía no necesitamos.

Ya se verá más adelante la forma de usarlo.

Curso de Pascal. Tema 6.8: Ejemplo - Adivinar números.

Vamos a ver un ejemplo sencillo que use parte de lo que hemos visto hasta ahora.

Será un programa de adivinar números: un primer usuario deberá introducir un número que un segundo usuario deberá adivinar. También deberá indicar cuantos intentos va a permitir que tenga. Entonces se borra la pantalla, y se comienza a preguntar al segundo usuario qué número cree que es. Se le avisará si se pasa o se queda corto. El juego termina cuando el segundo usuario agote todos sus intentos o acierte el número propuesto.

Allá va...

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de "Repeat":  }
{    adivinar un número    }
{    ADIVINA.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
```

```
{     - Surpas 1.00            }
{     - Tmt Pascal Lt 1.20     }
{--------------------------}

program adivina;

var
  correcto,                 { Número que se debe acertar }
  probado,                  { El número que se prueba }
  maxIntentos,              { Máximo de intentos permitido }
  intentoAct: integer;      { El número de intento actual }
  i: byte;                  { Para bucles }

begin
  { Primero pedimos los datos iniciales }
  write('Usuario 1: Introduzca el número a adivinar. ');

  readln(correcto);
  write('¿Cuantos intentos va a permitir? ');
  readln(maxIntentos);
  { Borramos la pantalla de forma "fea" pero que sirve }
  for i:= 1 to 25 do writeln;

  intentoAct := 0;  { Aún no ha probado }
  { Comienza la parte repetitiva }
  repeat

    writeln;
    write('Usuario 2: Introduzca un número. ');
    readln(probado);                    { Pedimos un número }

    if probado > correcto then          { Puede ser mayor }

      writeln('Se ha pasado!')
      else if probado < correcto then   { Menor }
          writeln('Se ha quedado corto!')
          else writeln('Acertó!');      { O el correcto }

    intentoAct := intentoAct + 1;       { Ha gastado un intento más }
    writeln('Ha gastado ', intentoAct,
      ' intentos de ',                  { Le recordamos cómo va }
      maxIntentos,' totales.');

  until (intentoAct >= maxIntentos)     { Seguimos hasta que gaste
todos }
    or (probado = correcto);            { o acierte }

  if (intentoAct >= maxIntentos) and (probado <> correcto) then
    writeln('Lo siento, ha agotado sus intentos.');

end.
```

(Más adelante podrás encontrar un ejemplo más desarrollado: el juego del ahorcado, como parte del tema 10).

Tema 7: Constantes y tipos.

Definición de constantes

Cuando desarrollamos un programa, nos podemos encontrar con que hay variables que realmente "no varían" a lo largo de la ejecución de un programa, sino que su valor es **constante**.

Hay una manera especial de definirlas, que es con el especificador "**const**", que tiene el formato

```
const Nombre = Valor;
```

Veamos un par de ejemplos antes de seguir

```
const MiNombre = 'Nacho Cabanes';
const PI = 3.1415926535;
const LongitudMaxima = 128;
```

Estas constantes se manejan igual que variables como las que habíamos visto hasta hora, sólo que no se puede cambiar su valor. Así, es valido hacer

```
Writeln(MiNombre);
if Longitud > LongitudMaxima then ...
OtraVariable := MiNombre;
LongCircunf := 2 * PI * r;
```

pero no podríamos hacer

```
PI := 3.14;
MiNombre := 'Nacho';
LongitudMaxima := LongitudMaxima + 10;
```

Las constantes son mucho más prácticas de lo que puede parecer a primera vista (especialmente para quien venga de lenguajes como Basic, en el que no existen -en el Basic "de siempre", puede que sí existan en los últimas versiones del lenguaje-). Me explico con un ejemplillo:

Supongamos que estamos haciendo nuestra agenda en Pascal (ya falta menos para que sea verdad), y estamos tan orgullosos de ella que queremos que en cada pantalla de cada parte del programa aparezca nuestro nombre, el del programa y la versión actual. Si lo escribimos "de nuevas" cada vez, además de perder tiempo tecleando más, corremos el riesgo de que un día queramos cambiar el nombre (ya no se llamará "Agenda" sino "SuperAgenda" ;-)) pero lo hagamos en unas partes sí y en otras no, etc., y el resultado tan maravilloso quede estropeado por esos "detalles".
O si queremos cambiar la anchura de cada dato que guardamos de nuestros amigos, porque el espacio para el nombre nos había quedado demasiado escaso, tendríamos que recorrer todo el programa de arriba a abajo, con los mismos problemas, pero esta vez más graves aún, porque puede que intentemos grabar una ficha con una tamaño y leerla con otro distinto...

¿Solución? Pues definir todo ese tipo de datos como constantes al principio del programa, de modo que con un vistazo a esta zona podemos hacer cambios globales:

```
const
Nombre = 'Nacho';
Prog = 'SuperAgenda en Pascal';
Versión = 1.95;

LongNombre = 40;
LongTelef = 9;
LongDirec = 60;
...
```

Las declaraciones de las constantes se hacen antes del cuerpo del programa principal, y generalmente antes de las declaraciones de variables:

```
program MiniAgenda;
const
NumFichas = 50;
var
Datos: array[ 1..NumFichas ] of string;
begin
...
```

Ejercicio propuesto: *Modificar el programa de adivinar números (apartado 6.8) para que el número a adivinar deba estar entre 1 y 1000 (estos límites se definirán mediante constantes) y el máximo de intentos permitidos sea 10 (también mediante una constante).*

Tema 7.2: Constantes "con tipo".

El identificador "const" tiene también en Turbo Pascal otro uso menos habitual: definir lo que se suele llamar **constantes con tipo**, que son **variables normales** y corrientes, pero a las que damos un valor inicial antes de que comience a ejecutarse el programa. Se usa

```
const variable: tipo = valor;
```

Así, volviendo al ejemplo de la clave de acceso, podíamos tener una variables "intentos" que dijese el número de intentos. Hasta ahora habríamos hecho

```
var
intentos: integer;
```

```
begin
intentos := 3;
...
```

Ahora ya sabemos que sería mejor hacer, si sabemos que el valor no va a cambiar:

```
const
intentos = 3;

begin
...
```

Pero si se nos da el caso de que vemos por el nombre que es alguien de confianza, que puede haber olvidado su clave de acceso, quizá nos interese permitirle 5 o más intentos. ¿Qué hacemos? Ya no podemos usar "const" porque el valor puede variar, pero por otra parte, siempre comenzamos concediendo 3 intentos, hasta comprobar si es alguien de fiar. Conclusión: podemos hacer

```
const intentos: integer = 3;

begin
...
```

Insisto: a efectos de manejo, una "constante con tipo" es **exactamente igual que una variable**, con las ventajas de que está más fácil de localizar si queremos cambiar su valor inicial y de que el compilador optimiza un poco el código, haciendo el programa unos bytes más pequeño.

Tema 7.3: Definición de tipos.

El "tipo" de una variable es lo que determina qué clase de valores podremos guardar en ella. Para nosotros, es lo que indicamos junto a su nombre cuando la declaramos. Por ejemplo,

```
var PrimerNumero: integer;
```

indica que vamos a usar una variable que se va a llamar PrimerNumero y que almacenará **valores** de tipo entero. Si queremos definir una de las fichas de lo que será nuestra agenda, también haríamos:

```
var ficha: record
nombre: string;
direccion: string;
edad: integer;
```

```
observaciones: string
end;
```

Tampoco hay ningún problema con esto, ¿verdad? Y si podemos utilizar variables creando los tipos "en el momento", como en el caso anterior, ¿para qué necesitamos definir tipos? Vamos a verlo con un ejemplo. Supongamos que vamos a tener ahora dos variables: una "ficha1" que contendrá el dato de la ficha actual y otra "ficha2" en la que almacenaremos datos temporales. Veamos qué pasa...

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Intenta asignar dos   }
{    tipos definidos como  }
{    distintos. Da error   }
{    y no compila.         }
{    TIPOS1.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program PruebaTipos;

var
  ficha1: record
    nombre: string[20];
    direccion: string[50];
    edad: integer;
    observaciones: string[70]
  end;
  ficha2: record
    nombre: string[20];
    direccion: string[50];
    edad: integer;
    observaciones: string[70]
  end;

begin
  ficha1.nombre := 'Pepe';
  ficha1.direccion := 'Su casa';

  ficha1.edad := 65;
  ficha1.observaciones := 'El mayor de mis amigos... ;-)  ';
  ficha2 := ficha1;
  writeln( ficha2.nombre );
end.
```

¿Qué haría este programa? Es fácil de seguir: define dos variables que van a guardar la misma clase de datos. Da valores a cada uno de los datos que almacenará una de ellas. Después hacemos que la segunda valga lo mismo que la primera, e imprimimos el nombre de la segunda. Aparecerá escrito "Pepe" en la pantalla, ¿verdad?

¡Pues **no**! Aunque a nuestros ojos "ficha1" y "ficha2" sean iguales, para el compilador no es así, por lo que protesta y el programa ni siquiera llega a ejecutarse. Es decir: las hemos definido para que almacene la misma clase de valores, pero **no son del mismo tipo**.

Esto es fácil de solucionar:

```
var ficha1, ficha2: record
    nombre: string;
    direccion: string;
    edad: integer;
    observaciones: string
  end;

begin
...
```

Si las definimos a la vez, SI QUE SON DEL MISMO TIPO. Pero surge un problema del que os iréis dando cuenta a partir del próximo día, que empezaremos a crear funciones y procedimientos. ¿Qué ocurre si queremos usar en alguna parte del programa otras variables que también sean de ese tipo? ¿Las definimos también a la vez? En muchas ocasiones no será posible.

Así que tiene que haber una forma de indicar que todo eso que sigue a la palabra "record" es un tipo al que nosotros queremos acceder con la misma comodidad que si fuese "integer" o "boolean", queremos **definir** un tipo, no simplemente declararlo, como estábamos haciendo.

Pues es sencillo:

```
type NombreDeTipo = DeclaracionDeTipo;
```

o en nuestro caso

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Asignación correcta   }
{    de tipos              }
{    TIPOS2.PAS            }
{                          }
```

```
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program PruebaTipos2;

type TipoFicha = record
    nombre: string[20];
    direccion: string[50];
    edad: integer;
    observaciones: string[70]
  end;

var ficha1, ficha2: TipoFicha;

begin
  ficha1.nombre := 'Pepe';
  ficha1.direccion := 'Su casa';
  ficha1.edad := 65;
  ficha1.observaciones := 'El mayor de mis amigos... ;-)  ';
  ficha2 := ficha1;
  writeln( ficha2.nombre );
end.
```

Ahora sí que podremos asignar valores entre variables que hayamos definido en distintas partes del programa, podremos usar esos tipos para crear ficheros (eso lo veremos en el tema 11), etc, etc, etc...

Ejercicio propuesto: *Crear un tipo de datos llamado "punto", que sea un registro formado por dos componentes, X e Y, números reales. Pedir al usuario el valor de X e Y para dos puntos distintos y calcular la distancia entre esos dos puntos.*

Tema 8: Procedimientos y funciones.

Tema 8.1: Procedimientos.

La **programación estructurada** trata de dividir el programa el bloques más pequeños, buscando una mayor legibilidad, y más comodidad a la hora de corregir o ampliar.

Por ejemplo, en el caso de nuestra maravillosa agenda, podemos empezar a teclear directamente y crear un programa de 2000 líneas que quizás incluso funcione, o dividirlo en partes, de modo que el cuerpo del programa sea

```
begin
  InicializaVariables;
  PantallaPresentacion;
  Repeat
  PideOpcion;
  case Opcion of
      '1': MasDatos;
      '2': CorregirActual;
      '3': Imprimir;
    ...
    end;
  Until Opcion = OpcionDeSalida;
  GuardaCambios;
  LiberaMemoria
end.
```

```
begin
  InicializaVariables;
  PantallaPresentacion;
  Repeat
  PideOpcion;
  case Opcion of
      '1': MasDatos;
      '2': CorregirActual;
      '3': Imprimir;
    ...
    end;
  Until Opcion = OpcionDeSalida;
  GuardaCambios;
  LiberaMemoria
end.
```

Bastante más fácil de seguir, ¿verdad?

En nuestro caso (en el lenguaje Pascal), estos bloques serán de dos tipos: **procedimientos** (procedure) y **funciones** (function).

La **diferencia** entre ellos es que un procedimiento ejecuta una serie de acciones que están relacionadas entre sí, y no devuelve ningún valor, mientras que la función sí que va a devolver valores. Veamoslo con un par de ejemplos:

```
procedure Acceso;
var
    clave: string; (* Esta variable es local *)
begin
    writeln(' Bienvenido a SuperAgenda ');
    writeln('=========================='); (* Para subrayar *)
    writeln; writeln; (* Dos líneas en blanco *)
    writeln('Introduzca su clave de acceso');
    readln( clave ); (* Lee un valor *)
    if clave <> ClaveCorrecta then (* Compara con el correcto *)
        begin (* Si no lo es *)
        writeln('La clave no es correcta!'); (* avisa y *)
        exit (* abandona el programa *)
```

```
        end
end;
```

Primeros **comentarios** sobre este ejemplo:

- El **cuerpo de un procedimiento** se encierra entre "begin" y "end", igual que las sentencias compuestas y que el propio cuerpo del programa.
- Un procedimiento puede tener sus propias variables, que llamaremos **variables locales**, frente a las del resto del programa, que son **globales**. Desde dentro de un procedimiento podemos acceder a las variables globales (como ClaveCorrecta del ejemplo anterior), pero desde fuera de un procedimiento no podemos acceder a las variables locales que hemos definido dentro de él.
- La orden **exit**, que no habíamos visto aún, permite interrumpir la ejecución del programa (o de un procedimiento) en un determinado momento.

Ejercicio propuesto: *Crear un procedimiento "LimpiarPantalla", que escriba 25 líneas en blanco en la pantalla, de modo que el texto que estaba escrito anteriormente en pantalla deje de estar visible (en una pantalla de texto convencional de 80x25 caracteres).*

Tema 8.2: Funciones.

Veamos el segundo ejemplo: una **función** que eleve un número a otro (esa posibilidad no existe "de forma nativa" en Pascal), se podría hacer así, si ambos son enteros:

```
function potencia(a,b: integer): integer; (* a elevado a b *)
var
    i: integer;          (* para bucles *)
    temporal: integer;   (* para el valor temporal *)
begin
    temporal := 1;                (* incialización *)
    for i := 1 to b do
    temporal := temporal * a;     (* hacemos 'b' veces 'a*a' *)
    potencia := temporal;         (* y finalmente damos el valor *)
end;
```

Comentemos cosas también:

- Esta función se llama "potencia".
- Tiene dos **parámetros** llamados "a" y "b" que son números enteros (valores que "se le pasan" a la función para que trabaje con ellos).
- El resultado va a ser también un número entero.

- "i" y "temporal" son variables locales: una para el bucle "for" y la otra almacena el valor temporal del producto.
- Antes de salir es cuando asignamos a la función el que será su valor definitivo.

Ejercicio propuesto: *Crear una función "triple", que reciba un número real como parámetro, y devuelva como resultado ese número multiplicado por tres.*
Ejercicio propuesto: *Crear una función "media", que reciba dos números reales como parámetros, y devuelva como resultado su media artimética.*

(Nota: al final de este apartado tienes la lista de funciones matemáticas y de manipulación de cadenas que incluye Turbo Pascal)

Tema 8.3: Uso de las funciones.

Pero vamos a ver un programita que use esta función, para que quede un poco más claro:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer ejemplo de     }
{    función: potencia     }
{    POTENCIA.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program PruebaDePotencia;

var
  numero1, numero2: integer;                (* Variables globales *)

function potencia(a,b: integer): integer;  (* Definimos la función *)
var
  i: integer;                     (* Locales: para bucles *)
  temporal: integer;              (* y para el valor temporal *)
begin
  temporal := 1;                  (* incialización *)
  for i := 1 to b do
    temporal := temporal * a;     (* hacemos "b" veces "a*a" *)
  potencia := temporal;           (* y finalmente damos el valor *)
end;
```

```
begin                                          (* Cuerpo del programa *)
  writeln('Potencia de un número entero');
  writeln;
  writeln('Introduce el primer número');
  readln( numero1 );
  writeln('Introduce el segundo número');
  readln( numero2 );
  writeln( numero1 ,' elevado a ', numero2 ,' vale ',
    potencia (numero1, numero2) )
end.
```

Tema 8.4: Procedimientos con parámetros.

Un procedimiento también puede tener "parámetros", igual que la función que acabamos de ver:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de procedi-   }
{    miento al que se le   }
{    pasan parámetros      }
{    PROCPAR.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{--------------------------}

program ProcConParametros;
{ Para usarlo con SURPAS 1.00 habría     }
{ definir un tipo "str20", por ejemplo,  }
{ que usar en vez de "string".           }

procedure saludo (nombre: string);      (* Nuestro procedimiento *)
begin
  writeln('Hola ', nombre, ' ¿qué tal estás?');
end;

begin                    (* Comienzo del programa *)
  writeln;               (* Linea en blanco *)
  saludo( 'Aurora' );    (* Saludamos a Aurora *)
end.                     (* Y se acabó *)
```

En el próximo apartado veremos la diferencia entre pasar parámetros por valor (lo que hemos estado haciendo) y por referencia (para poder modificarlos), y jugaremos un poco con la recursividad.

Ejercicio propuesto: *Crear una función "potenciaReal", que trabaje con números reales, y permita cálculos como 3.2 ^ 1.7 (pista; hay que usar una par de*

funciones matemáticas: las exponenciales y los logaritmos; si te rindes, puedes mirar la ampliación 1 del curso).
Ejercicio propuesto: *Hacer una función que halle la raíz cúbica del número que se le indique (pista: hallar una raíz cúbica es lo mismo que elevar a 1/3).*
Ejercicio propuesto: *Definir las funciones suma y producto de tres números, y crear un programa que haga una operación u otra según le indiquemos (empleando "case", etc).*
Ejercicio propuesto: *Un programita que halle la letra (NIF) que corresponde a un cierto DNI (documento nacional de identidad) español.*
Ejercicio propuesto: *Crea un programa que sume dos números "grandes", de 30 cifras. Esos números deberemos leerlos como "string" y sumarlos "letra a letra". Para ello, deberás crear una función "sumar", que reciba como parámetros dos "strings", y que devuelva su resultado en un nuevo "string".*
Ejercicio propuesto: *Crea un programa que multiplique dos números "grandes", de entre 30 y 100 cifras, por ejemplo. Para esos números no nos basta con los tipos numéricos que incorpora Pascal, sino que deberemos leerlos como "string" y pensar cómo multiplicar dos strings: ir cifra por cifra en cada uno de los factores y tener en cuenta lo que "me llevo"...*

Tema 8.5: Modificación de parámetros.

Ya habíamos visto qué era eso de los procedimientos y las funciones, pero habíamos dejado aparcados dos temas importantes: los tipos de parámetros y la recursividad.

Vamos con el primero. Ya habíamos visto, sin entrar en detalles, qué es eso de los **parámetros**: una serie de datos extra que indicábamos entre paréntesis en la cabecera de un procedimiento o función.

Es algo que estamos usando, sin saberlo, desde el primer día, cuando empezamos a usar "WriteLn":

writeln('Hola');

Esta línea es una llamada al procedimiento "WriteLn", y como parámetros le estamos indicando lo que queremos que escriba, en este caso, el texto "Hola".

Pero vamos a ver qué ocurre si hacemos cosas como ésta:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer procedimiento  }
{    que modif. variables  }
{    PROCMOD1.PAS          }
{                          }
```

```
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{    - Turbo Pascal 5.0     }
{    - Surpas 1.00          }
{--------------------------}

program PruebaDeParametros;

var dato: integer;

procedure modifica( variable : integer);
begin
  variable := 3 ;
  writeln( 'Dentro: ', variable );
end;

begin
  dato := 2;
  writeln( 'Antes: ', dato );
  modifica( dato );
  writeln( 'Después: ', dato );
end.
```

¿Qué podemos esperar que pase? Vamos a ir siguiendo cada instrucción:

- Declaramos el nombre del programa. No hay problema.
- Usaremos la variable "dato", de tipo entero.
- El procedimiento "modifica" toma una variable de tipo entero, le asigna el valor 3 y la escribe. Lógicamente, siempre escribirá 3.
- Empieza el cuerpo del programa: damos el valor 2 a "dato".
- Escribimos el valor de "dato". Claramente, será 2.
- Llamamos al procedimiento "modifica", que asigna el valor 3 a "dato" y lo escribe.
- Finalmente volvemos a escribir el valor de "dato"... ¿3?

¡¡¡¡NO!!!! Escribe **un 2**. Las modificaciones que hagamos a "dato" dentro del procedimiento modifica sólo son válidas mientras estemos **dentro** de ese procedimiento. Lo que modificamos es la variable genérica que hemos llamado "variable", y que no existe fuera del procedimiento.

Eso es **pasar un parámetro por valor**. Podemos leer su valor, pero no podemos alterarlo.

Pero, ¿cómo lo hacemos si realmente queremos modificar el parámetro. Pues nada más que añadir la palabra "var" delante de cada parámetro que queremos permitir que se pueda modificar...

Tema 8.6: Parámetros por referencia.

El programa quedaría:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Segundo proc. que     }
{    modifica variables    }
{    PROCMOD2.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program PruebaDeParametros2;

var dato: integer;

procedure modifica( var variable : integer);
begin
  variable := 3 ;
  writeln( 'Dentro: ', variable );
end;

begin
  dato := 2;
  writeln( 'Antes: ', dato );
  modifica( dato );
  writeln( 'Después: ', dato );
end.
```

Esta vez la última línea del programa sí que escribe un 3 y no un 2, porque hemos permitido que los cambios hechos a la variable salgan del procedimiento (para eso hemos añadido la palabra "var"). Esto es **pasar un parámetro por referencia**.

El nombre "**referencia**" alude a que no se pasa realmente al procedimiento o función el valor de la variable, sino la dirección de memoria en la que se encuentra, algo que más adelante llamaremos un "puntero".

Una de las aplicaciones más habituales de pasar parámetros por referencia es cuando una función debe devolver **más de un valor**. Habíamos visto que una función era como un procedimiento, pero además devolvía un valor (pero **sólo**

uno). Si queremos obtener más de un valor de salida, una de las formas de hacerlo es pasándolos como parámetros, precedidos por la palabra "var".

Ejercicio propuesto: *Crear una función "intercambia", que intercambie el valor de los dos números enteros que se le indiquen como parámetro.*
Ejercicio propuesto: *Crear una función "iniciales", que reciba una cadena como "Nacho Cabanes" y devuelva las letras N y C (primera letra, y letra situada tras el primer espacio), usando parámetros por referencia.*

Tema 8.7: Parámetros con el mismo nombre que las variables locales.

Y como ejercicio queda un caso un poco más "enrevesado". Qué ocurre si el primer programa lo modificamos para que sea así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Tercer proc. que      }
{    modifica variables    }
{    PROCMOD3.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program PruebaDeParametros3;

var dato: integer;

procedure modifica( dato : integer);
begin
  dato := 3 ;
  writeln( 'Dentro: ', dato );
end;

begin
  dato := 2;
  writeln( 'Antes: ', dato );
  modifica( dato );
  writeln( 'Después: ', dato );
end.
```

(Puedes consultar la respuesta)

Tema 8.8: Recursividad.

Vamos al segundo apartado de hoy: qué es eso de la "**recursividad**". Pues la idea en sí es muy sencilla: un procedimiento o función es recursivo si se llama a sí mismo.

¿Y qué utilidad puede tener eso? Habrá muchos problemas que son más fáciles de resolver si se descomponen en pasos cada vez más sencillos.

Vamos a verlo con un ejemplo clásico: el factorial de un número.

Partimos de la definición de factorial de un número n:

```
n! = n · (n-1) · (n-2) · ... · 3 · 2 · 1
```

Por otra parte, el factorial del siguiente número más pequeño (n-1) es

```
(n-1)! = (n-1) · (n-2) · ... · 3 · 2 · 1
```

Se parecen mucho. Luego podemos escribir cada factorial en función del factorial del siguiente número:

```
n! = n · (n-1)!
```

¡Acabamos de dar la definición recursiva del factorial! Así vamos "delegando" para que el problema lo resuelva el siguiente número, y este a su vez se lo pasa al siguiente, y este al otro, y así sucesivamente hasta llegar a algún caso que sea muy fácil.

Pues ahora sólo queda ver cómo se haría eso programando:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Factorial, ejemplo de }
{    recursividad          }
{    FACT.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{--------------------------}

program PruebaDeFactorial;

var numero: integer;
```

```
function factorial( num : integer) : integer;
begin
  if num = 1 then
    factorial := 1        (* Aseguramos que tenga salida siempre *)
  else
    factorial := num * factorial( num-1 );       (* Caso general *)
end;

begin
  writeln( 'Introduce un número entero (no muy grande)  ;-)' );
  readln(numero);
  writeln( 'Su factorial es ', factorial(numero) );
end.
```

Un par de comentarios sobre este programilla:

- Atención a la primera parte de la función recursiva: es **muy importante** comprobar que hay salida de la función, para que no se quede dando vueltas todo el tiempo y nos cuelgue el ordenador. Normalmente, la condición de salida será que ya hallamos llegado al caso fácil (en este ejemplo: el factorial de 1 es 1).
- No pongais números demasiado **grandes**. Recordad que los enteros van desde -32768 hasta 32767, luego si el resultado es mayor que este número, tendremos un desbordamiento y el resultado será erróneo. ¿Qué es "demasiado grande"? Pues el factorial de 8 es cerca de 40.000, luego sólo podremos usar números del 1 al 7.

Si este límite del tamaño de los enteros os parece preocupante, no le deis muchas vueltas, porque en la próxima lección veremos que hay otros tipos de datos que almacenan números más grandes o que nos permiten realizar ciertas cosas con más comodidad.

Ejercicio propuesto: *Crear una función recursiva que halle el producto de dos números enteros.*

Ejercicio propuesto: *Crear otra función que halle la potencia (a elevado a b), también recursiva.*

Ejercicio propuesto: *Hacer un programa que halle de forma recursiva el factorial de cualquier número, por grande que sea. (Pista: habrá que usar la rutina de multiplicar números grandes, que se propuso como ejemplo en el apartado anterior)*

Ejercicio propuesto: *Crear un programa que emplee recursividad para calcular un número de la serie Fibonacci (en la que los dos primeros elementos valen 1, y para los restantes, cada elemento es la suma de los dos anteriores).*

Ejercicio propuesto: *Crea un programa que emplee recursividad para calcular la suma de los elementos de un vector.*

Ejercicio propuesto: *Crear un programa que encuentre el máximo común divisor de dos números usando el algoritmo de Euclides: Dados dos números enteros positivos m y n, tal que m > n, para encontrar su máximo común divisor, es decir, el mayor entero positivo que divide a ambos: - Dividir m por n para*

obtener el resto r (0 ? r < n) ; - Si r = 0, el MCD es n.; - Si no, el máximo común divisor es MCD(n,r).

Tema 8.9: La sentencia "forward".

La sentencia **"forward"** es necesaria sólo en un caso muy concreto, y cada vez menos, gracias a la programación modular, pero aun así vamos a comentar su uso, por si alguien llega a necesitarla:

Cuando desde un procedimiento A se llama a otro B, este otro procedimiento B debe haberse declarado antes, o el compilador "no lo conocerá". Esto nos supone llevar un poco de cuidado, pero no suele ser problemático.

Eso sí, puede llegar a ocurrir (aunque es muy poco probable) que a su vez, el procedimiento B llame a A, de modo que A debería haberse declarado antes que B, o el compilador protestará porque "no conoce" a A.

En este (poco habitual) caso de que cada uno de los procedimientos exigiría que el otro se hubiese detallado antes, la única solución es decirle que uno de los dos (por ejemplo "A") ya lo detallaremos más adelante, pero que existe. Así ya podemos escribir "B" y después dar los detalles sobre cómo es "A", así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de "Forward"  }
{    FORW.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program forw;

procedure A(n: byte); forward;  { No detallamos cómo es "A", porque
                                  "A" necesita a "B", que el
compilador
                                  aún no conoce; por eso sólo
"declaramos"
                                  "A", e indicamos con "forward" que
                                  más adelante lo detallaremos }

procedure B (n: byte);          { "B" ya puede llamar a "A" }
begin
  writeLn('Entrando a B, con parámetro ', n);
  if n>0 then A (n-1);
end;
```

```
procedure A;                    { Y aquí detallamos "A"; ya no hace
falta }
begin                           { volver a indicar los parámetros }
  writeLn('Entrando a A, con parámetro ', n);
  if n>0 then B (n-1);
end;

begin
  A(10);
end.
```

Nota: en los Pascal actuales, más modulares, si usamos "unidades" (que veremos en el tema 12) se evita la necesidad de emplear "forward", porque primero se dan las "cabeceras" de todos los procedimientos (lo que se conocerá como "interface") y más adelante los detalles de todos ellos (lo que será la "implementation").

Curso de Pascal. Tema 8.10. Funciones matemáticas: abs, sin, cos, arctan, round, trunc, sqr, sqrt, exp, ln, odd, potencias.

La mayoría de las que vamos a ver son funciones matemáticas que están ya predefinidas en Pascal. Muchas de ellas son muy evidentes, pero precisamente por eso no podía dejarlas sin mencionar al menos:

- **Abs**: valor absoluto de un número.
- **Sin**: seno de un cierto ángulo dado en radianes.
- **Cos**: coseno, análogo.
- **ArcTan**: arco tangente. No existe función para la tangente, que podemos calcular como sin(x)/cos(x).
- **Round**: redondea un número real al entero más cercano.
- **Trunc**: trunca los decimales de un número real para convertirlo en entero.
- **Int**: igual que trunc, pero el resultado sigue siendo un número real.
- **Sqr**: eleva un número al cuadrado.
- **Sqrt**: halla la raíz cuadrada de un número.
- **Exp**: exponencial en base e, es decir, eleva un número a e.
- **Ln**: calcula el logaritmo neperiano (base e) de un número.
- **Odd**: devuelve TRUE si un número es impar.
- **Potencias**: no hay forma directa de elevar un número cualquiera a otro en pascal, pero podemos imitarlo con "exp" y "ln", así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Elevar un número real }
{    a otro                }
{    ELEVAR.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program elevar;

function elevado(a,b: real): real;
begin
  elevado := exp(b *ln(a) );
end;

begin
  writeln(elevado(2,3));
end.
```

La deducción de esta fórmula es fácil, conociendo las propiedades de los logaritmos y las exponenciales

```
  a^b = exp ( ln ( a^b )) = exp ( b * ln ( a ))
```

Curso de Pascal. Tema 8.11. Cadenas de texto.

En Turbo Pascal tenemos facilidades para trabajar con cadenas de texto, y desde luego, con más comodidad que desde otros lenguajes como C.

Las funciones **incorporadas** para el manejo de cadenas son:

- **copy**. Devuelve una subcadena. Su formato es function copy(cad: string; posic: integer; cuantos: integer): string; Es decir, le indicamos en qué cadena nos queremos basar, desde qué posición queremos empezar a tomar las letras y cuántas queremos leer. Así, copy('JUAN PEREZ', 1, 4) daría 'JUAN'

- **concat**. Concatena varias cadenas. Su formato es function concat(cad1 [, cad2,..., cadN]: string): string; (lo que hay entre corchetes es opcional). Lo que hace es crear una cadena de texto a partir de varias que va concatenando. `cadena := concat(cad1, cad2, cad3)` es lo mismo que si tecleásemos: `cadena := cad1 + cad2 + cad3`

- **delete**. Borra parte de una cadena. El formato es procedure delete(var cad: string; posic: integer; cuantos: integer): string; y lo que hace es eliminar "cuantos" letras a partir de la posición "posic": si TEXTO es 'JUAN PEREZ', delete(texto, 2, 5) haría que TEXTO pasase a ser 'JEREZ'.

- **insert**. Inserta una subcadena dentro de una cadena. Su formato es procedure Insert(texto: string; var donde: string; posic: integer); donde "texto" es la cadena que queremos insertar, "donde" es la cadena inicial en la que queremos que se inserte, y "posic" es la posición de "donde" en la que se insertará "texto".

- **length**. Dice la longitud de una cadena de texto: function length(cad: string): integer;

- **pos**. Indica la posición de una subcadena dentro de una cadena: function pos(subcadena: string; cadena: string): byte; Por ejemplo, pos('Pérez', 'Juan Pérez') devolvería un 6. Si la subcadena no es parte de la cadena, devuelve 0.

Vamos a ver un ejemplo que las use:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de funciones  }
{    de cadenas            }
{    CADENAS.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{--------------------------}

program cadenas;

var
  frase: string;

begin
  frase := 'Esta es una frase de ejemplo';
  writeln('La primera palabra (letras 1 a 4) es: ', copy(frase, 1, 4)
);
  writeln('Si añadimos más texto: ', concat(frase, ' facilito') );
```

```
   delete(frase, 6, 2);
   writeln('Si borramos la segunda palabra (letras 5 a 7) es: ', frase
);

   insert('si es', frase, 6);
   writeln('Y si insertamos una nueva segunda (y tercera) palabra: ',
frase);
   writeln('La longitud de la frase es: ', length(frase) );
   writeln('Y la primera a parece en la posición: ', pos('a', frase)
);
 end.
```

Que daría como resultado

```
La primera palabra (letras 1 a 4) es: Esta
Si añadimos más texto: Esta es una frase de ejemplo facilito
Si borramos la segunda palabra (letras 5 a 7) es: Esta  una frase de
ejemplo
Y si insertamos una nueva segunda (y tercera) palabra: Esta si es una
frase de ejemplo
La longitud de la frase es: 31
Y la primera a parece en la posición: 4
```

Pero esto tampoco es perfecto, y quien venga de programar en Basic echará de menos construcciones como Left$ (extraer varias letras de la izquierda) o Rigth$ (varias letras de la derecha). También podríamos hacernos alguna funcioncita para convertir a hexadecimal, o de texto a número y al revés, o para manejar la fecha, etc.

La mayoría no son difíciles, así que allá vamos...

(**Nota**: estas rutinas las hice allá por el 93, y en su momento me sirvieron, aunque posiblemente muchas de ellas no estén programadas de la forma más eficiente).

Vamos a crear una **Unit** y a hacer un ejemplo que la use:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Unidad con funciones  }
{    mejoradas para mane-  }
{    jar cadenas           }
{    NSTR.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
```

```
{     - Tmt Pascal Lt 1.20   }
{--------------------------}

unit nSTR;

{=====================}
{                     }
{    Unidad nSTR      }
{  Manejo de cadenas  }
{                     }
{  Nacho Cabanes, 93  }
{                     }
{=====================}

interface

  type
    TipoFecha = string[8];
  FUNCTION CADENA(const carc:char;            {Cadena de caracteres
repetidos}
    const veces:byte):string;
  FUNCTION COMAS(var s:string): string;        {Cambia , por . para
hallar valor}
  FUNCTION CONPUNTO(const num:longint):string;  {Nº como cadena con
puntos: x.xxx.xxx}
  FUNCTION FECHA(const separ:char):tipofecha;  {Fecha, indicando el
separador}
  FUNCTION HEX(const a:byte):string;           {Pasa byte a
hexadecimal}
  FUNCTION LEFT(const cad:string;              {Letras de la
izquierda}
    const n:byte):string;
  FUNCTION MAYUSC(const cad:string):string;    {Convierte a
mayúsculas}
  FUNCTION RIGHT(const cad:string;             {Letras de la derecha}
    const n:byte):string;
  FUNCTION SPC(const n:byte):string;           {Espacios en blanco}
  FUNCTION STRS(const valor:longint):string;   {Como STR$ de Basic}
  FUNCTION STRSR(const valor:real):string;     {STRS$ para reales}
  FUNCTION VALORLI(const cad:string):longint;  {Valor de un string
como LongInt}
  FUNCTION VALORR(const cad:string):real;      {Valor de un string
como REAL}
  FUNCTION VALORW(const cad:string):word;      {Valor de un string
como WORD}

implementation

  uses dos;

  FUNCTION CADENA(const carc:char; const veces:byte):string;
  { Crea una cadena formada por un mismo carácter repetido varias
veces }
  var
    cadtmp: string;
    i: byte;
  begin
    cadtmp := '';                                 { Cadena vacía }
    for i := 1 to veces do cadtmp[i] := carc;    { Voy dando valores }
    cadtmp[0] := chr(veces);                      { Ajusto la longitud }
    { Las dos líneas anteriores se podrían
```

```
    reemplazar también por:
    for i := 1 to veces do cadtmp := cadtmp + carc;
    }
    cadena := cadtmp                            { Valor definitivo }
  end;

  FUNCTION COMAS(var s:string): string;
  { Cambia comas por puntos (decimales en formato americano) para
poder
    introducir un número en formato español y hallar su valor}
  var
    i: byte;
    t: string;
  begin
    t := s;                               { Copio la cadena inicial }
    for i :=1 to length(t) do             { La recorro }
      if t[i]=',' then t[i] := '.';       { Y si encuentro , pongo . }
    comas := t;
  end;

  FUNCTION CONPUNTO(const num:longint):string;
  { Devuelve un número como cadena con puntos: x.xxx.xxx }
  var
    cadena: string;
    d: byte;
  const
    punto: char = '.';
  begin
    str(abs(num),cadena);                 { Empiezo a trabajar sin signo
}
    d := length(cadena);                  { Veo la longitud }
    if d>3 then                           { Si es mayor de 1.000 }
      insert(punto,cadena,d-2);           { inserto un punto }
    d := length(cadena);                  { Vuelvo a mirar }
    if d>7 then                           { Si es mayor de 1.000.000 }
      insert(punto,cadena,d-6);           { pongo otro punto }
    if num<0 then cadena:='-'+cadena;     { Si es negativo, le pongo
signo }
    ConPunto := cadena;
  end;

  FUNCTION FECHA(const separ:char):tipofecha;
  { Para ver la fecha como DD/MM/AA, o DD-MM-AA, etc. }
  var
    dia,mes,ano,disem: word;             { Fecha leida del DOS }
    temporal: string[8];                 { Temporal, para valor final }
    tempo2: string[2];                   { Temporal, para cada dato }
  begin
    getdate(ano,mes,dia,disem);          { Leo la fecha del DOS }
    str(dia,tempo2);                     { Paso el día a cadena }
    temporal:=tempo2+separ;              { Y le añado el separador }
    str(mes,tempo2);                     { Paso el mes }
    temporal:=temporal+tempo2+separ;     { Y lo añado }
    str(ano mod 100,tempo2);             { Paso el año (dos ultimas
cifras) }
    temporal:=temporal+tempo2;           { Lo añado }
    fecha:=temporal;                     { Y se acabó }
  end;
  FUNCTION HEX(const a:byte):string;
  { Convierte un byte a hexadecimal }
  const
```

```
    cadena: string[16] = '0123456789ABCDEF';
  begin
    hex := cadena[a div 16 + 1]        { La primera letra: dividir
entre 16 }
      + cadena[a mod 16 + 1];          { La segunda: resto de la
división }
  end;

  FUNCTION LEFT(const cad:string; const n:byte):string;
  { Cadena formada por los n caracteres más a la izquierda }
  var
    temp: string;
  begin
    If n > length(cad) then          { Si piden más que la longitud }
      temp := cad                    { La cojo entera }
    else
      temp := copy(cad,1,n);         { Si no, una subcadena }
    Left := temp;
  end;

  FUNCTION MAYUSC(const cad:string):string;
  { Convierte una cadena a mayúsculas.  La forma de tratar los
caracteres
    internacionales no es la más adecuada, porque pueden variar en las
    distintas páginas de códigos, pero sirva al menos como ejemplo de
    cómo comprobar si un valor está entre varios dados  O:-) }
  var
    buc: byte;
    cad2: string;
  begin
    cad2 := cad;
    for buc := 1 to length(cad2) do
      begin
      if cad2[buc] in ['a'..'z'] then           { Letras "normales" }
        cad2[buc] := upcase(cad2[buc]);
      { Internacionales: esto es lo que puede dar problemas }
      if cad2[buc] in ['á','à','ä','â','Ä'] then cad2[buc]:='A';
      if cad2[buc] in ['é','è','ë','ê','É'] then cad2[buc]:='E';
      if cad2[buc] in ['í','ì','ï','î'] then cad2[buc]:='I';
      if cad2[buc] in ['ó','ò','ö','ô','Ö'] then cad2[buc]:='O';
      if cad2[buc] in ['ú','ù','ü','û','Ü'] then cad2[buc]:='U';
      if cad2[buc] = 'ñ' then cad2[buc]:='Ñ'
      end;
      mayusc := cad2                             { Valor definitivo }
  end;

  FUNCTION RIGHT(const cad:string; const n:byte):string;
  { Cadena formada por los n caracteres más a la derecha }
  var
    temp: string;
  begin
    If n > length(cad) then                    { Si piden más que la
longitud }
      temp := cad                              { La cojo entera }
    else
      temp := copy(cad, length(cad)-n+1, n); { Si no, subcadena }
    right := temp;
  end;

  FUNCTION SPC(const n: byte):string;
  { Cadena formada por n espacios en blanco }
```

```
var
  t: string;
  i: byte;
begin
  t := ' ';                              { Al menos uno }
  for i := 2 to n do t:=t+' ';           { Si hay dos o más, añado }
  spc := t
end;

FUNCTION STRS(const valor:longint):string;
{ Simplemente porque me gusta más usar STR como función que como
  procedimiento (recuerdos del Basic, supongo) }
var
  temporal: string;
begin
  str(valor,temporal);                  { Llamo al procedimiento STR }
  strs := temporal;
end;

FUNCTION STRSR(const valor:real):string;
{ Igual que STRS, pero para números reales (deja 2 decimales) }
var
  temporal: string;
begin
  str(valor:3:2,temporal);
  strsr := temporal;
end;

FUNCTION VALORW(const cad:string):word;
{ Valor de un string como WORD, para evitar usar VAL como
procedimiento
    que es más potente pero también más incómodo a veces
(posiblemente,
    esto también está aquí por culpa del Basic ;-) ) }
var
  uno:word;
  code:integer;
begin
  val(cad,uno,code);
  valorw:=uno;
end;

FUNCTION VALORLI(const cad:string):longint;
{ Igual que VALORW, pero para LongInt }
var
  uno:longint;
  code:integer;
begin
  val(cad,uno,code);
  valorli:=uno;
end;

FUNCTION VALORR(const cad:string):real;
{ Igual que VALORW, pero para reales.  Primero cambia las , por . }
var
  uno:real;
  code:integer;
  cad2:string;
begin
  cad2:=cad;
```

```
    while Pos(',', cad2) > 0 do                 { Le cambio , por . (si
hay) }
      cad2[Pos(',', cad2)]:= '.';
    val(cad2,uno,code);                         { Y llamo al
procedimiento }
    valorr:=uno;
  end;

end.
```

Un ejemplo de programa que usaría estas funciones sería:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Prueba de la unidad   }
{    nStr                  }
{    USASTR.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program UsaStr;  { Ejemplo de la unidad nStr }

uses
  crt, nSTR;

var
  texto: string;
  numero1: longint;
  numero2: real;

begin
  texto := 'Este será el texto inicial.';
  numero1  := 1234567;
  numero2  := 1234567.895;
  clrscr;
  writeln('  ----- Prueba de la Unidad nSTR, día '+ fecha('-')+ ' ----
-');
  writeln;
  writeln( texto );
  writeln('Convertido a mayúsculas: ', mayusc(texto) );
  writeln('Sus 8 primeras letras: ', left(texto,8) );
  writeln('Las 7 últimas: ', right(texto,7) );
  writeln('La 3ª y las 5 siguientes: ', copy(texto,3,5) );
  writeln;
  writeln('Siete espacios y 10 X: ', spc(7), cadena('X',10), '.');
  writeln;
  writeln('El primer número es: ', numero1);
  writeln('Si le ponemos puntos separadores: ', ConPunto(numero1) );
  writeln('Las dos últimas cifras son: ', numero1 mod 100,
    ', que en hexadecimal es: $', hex(numero1 mod 100) );
```

```
  writeln('La segunda y la tercera cifra, como cadena: ',
    copy(strs(numero1),2,2) );
  writeln;
  writeln('El segundo número: ', numero2 );
  writeln('En notación "normal": ', numero2:10:3 );
  writeln('Como cadena: ', strsr(numero2) );
  writeln('El valor de "123,5" es: ', valorr('123,5') :6:1 );
end.
```

Que mostraría algo como

```
  ----- Prueba de la Unidad nSTR, día 16-6-10 -----

Este será el texto inicial.
Convertido a mayúsculas: ESTE SERA EL TEXTO INICIAL.
Sus 8 primeras letras: Este ser
Las 7 últimas: nicial.
La 3ª y las 5 siguientes: te se

Siete espacios y 10 X:        XXXXXXXXXX.

El primer número es: 1234567
Si le ponemos puntos separadores: 1.234.567
Las dos últimas cifras son: 67, que en hexadecimal es: $43
La segunda y la tercera cifra, como cadena: 23

El segundo número:  1.234567895000000E+006
En notación "normal": 1234567.895
Como cadena: 1234567.90
El valor de "123,5" es:  123.5
```

Tema 9: Otros tipos de datos.

Comenzamos a ver los tipos de datos que podíamos manejar en el tema 2. En aquel momento tratamos los siguientes:

- Byte. Entero, 0 a 255. Ocupa 1 byte de memoria.
- Integer. Entero con signo, -32768 a 32767. Ocupa 2 bytes.
- Char. Carácter, 1 byte.
- String[n]. Cadena de n caracteres (hasta 255). Ocupa n+1 bytes.
- Real. Real con signo. De 2.9e-39 a 1.7e38, 11 o 12 dígitos significativos, ocupa 6 bytes.
- Boolean. TRUE o FALSE.
- Array. Vectores o matrices.
- Record. Con campos de distinto tamaño.

Pues esta vez vamos a ampliar con otros tipos de datos. :

- Enteros.
- Correspondencia byte-char.
- Reales del 8087.
- Tipos enumerados.
- Más detalles sobre Strings.
- Registros variantes.
- Conjuntos.

Vamos allá:

Comencemos por los demás tipos de **números enteros**. Estos son:

- **Shortint**. Entero con signo, de -128 a 127, ocupa 1 byte.
- **Word**. Entero sin signo, de 0 a 65535. Ocupa 2 bytes.
- **Longint**. Con signo, de -2147483648..2147483647. Ocupa 4 bytes.

Estos tipos, junto con "char" (y "boolean" y otros para los que no vamos a entrar en tanto detalle) son tipos **ordinales**, existe una relación de orden entre ellos y cada elemento está precedido y seguido por otro que podemos conocer (cosa que no ocurre en los reales). Para ellos están definidas las funciones:

- **pred** - Predecesor de un elemento : pred(3) = 2
- **succ** - Sucesor: succ(3) = 4
- **ord** - Número de orden (posición) dentro de todo el conjunto.

El uso más habitual de "ord" es para convertir de "**char**" a "**byte**". Los caracteres se almacenan en memoria de tal forma que a cada uno se le asigna un número entre 0 y 255, su "código ASCII" (ej: A=65, a=96, 0=48, _=178). La forma de hallar el código ASCII de una letra es simplemente ord(letra), como ord('_').

El paso contrario, la letra que corresponde a cierto número, se hace con la función "**chr**". Así, podemos escribir los caracteres "imprimibles" de nuestro ordenador sabiendo que van del 32 al 255 (los que están por debajo de 32 suelen ser caracteres de control, que en muchos casos no se podrán mostrar en pantalla; en algunos sistemas ocurre lo mismo con los caracteres que están por encima del 127):

```
var
  bucle: byte;
begin
  for bucle := 32 to 255 do
    write( chr(bucle) );
end.
```

Si tenemos **coprocesador** matemático, podemos utilizar también los siguientes tipos de números reales del 8087:

Nombre	Rango	Dígitos	Bytes
single	1.5e-45 a 3.4e38	7-8	4
double	5.0e-324 a 1.7e308	15-16	8
extended	3.4e-4932 a 1.1e4932	19-20	10
comp	-9.2e18 a 9.2e18	19-20	8

El tipo "comp" es un entero (no real, sin decimales) de 8 bytes, que sólo tenemos disponible si usamos el coprocesador.

Nosotros podemos crear nuestros propios tipos de **datos enumerados**:

```
type DiasSemana = (Lunes, Martes, Miercoles, Jueves, Viernes,
  Sabado, Domingo);
```

Declaramos las variables igual que hacíamos con cualquier otro tipo:

```
var dia: DiasSemana;
```

Y las empleamos como siempre: podemos darles un valor, utilizarlas en comparaciones, etc.

```
begin
  dia := Lunes;
  [...]

  if dia = Viernes then
    writeln( 'Se acerca el fin de semana!' );
  [...]
```

Los tipos enumerados también son tipos ordinales, por lo que podemos usar pred, succ y ord con ellos. Así, con los datos del ejemplo anterior tendríamos

```
pred(Martes) = Lunes, succ(Martes) = Miercoles, ord(Martes) = 1
```

(el número de orden de Martes es 1, porque es el segundo elemento, y se empieza a numerar en cero).

Nota: también podemos definir los valores que puede tomar una variable, indicándolo en forma de **subrango**, al igual que se hace con los índices de un array:

```
var
  dia: 1..31;
```

Volvamos a los **strings**. Habíamos visto que se declaraban de la forma "string[n]" y ocupaban n+1 bytes (si escribimos sólo "string", es válido en las últimas versiones de Pascal y equivale a "string[255]"). ¿Por qué n+1 bytes? Pues porque también se guarda la longitud de la cadena.

Ya que estamos, vamos a ver cómo acceder a caracteres individuales de una cadena. Nada más fácil. A lo mejor a alguien la definición anterior, indicando el tamaño entre corchetes le recuerda a la de un Array. Así es. De hecho, la definición original en Pascal del tipo String[x] era "Packed Array[1..x] of char" ("**packed**" era para que el compilador intentase "empaquetar" el array, de modo que ocupase menos; esto no es necesario en Turbo Pascal). Así, con nombre[1] accederíamos a la primera letra del nombre, con nombre[2] a la segunda, y así sucesivamente.

Una última curiosidad: habíamos dicho que se guarda también la longitud de la cadena. ¿Donde? Pues en la posición 0. Va programita de ejemplo:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Longitud de una       }
{    cadena; posiciones    }
{    POSCAD.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
```

```
{     - Surpas 1.00          }
{--------------------------}

program PosCad;

var
  linea: string [20];     (* Cadena inicial: limitada a 20 letras *)
  pos: byte;                       (* Posición que estamos mirando *)

begin
  writeln( 'Introduce una línea de texto...' );
  readln( linea );
  for pos := 1 to ord(linea[0]) do
    writeln(' La letra número ', pos,' es una ', linea[pos]);
end.
```

Comentarios:

- "linea[0]" da la longitud de la cadena, pero es un carácter, luego debemos convertirlo a byte con "ord".
- Entonces, recorremos la cadena desde la primera letra hasta la última.
- Si tecleamos más de 20 letras, las restantes se desprecian.

Tema 9.2: Otros tipos de datos (2).

También habíamos visto ya los registros (records), pero con unos campos fijos. No tiene por qué ser necesariamente así. Tenemos a nuestra disposición los **registros variantes**, en los que con un "**case**" podemos elegir unos campos u otros. La mejor forma de entenderlos es con un ejemplo.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Registros variantes   }
{    REGVAR.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{    - Tmt Pascal Lt 1.01  }
{--------------------------}

program RegistrosVariantes;

type
  TipoDato   = (Num, Fech, Str);
```

```
   Fecha    = record
      D, M, A: Byte;
   end;
   Ficha = record
     Nombre: string[20];            (* Campo fijo *)
     case Tipo: TipoDato of         (* Campos variantes *)
       Num:  (N: real);                 (* Si es un número: campo N *)
       Fech: (F: Fecha);                (* Si es fecha: campo F *)
       Str:  (S: string[30]);           (* Si es string: campo S *)
   end;

 var
   UnDato: Ficha;

 begin
   UnDato.Nombre := 'Nacho';      (* Campo normal de un record *)
   UnDato.Tipo   := Num;          (* Vamos a almacenar un número *)
   UnDato.N      := 3.5;          (* que vale 3.5 *)
   Writeln('Ahora el tipo es numérico, y el nombre es ',
     UnDato.Nombre, '.');
   Writeln('El campo N vale: ', UnDato.N);
   UnDato.Nombre := 'Nacho2';     (* Campo normal *)
   UnDato.Tipo   := Fech;         (* Ahora almacenamos una fecha *)
   UnDato.F.D    := 7;            (* Día: 7 *)
   UnDato.F.M    := 11;           (* Mes: 11 *)
   UnDato.F.A    := 93;           (* Año: 93 *)
   Writeln('Ahora el tipo es Fecha, y el nombre es ',
     UnDato.Nombre, '.');
   Writeln('El campo F.D (día) vale: ', UnDato.F.D);
   UnDato.Nombre := 'Nacho3';      (* Campo normal *)
   UnDato.Tipo   := Str;           (* Ahora un string *)
   UnDato.S      := 'Nada';        (* el texto "Nada" *)
   Writeln('Ahora el tipo es string, y el nombre es ',
     UnDato.Nombre, '.');
   Writeln('El campo S vale: ', UnDato.S);
 end.
```

Tema 9.3: Otros tipos de datos (3).

Finalmente, tenemos los **conjuntos** (sets). Un conjunto está formado por una serie de elementos de un tipo base, que debe ser un ordinal de no más de 256 valores posibles, como un "char", un "byte" o un enumerado.

```
type
  Letras = set of Char;

type DiasSemana = (Lunes, Martes, Miercoles, Jueves, Viernes,
  Sabado, Domingo);

Dias = set of DiasSemana;
```

Para construir un "**set**" utilizaremos los corchetes ([]), y dentro de ellos enumeramos los valores posibles, uno a uno o como rangos de valores separados por ".." :

```
var
  LetrasValidas : Letras;
  Fiesta : Dias;
begin
  LetrasValidas = ['a'..'z', 'A'..'z', '0'..'9', 'ñ', 'Ñ'];
  Fiesta = [ Sabado, Domingo ];
end.
```

Un conjunto vacío se define con [ ].

Las **operaciones** que tenemos definidas sobre los conjuntos son:

Operac	Nombre
+	Unión
-	Diferencia
*	Intersección
in	Pertenencia

Así, podríamos hacer cosas como

```
VocalesPermitidas := LetrasValidas * Vocales;

if DiaActual in Fiesta then
  writeln( 'No me dirás que estás trabajando... ;-D ' );
```

En el primer ejemplo hemos dicho que el conjunto de vocales permitidas (que deberíamos haber declarado) es la intersección de las vocales (que también debíamos haber declarado) y las letras válidas.

En el segundo, hemos comprobado si la fecha actual (que sería de tipo DiasSemana) pertenece al conjunto de los días de fiesta.

Vamos a ver un ejemplito sencillo "que funcione":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Conjuntos (sets)      }
{    EJSET.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - FreePascal 2.0.2    }
{--------------------------}

program EjSet;
```

```
var
  minusculasValidas,
    mayusculasValidas,
    letrasValidas: set of char;
  letra: char;

begin
  minusculasValidas := ['a', 'b', 'c', 'd'];
  mayusculasValidas := ['F', 'H', 'K', 'M'];
  letrasValidas := minusculasValidas
    + mayusculasValidas;
  repeat
    writeln( 'Introduce una letra...' );
    readln( letra );
    if not (letra in letrasValidas) then
      writeln('No aceptada!');
  until letra in letrasValidas;
end.
```

Pues eso es todo por hoy. Si es denso (eso creo), pues a releerlo un par de veces y a experimentar...

No propongo ejercicios porque es un tema árido y habrá gente que no use casi nada de esto. Quien le vea la utilidad, que se los proponga él sólo según la aplicación que les quiera dar. De todas formas, a medida que vayamos haciendo programas más grandes, iremos usando más de una de las cosas que hemos visto hoy.

Curso de Pascal. Tema 10: Pantalla en modo texto.

En este tema vamos a ver cómo acceder a la pantalla de texto, y algunos "bonus". :-)

Este tema va a ser específico de **Turbo Pascal para DOS**. Algunas de las cosas que veremos aparecen en otras versiones de Turbo Pascal (la 3.0 para CP/M, por ejemplo), pero no todas, y de cualquier modo, nada de esto es Pascal estándar. Aun así, como muchas versiones de Pascal posteriores (Tmt Pascal, Virtual Pascal, **Free Pascal**, etc) buscan una cierta compatibilidad con Turbo Pascal, es fácil que funcione con muchos compiladores recientes.

Me centraré primero en cómo se haría con las versiones **5.0 y superiores** de Turbo Pascal. Luego comentaré cómo se haría con Turbo Pascal 3.01.

Vamos allá... En la mayoría de los lenguajes de programación, existen "bibliotecas" (en inglés, "library") con funciones y procedimientos nuevos, que

permiten ampliar el lenguaje. En Turbo Pascal, estas bibliotecas reciben el nombre de "**unidades**" (UNIT), y existen a partir de la versión 5.

Veremos cómo crearlas un poco más adelante, pero de momento nos va a interesar saber cómo acceder a ellas, porque "de fábrica" ;-) Turbo Pascal incorpora unidades que aportan mayores posibilidades de manejo de la pantalla en modo texto o gráfico, de acceso a funciones del DOS, de manejo de la impresora, etc.

Así, la primera unidad que trataremos es la encargada de gestionar (entre otras cosas) la pantalla en modo texto. Esta unidad se llama **CRT**.

Para acceder a cualquier unidad, se emplea la sentencia "**uses**" justo después de "program" y antes de las declaraciones de variables:

```
program prueba;

uses crt;
var
[...]
```

Voy a mencionar algunos de los procedimientos y funciones más importantes. Al final de esta lección resumo todos los que hay y para qué sirven.

- **ClrScr** : Borra la pantalla.
- **GotoXY (x, y)** : Coloca el cursor en unas coordenadas de la pantalla.
- **TextColor (Color)** : Cambia el color de primer plano.
- **TextBackground (Color)** : Cambia el color de fondo.
- **WhereX** : Función que informa de la coordenada x actual del cursor.
- **WhereY** : Coordenada y del cursor.
- **Window (x1, y1, x2, y2)** : Define una ventana de texto.

Algunos "extras" no relacionados con la pantalla son:

- **Delay(ms)** : Espera un cierto número de milisegundos.
- **ReadKey** : Función que devuelve el carácter que se haya pulsado.
- **KeyPressed** : Función que devuelve TRUE si se ha pulsado alguna tecla.
- **Sound (Hz)** : Empieza a generar un sonido de una cierta frecuencia.
- **NoSound**: Deja de producir el sonido.

Comentarios generales, la mayoría "triviales" :

- X es la columna, de 1 a 80.
- Y es la fila, de 1 a 25.

- El cursor es el cuadrado o raya parpadeante que nos indica donde seguiríamos escribiendo.
- Los colores están definidos como constantes con el nombre en inglés. Así Black = 0, de modo que TextColor (Black) es lo mismo que TextColor(0).
- La pantalla se puede manejar también accediendo directamente a la memoria de video, pero eso lo voy a dejar, al menos por ahora...

Aquí va un programita de ejemplo que maneja todo esto... o casi

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de la unidad  }
{    CRT: acceso a panta-  }
{    lla en modo texto TP  }
{    EJCRT.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.01  }
{--------------------------}

program PruebaDeCRT;

uses crt;

var
  bucle : byte;
  tecla : char;

begin
  ClrScr;                         { Borra la pantalla }
  TextColor( Yellow );            { Color amarillo }
  TextBackground( Red );          { Fondo rojo }
  GotoXY( 40, 13 );               { Vamos al centro de la pantalla }
  Write(' Hola ');                { Saludamos ;-) }
  Delay( 1000 );                  { Esperamos un segundo }
  Window ( 1, 15, 80, 23 );       { Ventana entre las filas 15 y 23 }
  TextBackground ( Blue );        { Con fondo azul }
  ClrScr;                         { La borramos para que se vea }
  for bucle := 1 to 100
    do WriteLn( bucle );          { Escribimos del 1 al 100 }
  WriteLn( 'Pulse una tecla..');
  tecla := ReadKey;               { Esperamos que se pulse una tecla }
  Window( 1, 1, 80, 25 );         { Restauramos ventana original }
  GotoXY( 1, 24 );                { Vamos a la penúltima línea }
  Write( 'Ha pulsado ', tecla );  { Pos eso }
  Sound( 220 );                   { Sonido de frecuencia 220 Hz }
  Delay( 500 );                   { Durante medio segundo }
  NoSound;                        { Se acabó el sonido }
  Delay( 2000 );                  { Pausa antes de acabar }
  TextColor( LightGray );         { Colores por defecto del DOS }
```

```
  TextBackground( Black );        { Y borramos la pantalla }
  ClrScr;
end.
```

Finalmente, veamos los cambios para **Turbo Pascal 3.01**: En TP3 no existen unidades, por lo que la línea "uses crt;" no existiría. La otra diferencia es que para leer una tecla se hace con "**read(kbd, tecla);**" (leer de un dispositivio especial, el teclado, denotado con kbd) en vez de con "tecla := readkey". Con estos dos cambios, el programa anterior funciona perfectamente.

Tema 10.2: Pantalla en modo texto con Surpas.

Para Surpas la cosa cambia un poco:

- GotoXY empieza a contar desde 0.
- No existen ClrScr, TextColor ni TextBackground (entre otros), que se pueden emular como he hecho en el próximo ejemplo.
- No existen Window, Delay, Sound, y no son tan fáciles de crear como los anteriores.

Con estas consideraciones, el programa (que aun así se parece) queda:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Pantalla de texto     }
{    con Surpas            }
{    EJCRTSP.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Surpas 1.00         }
{--------------------------}

program PruebaDeCRT;

{ ----- Aqui empiezan las definiciones que hacen que SurPas
        maneje la pantalla de forma similar a Turbo Pascal  ------ }
        { Para comprender de donde ha salido esto, consulta el
          fichero IBMPC.DOC que acompaña a SURPAS }

const
  Black       =  0;
  Blue        =  1;
  Green       =  2;
  Cyan        =  3;
  Red         =  4;
  Magenta     =  5;
```

```
  Brown        =  6;
  LightGray    =  7;
  DarkGray     =  8;
  LightBlue    =  9;
  LightGreen   = 10;
  LightCyan    = 11;
  LightRed     = 12;
  LightMagenta= 13;
  Yellow       = 14;
  White        = 15;

procedure ClrScr;
begin
  gotoxy(0,0);
  write( CLREOS );
end;

procedure TextColor(n: byte);
begin
  write( chr(27), 'b', chr(n) );
end;

procedure TextBackground(n: byte);
begin
  write( chr(27), 'c', chr(n) );
end;

{ Se podrían añadir otras posibilidades, como TextMode, HighVideo y
  LowVideo, etc, siguiendo este esquema, si se cree necesario }

{ ----- Final del añadido ----- }
var
  bucle : byte;
  tecla : char;
begin
  ClrScr;                          { Borra la pantalla }
  TextColor( Yellow );             { Color amarillo }
  TextBackground( Red );           { Fondo rojo }
  GotoXY( 30, 13 );                { Vamos al centro de la pantalla }
  Write(' Hola. Pulse una tecla... ');                    { Saludamos ;-)
}
  read(kbd,tecla);                 { Esperamos que se pulse una tecla }
  TextBackground ( Blue );         { Con fondo azul }
  ClrScr;                          { La borramos para que se vea }
  for bucle := 1 to 100
    do Write( bucle, ' ' );        { Escribimos del 1 al 100 }
  WriteLn;                         { Avanzamos una línea }
  WriteLn( 'Pulse otra tecla..');
  read(kbd,tecla);                 { Esperamos que se pulse una tecla }
  GotoXY( 0, 12 );                 { Vamos al centro de la pantalla }
  Write( 'Ha pulsado ', tecla );   { Pos eso }
  TextColor( LightGray );          { Colores por defecto del DOS }
  TextBackground( Black );         { Y borramos la pantalla }
  GotoXY( 0, 23 );                 { Vamos a la penúltima línea }
end.
```

Tema 10.3: Procedimientos y funciones en CRT.

Finalmente, y por eso de que lo prometido es deuda, va un resumen de los **procedimientos** y **funciones** que tenemos disponibles en la unidad **CRT** (comprobado con Turbo Pascal 7.0):

- AssignCrt Asocia un fichero de texto con la pantalla, de modo que podríamos usar órdenes como write(output, 'Hola'),
  más cercanas al Pascal "original". Como creo que nadie la use (ni yo), no cuento más.
- ClrEol Borra desde la posición actual hasta el final de la línea.
- ClrScr Borra la pantalla y deja el cursor al comienzo de ésta (en la esquina superior izquierda).
- Delay Espera un cierto número de milisegundos.
- DelLine Borra la línea que contiene el cursor.
- GotoXY Mueve el cursor a una cierta posición de la pantalla.
- HighVideo Modo de "alta intensidad". Es casi un "recuerdo" de cuando las pantallas monocromas no podían mostrar colores (ni siquiera varios tonos de gris) y sólo podíamos usar dos tonos: "normal" o "intenso" (si alguien conserva una
  tarjeta gráfica Hercules sabrá a qué me refiero, y con una VGA basta con escribir MODE MONO desde el DOS).
- InsLine Inserta una línea en la posicón del cursor.
- KeyPressed Dice si se ha pulsado una tecla. El valor de esta tecla se puede comprobar después con ReadKey.
- LowVideo Texto de "baja intensidad" (ver HighVideo).
- NormVideo Texto de "intensidad normal" (la que tuviera el carácter del cursor al comenzar el programa).
- NoSound Para el sonido del altavoz interno.
- ReadKey Lee un carácter de el teclado. Si no se ha pulsado ninguna tecla, espera a que se pulse.
- Sound Hace que el altavoz interno comience a producir un sonido. La duración se controlará con Delay o algún método similar, y el sonido se debe parar con NoSound.
- TextBackground Elige el color de fondo.
- TextColor Fija el color de primer plano.
- TextMode Cambia a un cierto modo de pantalla.
- WhereX Devuelve la posición X en la que se encuentra el cursor.
- WhereY Posición Y en la que se encuentra.
- Window Define una ventana de texto.

También tenemos las **variables** siguientes (cuyo valor podemos leer o cambiar):

- CheckBreak Boolean. Indica si puede interrumpir el programa pulsando Ctrl+C ó Ctrl+Break.
- CheckEOF Boolean. Muestra un carácter de fin de fichero o no al pulsar Ctrl+Z.

- DirectVideo Boolean. Escritura directa a video o no. Si el valor es True (por defecto), Write y WriteLn escriben en la memoria de pantalla. Si es False, las llamadas utilizan servicios de la Bios, más lentos. La pregunta es "¿y quien quiere lentitud? ¿para qué vamos a ponerlo a False?" La respuesta es que a través de los servicios de la Bios podemos usar Write para escribir también en modo gráfico. Se verá un ejemplo del uso de DirectVideo más adelante, en la Ampliación 2 ("Gráficos sin BGI").
- CheckSnow Boolean. Cuando se escribe directamente en memoria de pantalla con una tarjeta CGA antigua puede aparecer "nieve". Si es nuestro caso, o puede darse en alguien para quien estemos haciendo el programa, deberemos
  añadir CheckSnow := True.
- LastMode Word. Guarda el modo de pantalla activo cuando comenzó el programa.
- TextAttr Byte. Atributos (colores) actuales del texto.
- WindMin Word.
- WindMax Word. Coordenadas mínimas y máximas de la ventana actual. Cada word son dos bytes: el byte bajo devuelve la coordenada X (p .: lo(WindMin)) y el alto la Y (p .: hi(WindMin)).

Y como **constantes** (por ejemplo, para poder escribir el nombre de un color en vez de recordar qué número indica ese color, como hemos hecho en los ejemplos anteriores):

- Black = 0
- Blue = 1
- Green = 2
- Cyan = 3
- Red = 4
- Magenta = 5
- Brown = 6
- LightGray = 7
- DarkGray = 8
- LightBlue = 9
- LightGreen = 10
- LightCyan = 11
- LightRed = 12
- LightMagenta= 13
- Yellow = 14
- White = 15

Las constantes que indican los modos de pantalla son:

- BW40 = 0 Blanco y negro 40x25 en CGA o superiores.
- CO40 = 1 Color 40x25 en CGA o superiores.
- BW80 = 2 Blanco y negro 80x25 en CGA o superiores.

- CO80 = 3 Color 40x25 en CGA o superiores.
- Mono = 7 Monocromo 80x25 en MDA, Hercules, EGA Mono o VGA Mono.
- Font8x8 = 256 Modo de 43 o 50 líneas en EGA o VGA.

Y por compatibilidad con la versión 3.0:

- C40 = CO40
- C80 = CO80

Pues hala, a experimentar...

Tema 10.4: Ejemplo: juego del ahorcado.

Antes de dejar el tema, un ejemplo sencillo que ponga a prueba algunas de las cosas que hemos visto: vamos a hacer el juego del **ahorcado**:

- Un primer jugador deberá introducir una frase. La pantalla se borrará, y en lugar de cada letra aparecerá un guión.
- El segundo jugador deberá ir tecleando letras. Si falla, ha gastado uno de sus intentos. Si acierta, la letra acertada deberá aparecer en las posiciones en que se encuentre.
- El juego acaba cuando se aciertan todas las letras o se acaban los intentos.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Juego del Ahorcado    }
{    AHORCA.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

Program Ahorcado;

Uses crt;

Var
  palabra, intento, letras:string; { La palabra a adivinar, la que }
```

```
                                   {   el jugador 2 va consiguiendo y
}
                                   {   las letras que se han probado }
  oportunidades: integer;          { El número de intentos permitido }
  letra: char;                     { Cada letra que prueba el jug. dos
}
  i: integer;                      { Para mirar cada letra, con "for"
}
  acertado: boolean;               { Si ha acertado alguna letra }


begin
  clrscr;                        { Valores iniciales, del jugador 1 }
  gotoxy (10,5);
  write ('Jugador 1:  ¿Que frase hay que adivinar?  ');
  readln (palabra);
  gotoxy (10,7);
  write ('¿En cuantos intentos?  ');
  readln (oportunidades);

  intento := '';                 { Relleno con _ y " " lo que ve Jug.
2 }
  for i:=1 to length(palabra) do
    if palabra[i]= ' ' then
      intento:=intento+' '
    else
      intento:=intento+'_';

  repeat
    clrscr;
    gotoxy (10,6);               { Digo cuantos intentos le quedan }
    writeln('Te quedan ',oportunidades,' intentos');

    gotoxy (10,8);               { Le muestro cómo va }
    writeln(intento);

    gotoxy (10,10);              { Y le pido otra letra }
    write('Letras intentadas: ', letras);

    gotoxy (10,12);              { Y le pido otra letra }
    write('¿Qué letra? ');
    letra := readkey;
    letras := letras + letra;

    acertado := false;           { Miro a ver si ha acertado }
    for i:=1 to length(palabra) do
      if letra=palabra[i] then
        begin
        intento[i]:=palabra[i];
        acertado := true;
        end;

    if acertado = false then     { Si falló, le queda un intento menos
}
      oportunidades := oportunidades-1;

  until (intento=palabra)        { Hasta que acierte }
    or (oportunidades=0);        { o gaste sus oportunidades }

  gotoxy(10, 15);                { Le felicito o le digo cual era }
  if intento=palabra then
```

```
    writeln('¡Acertaste!')
  else
    writeln('Lo siento.  Era: ', palabra);
end.
```

Esto es muy mejorable. La primera mejora será que no haya necesidad de que un primer jugador sea el que escriba la palabra a adivinar y el número de intentos, sino que el número de intentos esté prefijado en el programa, y exista una serie de palabras de las que el ordenador escoja una al azar (para lo que usaremos "random" y "randomize", que se ven con más detalle en la Ampliación 1):

```
 {--------------------------}
 {  Ejemplo en Pascal:      }
 {                          }
 {    Juego del Ahorcado    }
 {    (segunda version)     }
 {    AHORCA2.PAS           }
 {                          }
 {  Este fuente procede de  }
 {  CUPAS, curso de Pascal  }
 {  por Nacho Cabanes       }
 {                          }
 {  Comprobado con:         }
 {    - Free Pascal 2.2.0w  }
 {    - Turbo Pascal 7.0    }
 {    - Tmt Pascal Lt 1.20  }
 {--------------------------}

 Program Ahorcado2;

 Uses crt;

 Const
   NUMPALABRAS = 10;
   MAXINTENTOS = 2;
   datosPalabras: array[1..NUMPALABRAS] of string =
     (
     'Alicante','Barcelona','Guadalajara','Madrid',
     'Toledo','Malaga','Zaragoza','Sevilla',
     'Valencia','Valladolid'
     );

 Var
   palabra, intento, letras:string; { La palabra a adivinar, la que }
                                    {   el jugador 2 va consiguiendo y
}
                                    {   las letras que se han probado
}
   numeroPalabra: word;
   oportunidades: integer;          { El numero de intentos permitido
}
   letra: char;                     { Cada letra que prueba el jug.
dos }
   i: integer;                      { Para mirar cada letra, con "for"
}
   acertado: boolean;               { Si ha acertado alguna letra }
```

```
 begin
   randomize;                      { Comienzo a generar numeros
aleatorios }
   numeroPalabra :=                { Tomo una palabra al azar }
     random(NUMPALABRAS);
   palabra := datosPalabras[numeroPalabra+1];
   oportunidades := MAXINTENTOS;
   intento := '';                  { Relleno con _ y " " lo que ve Jug.
2 }
   for i:=1 to length(palabra) do
     if palabra[i]= ' ' then
       intento:=intento+' '
     else
       intento:=intento+'*';

   repeat
     clrscr;
     gotoxy (10,6);                { Digo cuantos intentos le quedan }
     writeln('Te quedan ',oportunidades,' intentos');
     gotoxy (10,8);                { Le muestro como va }
     writeln(intento);
     gotoxy (10,10);               { Y le pido otra letra }
     write('Letras intentadas: ', letras);
     gotoxy (10,12);               { Y le pido otra letra }
     write('¿Qué letra? ');
     letra := upcase(readkey);     { Convierto a mayusculas }
     letras := letras + letra;

     acertado := false;            { Miro a ver si ha acertado }
     for i:=1 to length(palabra) do
       if letra=upcase(palabra[i]) then
         begin                     { Comparo en mayusculas }
         intento[i]:=palabra[i];
         acertado := true;
         end;
     if acertado = false then      { Si falla, le queda un intento
menos }
       oportunidades := oportunidades-1;
   until (intento=palabra)         { Hasta que acierte }
     or (oportunidades=0);         { o gaste sus oportunidades }
   gotoxy(10, 15);                 { Le felicito o le digo cual era }

   if intento=palabra then
     begin
     gotoxy (10,8);
     writeln(intento);
     gotoxy(10, 15);
     writeln('¡Acertaste!')
     end
   else
     writeln('Lo siento.  Era: ', palabra);
 end.
```

Una segunda mejora podría ser que realmente "se dibujara" el ahorcado en pantalla en vez de limitarse a decirnos cuantos intentos nos quedan. Como todavía no sabemos manejar la pantalla en modo gráfico, dibujaremos de un

modo rudimentario, empleando letras. El resultado será "feo", algo parecido a esto:

Y lo podríamos conseguir así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Juego del Ahorcado    }
{    (tercera version)     }
{    AHORCA3.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

Program Ahorcado3;

Uses crt;

Const
  NUMPALABRAS = 10;
  datosPalabras: array[1..NUMPALABRAS] of string =
    (
    'Alicante','Barcelona','Guadalajara','Madrid',
    'Toledo','Malaga','Zaragoza','Sevilla',
    'Valencia','Valladolid'
    );
  MAXINTENTOS = 5;  { No debe ser modificado: vamos a "dibujar" 5
cosas }

Var
  palabra, intento, letras:string; { La palabra a adivinar, la que }
```

```
                                        {   el jugador 2 va consiguiendo y
}
                                        {   las letras que se han probado
}
   numeroPalabra: word;
   oportunidades: integer;              { El numero de intentos permitido
}
   letra: char;                         { Cada letra que prueba el jug.
dos }
   i: integer;                          { Para mirar cada letra, con "for"
}
   acertado: boolean;                   { Si ha acertado alguna letra }


procedure PrimerFallo;            { Primer fallo: }
var
  j: byte;                        { Dibujamos la "plataforma" }
begin
  for j := 50 to 60 do
    begin
    gotoxy(j,20);
    write('-');
    end;
end;


procedure SegundoFallo;            { Segundo fallo: }
var
  j: byte;                        { Dibujamos el "palo vertical" }
begin
  for j := 14 to 19 do
    begin
    gotoxy(53,j);
    write('|');
    end;
end;


procedure TercerFallo;            { Tercer fallo: }
var
  j: byte;                        { Dibujamos el "palo superior" }
begin
  for j := 53 to 57 do
    begin
    gotoxy(j,14);
    write('-');
    end;
end;

procedure CuartoFallo;            { Cuarto fallo: }
var
  j: byte;                        { Dibujamos la "plataforma" }
begin
  gotoxy(57,15);
  write('|');
end;


procedure QuintoFallo;            { Quinto fallo: }
var
  j: byte;                        { Dibujamos la "plataforma" }
```

```
begin
  gotoxy(56,16);
  write(' O');
  gotoxy(56,17);
  write('/|\');
  gotoxy(56,18);
  write('/ \');

  for j := 50 to 60 do
    begin
    gotoxy(j,20);
    write('-');
    end;
end;



begin
  randomize;                       { Comienzo a generar numeros
aleatorios }
  numeroPalabra :=                 { Tomo una palabra al azar }
    random(NUMPALABRAS);
  palabra := datosPalabras[numeroPalabra+1];
  oportunidades := MAXINTENTOS;
  intento := '';                   { Relleno con _ y " " lo que ve Jug.
2 }
  for i:=1 to length(palabra) do
    if palabra[i]= ' ' then
      intento:=intento+' '
    else
      intento:=intento+'*';

  repeat
    clrscr;

                                   { Dibujo lo que corresponde del
"patibulo" }
    if oportunidades <= 4 then PrimerFallo;
    if oportunidades <= 3 then SegundoFallo;
    if oportunidades <= 2 then TercerFallo;
    if oportunidades <= 1 then CuartoFallo;

    gotoxy (10,6);                 { Digo cuantos intentos le quedan }
    writeln('Te quedan ',oportunidades,' intentos');
    gotoxy (10,8);                 { Le muestro como va }
    writeln(intento);
    gotoxy (10,10);                { Y le pido otra letra }
    write('Letras intentadas: ', letras);
    gotoxy (10,12);                { Y le pido otra letra }
    write('¿Qué letra? ');
    letra := upcase(readkey);      { Convierto a mayusculas }
    letras := letras + letra;

    acertado := false;             { Miro a ver si ha acertado }
    for i:=1 to length(palabra) do
      if letra=upcase(palabra[i]) then
        begin                      { Comparo en mayusculas }
        intento[i]:=palabra[i];
        acertado := true;
        end;
```

```
     if acertado = false then      { Si falla, le queda un intento
menos }
       oportunidades := oportunidades-1;
   until (intento=palabra)         { Hasta que acierte }
     or (oportunidades=0);         { o gaste sus oportunidades }
                   { Le felicito o le digo cual era }

   if intento=palabra then
     begin
     gotoxy (10,8);
     writeln(intento);
     gotoxy(10, 15);
     writeln('¡Acertaste!')
     end
   else
     begin
     QuintoFallo;
     gotoxy(10, 15);
     writeln('Lo siento.  Era: ', palabra);
     end;
 end.
```

Tema 10.5: Ejemplo: entrada mejorada.

Finalmente, vamos a hacer algo mejor que el "readln". Crearemos una rutina de introducción de datos que permita corregir con mayor facilidad: en cualquier posición de la pantalla, limitando el tamaño máximo de la respuesta, desplazándonos con las flechas, usando las teclas Inicio y Fin, permitiendo insertar o sobreescribir, etc. Podría ser así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Entrada mejorada de   }
{    datos                 }
{    ENTRMEJ.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{--------------------------}

{
  PIDEXY: Entrada mejorada de datos.
  - En cualquier posición de la pantalla.
  - Muestra el tamaño máximo y lo limita.
  - Permite usar las flechas, Inicio, Fin, Ctrl+Flechas, Supr.
  - Modo de inserción o sobreescritura.

 -- Esto es parte de CUPAS, curso de Pascal por Nacho Cabanes --

 -- Este ejemplo sólo funcionará en Turbo Pascal 5.0 o superior --

 Mejoras posibles no incluidas:
```

```
  - Cambiar la forma del cursor según se esté en modo de inserción o
    sobreescritura.
  - Quitar los espacios sobrantes a la derecha de la palabra.
  - Permitir la edición en varias líneas.
  - Seguro que alguna más...  :-)
}

uses crt;


FUNCTION SPC(n:byte):string;              {Espacios en blanco}
var b:string; c:byte;
begin
 b:='';
 for c:=1 to n do b:=b+' ';
 spc:=b
end;


PROCEDURE PIDEXY(var valor:string; long:byte; x,y:byte;
nuevo:boolean);
  { Valor: la variable en la que se va a devolver }
  { Long: longitud máxima permitida }
  { x,y: posición en pantalla }
  { Nuevo: ¿borrar el contenido anterior de "valor"? }

var
  i,fila,posix,posiy: byte;  {bucles, fila, posición, inicial, final}
  inicial,final:byte;           { inicial, final }
  tecla:char;                { tecla que se pulsa }
  insertar:boolean;          { ¿modo de inserción? }

begin
  if nuevo=true then valor:='';
  gotoxy(x,y);
  fila:=wherey;
   write('['); ;
  for i:=1 to long do write('·');            { Para que se vea mejor }
   write(']'); ;
  for i:=length(valor) to long-1 do valor:=valor+'·';
  insertar:=false;
  posix:=1;
  posiy:=y;
  repeat
    gotoxy(x+1,y);
    write(valor);
    gotoxy(posix+x,y);
    tecla:=readkey;                            { Lee una pulsación }

    if ord(tecla)=0 then                       { Si es tecla de función
}
      case ord(readkey) of
      { Flecha izquierda }
      75: if posix>1 then posix:=posix-1;
      { Flecha derecha }
      77: if posix<long then posix:=posix+1;
      { Insert }
      82: if insertar then insertar:=false else insertar:=true;
      { Inicio }
      71: posix:=1;
      { Fin }
```

```
      79: begin i:=long; while valor[i]='·' do dec(i);
          posix:=i end;
      { Supr }
      83: begin for i:=posix to long-1 do valor[i]:=valor[i+1];
          valor[long]:='·' end;
      { Ctrl + <- }
      115: begin inicial:=posix;
           for i:=1 to posix-2 do
             if (valor[i]=' ') and (valor[i+1]<>' ')
           then posix:=i+1;
           if  posix=inicial then posix:=1
           end;
      { Ctrl + -> }
      116: for i:=long-1 downto posix do
             if (valor[i]=' ') and (valor[i+1]<>' ') then posix:=i+1;
      end
    else                                        { Si es carácter
imprimible }
      if tecla>=' ' then
      if not insertar then                      { Si no hay que insertar }
        begin
        valor[posix]:=tecla;                    { Reemplaza }
        if posix<long then posix:=posix+1       { y avanza }
        end
      else                                      { Si hay que insertar }
        begin
        for i:=long downto posix do
          valor[i]:=valor[i-1];                 { Avanza todo }
        valor[posix]:=tecla;
        if posix<long then posix:=posix+1
        end
    else                                        { Si es la tecla de
borrado }
      if (ord(tecla)=8) and (posix>1) then
        begin
        for i:=posix-1 to long-1 do valor[i]:=valor[i+1];
        valor[long]:='·';
        posix:=posix-1
        end;
  until ord(tecla)=13;                        { Hasta pulsar INTRO }

  for i:=1 to length(valor) do                { Al terminar, pone espacios
}
     if valor[i]='·' then valor[i]:=' ';
  gotoxy(x,y); write(' ' + valor + ' '); { Y muestra el resultado }
end;


{ --- Mini-Programa de prueba --- }

var
  nombre: string;

begin
  clrscr;
  gotoxy(3,3); write( '¿Cómo te llamas?' );
  pidexy( nombre, 20, 3,4, TRUE );
  gotoxy(3, 10); writeln( 'Tu nombre es: ', nombre );
  readln;
end.
```

Curso de Pascal. Tema 11a: Manejo de ficheros.

Tema 11.1. Manejo de ficheros (1) - leer un fichero de texto.

Ahora vamos a ver cómo podemos leer ficheros y crear los nuestros propios. Voy a dividir este tema en varias partes: primero veremos como manejar los ficheros de texto, luego los ficheros "con tipo" (que usaremos para hacer una pequeña agenda), y finalmente los ficheros "generales".

Las diferencias entre las dos últimas clases de fichero pueden parecer poco claras ahora e incluso en el próximo apartado, pero en cuanto hayamos visto todos los tipos de ficheros se comprenderá bien cuando usar unos y otros, así que de momento no adelanto más.

Vayamos a lo que nos interesa hoy: un **fichero de texto**. Es un fichero formado por caracteres (letras) normales, que dan lugar a líneas de texto legible.

Si escribimos TYPE AUTOEXEC.BAT desde el DOS, veremos que se nos muestra el contenido de este fichero: una serie de líneas de texto que, al menos, se pueden leer (aunque comprender bien lo que hace cada una es todo un mundo ;-D).

En cambio, si hacemos TYPE COMMAND.COM, aparecerán caracteres raros, se oirá algún que otro pitido... es porque es un fichero ejecutable, que no contiene texto, sino una serie de instrucciones para el ordenador, que nosotros normalmente no sabremos descifrar.

Casi salta a la vista que los ficheros del primer tipo, los de texto, van a ser más fáciles de tratar que los "ficheros en general". Hasta cierto punto es así, y por eso es por lo que vamos a empezar por ellos.

Me voy a centrar en el manejo de ficheros con Turbo Pascal y versiones posteriores (lo poco que hice en Pascal estándar con las órdenes "get" y "put" me cansó lo bastante como para no dar el latazo ahora a no ser que alguien realmente necesite usarlos). Si alguien encuentra problemas con algun otro compilador, que lo diga...

Para acceder a un fichero, hay que seguir unos cuantos **pasos**:

1- Declarar el fichero junto con las demás variables.
2- Asignarle un nombre.
3- Abrirlo, con la intención de leer, escribir o añadir.

4- Trabajar con él.
5- Cerrarlo.

La mejor forma de verlo es con un ejemplo. Vamos a imitar el funcionamiento de la orden del DOS anterior: TYPE AUTOEXEC.BAT.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Muestra AUTOEXEC.BAT  }
{    MAEXEC1.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program MuestraAutoexec;

var
  fichero: text;                          (* Fichero de texto *)
  linea: string;                          (* Línea que leemos *)

begin
  assign( fichero, 'C:\AUTOEXEC.BAT' );   (* Le asignamos el nombre *)
  reset( fichero );                       (* Lo abrimos para lectura
*)
  while not eof( fichero ) do             (* Mientras que no se acabe
*)
    begin
    readln( fichero, linea );             (* Leemos una línea *)
    writeln( linea );                     (* y la mostramos *)
    end;
  close( fichero );                       (* Se acabó: lo cerramos *)
end.
```

Eso es todo. Confío en que sea bastante autoexplicativo, pero aun así vamos a comentar algunas cosas:

- **text** es el tipo de datos que corresponde a un fichero de texto.
- **assign** es la orden que se encarga de asignar un nombre físico al fichero que acabamos de declarar.
- **reset** abre un fichero para lectura. El fichero debe existir, o el programa se interrumpirá avisando con un mensaje de error.
- **eof** es una función booleana que devuelve TRUE (cierto) si se ha llegado ya al final del fichero (end of file).
- Se leen datos con **read** o **readln** igual que cuando se introducían por el teclado. La única diferencia es que debemos indicar desde qué fichero se lee, como aparece en el ejemplo.

- El fichero se cierra con **close**. No cerrar un fichero puede suponer no guardar los últimos cambios o incluso perder la información que contiene.

Si alguien usa Delphi, alguna de esas órdenes no se comporta exactamente igual. Al final de este tema veremos las diferencias.

Curso de Pascal. Tema 11: Manejo de ficheros.

Tema 11.1. Manejo de ficheros (2) - Escribir en un fichero de texto.

Este fichero está abierto para lectura. Si queremos abrirlo para **escritura**, empleamos "**rewrite**" en vez de "reset", pero esta orden hay que utilizarla con cuidado, porque si el fichero ya existe lo machacaría, dejando el nuevo en su lugar, y perdiendo los datos anteriores. Más adelante veremos cómo comprobar si el fichero ya existe.

Para abrir el fichero para **añadir** texto al final, usaríamos "append" en vez de "reset". En ambos casos, los datos se escribirían con

```
writeln( fichero, linea );
```

Ejercicio propuesto: realizar un programa que cree un fichero llamado "datos.txt", que contenga dos líneas de información. La primera será la palabra "Hola" y la segunda será la frase "Que tal?"

Una **limitación** que puede parecer importante es eso de que el fichero debe existir, y si no el programa se interrumpe. En la práctica, esto no es tan drástico. Hay una forma de comprobarlo, que es con una de las llamadas "directivas de compilación". Obligamos al compilador a que temporalmente no compruebe las entradas y salidas, y lo hacemos nosotros mismos. Después volvemos a habilitar las comprobaciones. Ahí va un ejemplo de cómo se hace esto:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Muestra AUTOEXEC.BAT  }
```

```
{    comprobando errores    }
{    MAEXEC2.PAS            }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{---------------------------}

program MuestraAutoexec2;

var

  fichero: text;                          (* Fichero de texto *)
  linea: string;                          (* Línea que leemos *)

begin
  assign( fichero, 'C:\AUTOEXEC.BAT' );   (* Le asignamos el nombre *)
  {$I-}                                   (* Deshabilita comprobación
                                             de entrada/salida *)
  reset( fichero );                     (* Lo intentamos abrir *)
  {$I+}                                 (* La habilitamos otra vez *)

  if ioResult = 0 then                  (* Si todo ha ido bien *)
    begin
    while not eof( fichero ) do         (* Mientras que no se acabe *)
      begin
      readln( fichero, linea );         (* Leemos una línea *)
      writeln( linea );                 (* y la mostramos *)
      end;
    close( fichero );                   (* Se acabó: lo cerramos *)
    end;                                (* Final del "if" *)
end.
```

De modo que **{$I-}** deshabilita la comprobación de las operaciones de entrada y salida, **{$I+}** la vuelve a habilitar, y "**ioresult**" devuelve un número que indica si la última operación de entrada/salida ha sido correcta (cero) o no (otro número, que indica el tipo de error).

Para terminar (al menos momentáneamente) con los ficheros de texto, vamos a ver un pequeño ejemplo, que muestre cómo leer de un fichero y escribir en otro.

Lo que haremos será leer el AUTOEXEC.BAT y crear una copia en el directorio actual llamada AUTOEXEC.BAN (la orden del DOS equivalente sería COPY C:\AUTOEXEC.BAT AUTOEXE.BAN):

```
{---------------------------}
{  Ejemplo en Pascal:       }
{                           }
{    Copia AUTOEXEC.BAT     }
{    CAEXEC1.PAS            }
```

```
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program CopiaAutoexec;

var
  fichero1, fichero2: text;            (* Ficheros de texto *)
  linea: string;                       (* Línea actual *)

begin
  assign( fichero1, 'C:\AUTOEXEC.BAT' );  (* Le asignamos nombre *)
  assign( fichero2, 'AUTOEXEC.BAN' );     (* y al otro *)
  {$I-}                                (* Sin comprobación E/S *)
  reset( fichero1 );                   (* Intentamos abrir uno *)
  {$I+}                                (* La habilitamos otra vez *)
  if ioResult = 0 then                 (* Si todo ha ido bien *)
    begin
    rewrite( fichero2 );               (* Abrimos el otro *)
    while not eof( fichero1 ) do       (* Mientras que no acabe 1 *)
      begin
      readln( fichero1, linea );       (* Leemos una línea *)
      writeln( fichero2, linea );      (* y la escribimos *)
      end;
    writeln( 'Ya está ');              (* Se acabó: avisamos, *)
    close( fichero1 );                 (* cerramos uno *)
    close( fichero2 );                 (* y el otro *)
    end                                (* Final del "if" *)
  else
    writeln(' No he encontrado el fichero! ');      (* Si no existe *)
end.
```

Eso es todo por ahora. Como **ejercicios** propuestos:

- Un programa que vaya grabando en un fichero lo que nosotros tecleemos, como haríamos en el DOS con COPY CON FICHERO.EXT
- Un programa que copie el AUTOEXEC.BAT excluyendo aquellas líneas que empiecen por K o P (mayúsculas o minúsculas).
- Un programa que elimine los espacios innecesarios al final de cada línea de un fichero ASCII. Debe pedir el nombre del fichero de entrada y el de salida, y salir con un mensaje de error si no existe el fichero de entrada o si existe ya uno con el nombre que queremos dar al de salida.

Curso de Pascal. Tema 11: Manejo de ficheros.

Tema 11.3. Manejo de ficheros (3) - Ficheros con tipo.

Hemos visto cómo acceder a los ficheros de texto, tanto para leerlos como para escribir en ellos. Ahora nos centraremos en lo que vamos a llamar "**ficheros con tipo**".

Estos son ficheros en los que cada uno de los elementos que lo integran es del mismo tipo (como vimos que ocurre en un array).

En los de, texto se podría considerar que estaban formados por elementos iguales, de tipo "char", pero ahora vamos a llegar más allá, porque un fichero formado por datos de tipo "record" sería lo ideal para empezar a crear nuestra propia agenda.

Una vez que se conocen los ficheros de texto, no hay muchas diferencias a la hora de un primer manejo: debemos declarar un fichero, asignarlo, abrirlo, trabajar con él y cerrarlo.

Pero ahora podemos hacer más cosas también. Con los de texto, el uso habitual era leer línea por línea, no carácter por carácter. Como las líneas pueden tener cualquier longitud, no podíamos empezar por leer la línea 4 (por ejemplo), sin haber leído antes las tres anteriores. Esto es lo que se llama ACCESO SECUENCIAL.

Ahora sí que sabemos lo que va a ocupar cada dato, ya que todos son del mismo tipo, y podremos aprovecharlo para acceder a una determinada posición del fichero cuando nos interese, sin necesidad de pasar por todas las posiciones anteriores. Esto es el **ACCESO ALEATORIO** (o directo).

La idea es sencilla: si cada ficha ocupa 25 bytes, y queremos leer la número 8, bastaría con "saltarnos" 25*7=175 bytes.

Pero Turbo Pascal (y muchos de los compiladores que nacieron después de él,como Free Pascal) nos lo facilita más aún, con una orden, **seek**, que permite saltar a una determinada posición de un fichero sin tener que calcular nada nosotros mismos. Veamos un par de ejemplos...

Primero vamos a introducir varias fichas en un fichero con tipo:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Crea un fichero "con  }
{    tipo"                 }
{    CREAFT.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
```

```
{     - Turbo Pascal 7.0     }
{--------------------------}

program IntroduceDatos;

type
  ficha = record                              (* Nuestras fichas *)
   nombre: string [80];
   edad:    byte
  end;

var
  fichero:     file of ficha;                 (* Nuestro fichero *)
  bucle:       byte;                          (* Para bucles, claro *)
  datoActual: ficha;                          (* La ficha actual *)

begin
  assign( fichero, 'basura.dat' );            (* Asignamos *)
  rewrite( fichero );                         (* Abrimos (escritura) *)
  writeln(' Te iré pidiendo los datos de cuatro personas...' );
  for bucle := 1 to 4 do                      (* Repetimos 4 veces *)
    begin
    writeln(' Introduce el nombre de la persona número ', bucle);
    readln( datoActual.nombre );
    writeln(' Introduce la edad de la persona número ', bucle);
    readln( datoActual.edad );
    write( fichero, datoActual );             (* Guardamos el dato *)
    end;
  close( fichero );                           (* Cerramos el fichero *)
end.
```

Debería resultar fácil. La única diferencia con lo que ya habíamos visto es que los datos son de tipo "record" y que el fichero se declara de forma distinta, con "**file of TipoBase**".

Ahora vamos a ver cómo leeríamos sólo la tercera ficha de este fichero de datos que acabamos de crear:

```
{--------------------------}
{  Ejemplo en Pascal:       }
{                           }
{    Lee de un fichero      }
{    "con tipo"             }
{    LEEFT.PAS              }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{    - Turbo Pascal 5.0     }
{    - Surpas 1.00          }
{--------------------------}

program LeeUnDato;
```

```
type
  ficha = record
   nombre: string [80];
   edad:   byte
  end;

var
  fichero:    file of ficha;
  bucle:      byte;
  datoActual: ficha;

begin
  assign( fichero, 'basura.dat' );
  reset( fichero );                     (* Abrimos (lectura) *)
  seek( fichero, 2 );                   (* <== Vamos a la ficha 3 *)
  read( fichero, datoActual );          (* Leemos *)
  writeln(' El nombre es: ', datoActual.nombre );
  writeln(' La edad es: ',datoActual.edad );
  close( fichero );                     (* Y cerramos el fichero *)
end.
```

Espero que el listado sea autoexplicativo. La única cosa que merece la pena comentar es eso del "**seek**(fichero, 2)": La posición de las fichas dentro de un fichero de empieza a numerar en 0, que corresponderá a la primera posición. Así, accederemos a la segunda posición con un 1, a la tercera con un 2, y en general a la "n" con "**seek(fichero,n-1)**".

Y ya que como mejor se aprende es practicando, os propongo nuestra famosa **agenda**:

En primer lugar va a ser una agenda que guarde una sola ficha en memoria y que vaya leyendo cada ficha que nos interese desde el disco, o escribiendo en él los nuevos datos (todo ello de forma "automática", sin que quien maneje la agenda se de cuenta). Esto hace que sea más lenta, pero no tiene más limitación de tamaño que el espacio libre en nuestro disco duro. Las posibilidades que debe tener serán:

- Mostrar la ficha actual en pantalla (automático también).
- Modificar la ficha actual.
- Añadir fichas nuevas.
- Salir del programa.

Más adelante ya le iremos introduciendo mejoras, como buscar, ordenar, imprimir una o varias fichas, etc. El formato de cada ficha será:

- Nombre: 20 letras.
- Dirección: 30 letras.
- Ciudad: 15 letras.
- Código Postal: 5 letras.

- Teléfono: 12 letras.
- Observaciones: 40 letras.

A ver quien se atreve con ello...

Curso de Pascal. Tema 11: Manejo de ficheros.

Tema 11.4. Manejo de ficheros (4) - Ficheros generales.

Hemos visto cómo acceder a los ficheros de texto y a los fichero "con tipo". Pero en la práctica nos encontramos con muchos ficheros que no son de texto y que tampoco tienen un tipo de datos claro.

Muchos formatos estándar como PCX, DBF o GIF están formados por una cabecera en la que se dan detalles sobre el formato de los datos, y a continuación ya se detallan los datos en sí.

Esto claramente no es un fichero de texto, y tampoco se parece mucho a lo que habíamos llamado ficheros con tipo. Quizás, un fichero "de tipo byte", pero esto resulta muy lento a la hora de leer ficheros de un cierto tamaño. Como suele ocurrir, "debería haber alguna forma mejor de hacerlo..."

Pues la hay: declarar un **fichero sin tipo**, en el que nosotros mismos decidimos qué tipo de datos queremos leer en cada momento.

Ahora leeremos **bloques** de bytes, y los almacenaremos en un "**buffer**" (memoria intermedia). Para ello tenemos la orden "**BlockRead**", cuyo formato es:

```
procedure BlockRead(var F: Fichero; var Buffer; Cuantos: Word
[; var Resultado: Word]);
```

donde

- "F" es un fichero sin tipo (declarado como "var fichero: file").
- "Buffer" es la variable en la que queremos guardar los datos leídos.
- "Cuantos" es el número de datos que queremos leer.
- "Resultado" (opcional) almacena un número que indica si ha habido algún error.

Hay otra **diferencia** con los ficheros que hemos visto hasta ahora, y es que cuando abrimos un fichero sin tipo con "reset", debemos indicar el tamaño de cada dato (normalmente diremos que 1, y así podemos leer variables más o menos grandes indicándolo con el dato "cuantos" que aparece en BlockRead).

Así, abriríamos el fichero con

```
reset( fichero, 1 );
```

Los **bloques** que leemos con "BlockRead" deben tener un tamaño menor de 64K (resultado de multiplicar "cuantos"por el tamaño de cada dato), al menos en Turbo Pascal (quizá alguna versión posterior evite esta limitación).

El significado de "Resultado" es el siguiente: nos indica cuantos datos ha leido realmente. De este modo, si vemos que le hemos dicho que leyera 30 fichas y sólo ha leído 15, hábilmente podremos deducir que hemos llegado al final del fichero . Si no usamos "resultado" y tratamos de leer las 30 fichas, el programa se interrumpirá, dando un error.

Para **escribir** bloques de datos, utilizaremos "**BlockWrite**", que tiene el mismo formato que BlockRead, pero esta vez si "resultado" es menor de lo esperado indicará que el disco está lleno.

Esta vez, es en "**rewrite**" (cuando abrimos el fichero para escritura) donde deberemos indicar el tamaño de los datos (normalmente 1 byte).

Como las cosas se entienden mejor practicando, ahí va un primer **ejemplo**, tomado de la ayuda en línea de Turbo Pascal y ligeramente retocado, que es un programita que copia un fichero leyendo bloques de 2K:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Copia un fichero      }
{    COPIAFIC.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{--------------------------}

program CopiaFichero;
{ Sencillo y rápido programa de copia de ficheros, SIN comprobación
  de errores }

var
  Origen, Destino: file;
  CantLeida, CantEscrita: Word;
  NombreOrg, NombreDest: String;
  Buffer: array[1..2048] of Char;

begin
  Write( 'Introduzca el nombre del fichero ORIGEN... ' );
  ReadLn( NombreOrg );
```

```
  Write( 'Introduzca el nombre del fichero DESTINO... ' );
  ReadLn( NombreDest );
  Assign( Origen, NombreOrg );
  Reset( Origen, 1 );                                { Tamaño = 1 }
  Assign( Destino, NombreDest );
  Rewrite( Destino, 1 );                             { Lo mismo }
  WriteLn( 'Copiando ', FileSize(Origen), ' bytes...' );
  repeat
    BlockRead( Origen, Buffer, SizeOf(Buffer), CantLeida);
    BlockWrite( Destino, Buffer, CantLeida, CantEscrita);
  until (CantLeida = 0) or (CantEscrita <> CantLeida);
  Close( Origen );
  Close( Destino );
  WriteLn( 'Terminado.' )
end.
```

Un único comentario: es habitual usar "**SizeOf**" para calcular el tamaño de una variable, en vez de calcularlo a mano y escribir, por ejemplo, 2048. Es más fiable y permite modificar el tipo o el tamaño de la variable en la que almacenamos los datos leídos sin que eso repercuta en el resto del programa.

Aplicación a un fichero GIF.

Un segundo ejemplo, que muestra parte de la información contenida en la cabecera de un fichero GIF, leyendolo con la ayuda de un "record":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Lee la cabecera de    }
{    un fichero GIF        }
{    GIFHEAD.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program GifHeader;

Type
  Gif_Header = Record                    { Primeros 13 Bytes de un Gif }
    Firma, NumVer      : Array[1..3] of Char;
    Tam_X,
    Tam_Y              : Word;
    _Packed,
    Fondo,
    Aspecto            : Byte;
  end;
```

```
Var
  Fich : File;
  Cabecera : GIF_Header;
  Nombre: String;

begin
  Write( '¿Nombre del fichero GIF (con extensión)? ');
  ReadLn( Nombre );
  Assign( Fich, Nombre );
  Reset( Fich, 1 );                               { Tamaño base: 1 byte }
  BlockRead( Fich, Cabecera, SizeOf(Cabecera) );
  Close( Fich );
  With Cabecera DO
  begin
    WriteLn('Version: ', Firma, NumVer);
    WriteLn('Resolución: ', Tam_X, 'x',
      Tam_Y, 'x', 2 SHL (_Packed and 7));
  end;
end.
```

Como último **ejercicio** sobre ficheros queda algo que asusta, pero que no es difícil: un programa que deberá preguntarnos el nombre de un fichero, y nos lo mostrará en la pantalla página por página. En la parte izquierda de la pantalla deberán aparecer los bytes en hexadecimal, y en la parte derecha como letras.

Abrir exclusivamente para lectura.

Antes de dar casi por terminado el tema de ficheros, hay algo que nos puede sacad de algún apuro...

Puede interesarnos leer un fichero que se encuentra en un CdRom, o en una red de ordenadores. Normalmente, tanto los CdRom como las "unidades de red" se comportan de forma muy similar a un disco duro "local", de modo que no supondrá ningún problema acceder a su contenido desde MsDos y desde Windows. En cambio, si intentamos leer datos desde un CdRom o desde una unidad de red, con Turbo Pascal 7, de la misma manera que hemos hecho hasta ahora, nos encontraremos con que no podemos, sino que obtenemos un error de "File acces denied" (denegado el acceso al fichero).

El motivo es que Turbo Pascal intenta abrir el fichero tanto para leer de él como para escribir en él. Esto es algo útil si utilizamos nuestro disco duro o un diskette, pero normalmente no tendremos la posibilidad de grabar directamente datos en un CdRom convencional, ni tendremos a veces permisos para escribir en una unidad de red.

Pero podemos cambiar esta forma de comportarse de Turbo Pascal. Lo haremos mediante la variable **FileMode**, que está definida siempre en Turbo Pascal (está en la unidad System). Esta variable normalmente tiene el valor 2

(abrir para leer y escribir), pero también le podemos dar los valores 1 (abrir sólo para escribir) y 0 (abrir sólo para lectura).

Por tanto, la forma de abrir un fichero que se encuentre en un CdRom o en una unidad de red sería añadir la línea

```
FileMode := 0;
```

antes de intentar abrir el fichero con "reset".

Esta misma idea se puede aplicar también con Free Pascal.

Curso de Pascal. Tema 12: Creación de unidades.

Comentamos en el tema 10 que en muchos lenguajes de programación podemos manejar una serie de **bibliotecas** externas (en ingles, **library**) de funciones y procedimientos, que nos permitían ampliar el lenguaje base.

En Turbo Pascal, estas bibliotecas reciben el nombre de "**unidades**" (**unit**), y existen a partir de la versión 5. También existen en otras versiones de Pascal recientes, como Free Pascal.

En su momentos, empleamos la **unidad CRT**, que nos daba una serie de facilidades para manejar la pantalla en modo texto, el teclado y la generación de sonidos sencillos.

Iremos viendo otras unidades estándar cuando accedamos a la pantalla en modo gráfico, a los servicios del sistema operativo, etc. Pero hoy vamos a ver cómo podemos **crear** las nuestras propias.

¿Para qué? Nos podría bastar con teclear en un programa todas las funciones que nos interesen. Si creamos otro programa que las necesite, pues las copiamos también en ese y ya está, ¿no?

¡ NO ! Las unidades nos ayudan a conseguir dos cosas:

- La primera es que los programas sean más **modulares**. Que podamos dejar aparte las funciones que se encargan de batallar con el teclado, por ejemplo, y en nuestro programa principal sólo esté lo que realmente tenga este programa que lo diferencie de los otros. Esto facilita la legibilidad y con ello las posibles correcciones o ampliaciones.

- La segunda ventaja es que no tenemos **distintas versiones** de los mismos procedimientos o funciones. Esto ayuda a ganar espacio en el disco duro, pero eso es lo menos importante. Lo realmente interesante

es que si se nos ocurre una mejora para un procedimiento, todos los programas que lo usen se van a beneficiar de él automáticamente.

Me explico: imaginemos que estamos haciendo un programa de rotación de objetos en tres dimensiones. Creamos nuestra biblioteca de funciones, y la aprovechamos para todos los proyectos que vayamos a hacer en tres dimensiones. No solo evitamos reescribir en cada programa el procedimento RotaPunto, p.ej.., que ahora se tomará de nuestra unidad "MiGraf3D", sino que si descubrimos una forma más rápida de rotarlos, todos los programas que utilicen el procedimiento RotaPunto se verán beneficiados sólo con recompilarlos.

Pero vamos a lo práctico...

Una "**unit**" tiene **dos partes**: una pública, que es aquella a la que podremos acceder, y una privada, que es el desarrollo detallado de esa parte pública, y a esta parte no se puede acceder desde otros programas.

La parte **pública** se denota con la palabra "**interface**", y la **privada** con "**implementation**".

Debajo de *interface* basta indicar los nombres de los procedimientos que queremos "exportar", así como las variables, si nos interesase crear alguna. Debajo de *implementation* escribimos realmente estos procedimientos o funciones, tal como haríamos en un programa normal.

Veamos un **ejemplito** para que se entienda mejor.

Nota: este ejemplo **NO SE PUEDE EJECUTAR**. Recordemos que una Unit es algo auxiliar, una biblioteca de funciones y procedimientos que nosotros utilizaremos DESDE OTROS PROGRAMAS. Después de este ejemplo de Unit incluyo un ejemplo de programa cortito que la emplee.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Unidad que "mejora"   }
{    la CRT                }
{    MICRT1.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}
unit miCrt1;
```

```
interface                    { Parte "pública", que se exporta }
procedure AtXY( X, Y: byte ; texto: string );
                             { Escribe un texto en ciertas coordenadas }

implementation               { Parte "privada", detallada }
uses crt;                    { Usa a su vez la unidad CRT }
procedure AtXY( X, Y: byte ; texto: string );
begin
  gotoXY( X, Y);             { Va a la posición adecuada }
  write( texto );
end;
end.                         { Final de la unidad }
```

Este ejemplo declara un procedimiento "AtXY" que hace un GotoXY y un Write en un solo paso.

Un programa que lo emplease podría ser simplemente:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Programa que usa la   }
{    unit "MICRT1"         }
{    PMICRT1.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}
program PruebaDeMiCrt1;

uses miCrt1;

begin
  AtXY( 7, 5, 'Texto en la posición 7,5.' );
end.
```

Este programa no necesita llamar a la unidad CRT original, sino que nuestra unidad ya lo hace por él.

Ahora vamos a mejorar ligeramente nuestra unidad, añadiéndole un procedimiento "pausa":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Unidad que mejora la  }
{    CRT (segunda versión) }
{    MICRT2.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
```

```
{  por Nacho Cabanes         }
{                            }
{  Comprobado con:           }
{    - Free Pascal 2.2.0w    }
{    - Turbo Pascal 7.0      }
{    - Tmt Pascal Lt 1.20    }
{--------------------------}
unit miCrt2;                 { Unidad que "mejora más" la CRT }

{------------------}
interface                    { Parte "pública", que se exporta }

procedure AtXY( X, Y: byte ; texto: string );
procedure Pausa;

{------------------}
implementation               { Parte "privada", detallada }

uses crt;                    { Usa a su vez la unidad CRT }

var tecla: char;             { variable privada: el usuario no
                               puede utilizarla }

procedure AtXY( X, Y: byte ; texto: string );
begin
  gotoXY( X, Y);             { Va a la posición adecuada }
  write( texto );
end;

procedure Pausa;             { Pausa, llamando a ReadKey }
begin
  tecla := ReadKey;          { El valor de "tecla" se pierde }
end;

{------------------}
end.                         { Final de la unidad }
```

Un programa que usase esta unidad, junto con la CRT original podría ser:

```
{--------------------------}
{  Ejemplo en Pascal:        }
{                            }
{    Prueba de la unidad     }
{    MICRT2                  }
{    PMICRT2.PAS             }
{                            }
{  Este fuente procede de    }
{  CUPAS, curso de Pascal    }
{  por Nacho Cabanes         }
{                            }
{  Comprobado con:           }
{    - Free Pascal 2.2.0w    }
{    - Turbo Pascal 7.0      }
{    - Tmt Pascal Lt 1.20    }
{--------------------------}
program PruebaDeMiCrt2;
uses crt, miCrt2;
begin
  ClrScr;                                    { De Crt }
  atXY( 7, 5, 'Texto en la posición 7,5.' );    { de miCrt2 }
```

```
  pausa;                                      { de miCrt2 }
end.
```

Finalmente, hay que destacar que las unidades pueden contener más cosas además de funciones y procedimientos: pueden tener un "trozo de programa", su código de **inicialización**, como por ejemplo:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Unidad que mejora la  }
{    CRT (tercera versión) }
{    MICRT3.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}
unit miCrt3;                  { Unidad que "mejora más" la CRT }

{-------------------}
interface                     { Parte "pública", que se exporta }

var EraMono: boolean;         { Variable pública, el usuario puede
                                acceder a ella }
procedure AtXY( X, Y: byte ; texto: string );
procedure Pausa;

{-------------------}
implementation                { Parte "privada", detallada }

uses crt;                     { Usa a su vez la unidad CRT }
var tecla: char;              { variable privada: el usuario no
                                puede utilizarla }

procedure AtXY( X, Y: byte ; texto: string );
begin
  gotoXY( X, Y);              { Va a la posición adecuada }
  write( texto );
end;

procedure Pausa;              { Pausa, llamando a ReadKey }
begin
  tecla := ReadKey;           { El valor de "tecla" se pierde }
end;

{-------------------}         { Aquí va la inicialización }
begin
  if lastmode = 7             { Si el modo de pantalla era monocromo }
    then EraMono := true      { EraMono será verdadero }
  else EraMono := false;      { si no => falso }
end.                          { Final de la unidad }
```

y el programa podría usar la variable EraMono sin declararla:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Prueba de la unidad   }
{    MICRT3                }
{    PMICRT3.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}
program PruebaDeMiCrt3;
uses crt, miCrt3;
begin
  ClrScr;                                     { De Crt }
  atXY( 7, 5, 'Texto en la posición 7,5.' );   { de miCrt3 }
  if not EraMono then
    atXY ( 10, 10, 'Modo de color ' );
  pausa;                                      { de miCrt3 }
end.
```

Se podría hablar mucho más sobre las unidades, pero intentaré ser breve:

- Al compilar una unidad se crea un fichero con **extensión .TPU** (.PPU para Free Pascal), al que se puede acceder desde nuestros programas con dos condiciones: que empleemos la misma versión de compilador (el formato de estos ficheros variaba en cada versión de Turbo Pascal, y quizá también entre versiones de Free Pascal), y que sepamos cómo es la parte pública (interface).

Nota: Por eso mucha gente distribuía sus bibliotecas de rutinas Pascal en forma de TPU: se podía usar las facilidades que nos daban (si teníamos la misma versión de Pascal), pero como no teníamos disponible el fuente, no podíamos modificarlo ni redistribuirlo con nuestro nombre, por ejemplo. Hoy en día, se tiende más a ceder todo el código fuente, y pedir a los usuarios que conserven el copyright y/o envíen al autor las mejoras que propongan.

- En Turbo Pascal 7 para MsDos, cada unidad tiene su propio **segmento de código** (esto va para quien conozca la estructura de la memoria en los PC), así que cada unidad puede almacenar hasta 64k de procedimientos o funciones. Los datos son comunes a todas las unidades, con la limitación 64k en total (un segmento) para todos los datos (estáticos) de todo el programa. Si queremos almacenar datos de más de 64k en el programa, tenga una o más unidades, deberemos

emplear variables dinámicas, distintas en su manejo de las que hemos visto hasta ahora (estáticas), pero eso ya lo veremos el próximo día... :-)

Esta vez no propongo ejercicios. Que cada uno se construya las units que quiera, como quiera, y vaya consultando dudas...

Tema 13: Variables dinámicas.

En Pascal estándar, tal y como hemos visto hasta ahora, tenemos una serie de variables que declaramos al principio del programa o de cada módulo (función o procedimiento, unidad, etc). Estas variables, que reciben el nombre de **estáticas**, tienen un tamaño asignado desde el momento en que se crea el programa.

Esto es cómodo para detectar errores y rápido si vamos a manejar estruturas de datos que no cambien, pero resulta poco eficiente si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Es el caso de una agenda: tenemos una serie de fichas, e iremos añadiendo más. Si reservamos espacio para 10, no podremos llegar a añadir la número 11, estamos limitando el máximo. En este caso, la solución que vimos fue la de trabajar siempre en el disco. No tenemos límite en cuanto a número de fichas, pero es muchísimo más lento.

Lo ideal sería aprovechar mejor la memoria que tenemos en el ordenador, para guardar en ella todas las fichas o al menos todas aquellas que quepan en memoria.

Una solución "típica" es **sobredimensionar**: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a concer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobrepasado, y además en Turbo Pascal tenemos muy poca memoria disponible para variables estáticas: 64K (un segmento, limitaciones heredadas del manejo de memoria en el DOS en modo real).

Por ejemplo, si en nuestra agenda guardamos lo siguientes datos de cada persona: nombre (40 letras), dirección (2 líneas de 50 letras), teléfono (10 letras), comentarios (70 letras), que tampoco es demasiada información, tenemos que cada ficha ocupa 235 bytes, luego podemos almacenar menos de 280 fichas en memoria, incluso suponiendo que las demás variables que empleemos ocupen muy poco espacio.

Todas estas limitaciones se solucionan con el uso de **variables dinámicas**, para las cuales se reserva espacio en el momento de ejecución del programa,

sólo en la cantidad necesaria, se pueden añadir elementos después, y se puede aprovechar toda la memoria convencional (primeros 640K) de nuestro equipo.

Si además nuestro compilador genera programas en modo protegido del DOS, podremos aprovechar toda la memoria real de nuestro ordenador (4 Mb, 8 Mb, etc). Si crea programas para sistemas operativos que utilicen memoria virtual (como OS/2 o Windows, que destinan parte del disco duro para intercambiar con zonas de la memoria principal, de modo que aparentemente tenemos más memoria disponible), podremos utilizar también esa memoria de forma transparente para nosotros.

Así que se acabó la limitación de 64K. Ahora podremos tener, por ejemplo, 30 Mb de datos en nuestro programa y con un acceso muchísimo más rápido que si teníamos las fichas en disco, como hicimos antes.

Ahora "sólo" queda ver cómo utilizar estas variables dinámicas. Esto lo vamos a ver en 3 apartados. El primero (éste) será la introducción y veremos cómo utilizar arrays con elementos que ocupen más de 64K. El segundo, manejaremos las "listas enlazadas". El tercero nos centraremos en los "árboles binarios" y comentaremos cosas sobre otras estructuras.

Vamos allá... :-)

La idea de variable dinámica está muy relacionada con el concepto de **puntero** (o apuntador, en inglés "pointer"). Un puntero es una variable que "apunta" a una determinada posición de memoria, en la que se encuentran los datos que nos interesan.

Como un puntero almacena una dirección de memoria, sólo gastará 4 bytes de esos 64K que teníamos para datos estáticos. El resto de la memoria (lo que realmente ocupan los datos) se asigna en el momento en el que se ejcuta el programa y se toma del resto de los 640K. Así, si nos quedan 500K libres, podríamos guardar cerca de 2000 fichas en memoria, en vez de las 280 de antes. De los 64K del segmento de datos sólo estaríamos ocupando cerca de 8K (2000 fichas x 4 bytes).

Veamoslo con un **ejemplo** (bastante inútil, "púramente académico" X-D) que después comentaré un poco.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer ejemplo de     }
{    variables dinámicas   }
{    DINAM1.PAS            }
{                          }
{  Este fuente procede de  }
```

```
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Dinamicas;

type
  pFicha = ^Ficha;                    (* Puntero a la ficha *)

  Ficha = record                      (* Datos almacenados *)
    nombre: string[40];
    edad: byte
    end;

var
  fichero: file of ficha;             (* El fichero, claro *)
  datoLeido: ficha;                   (* Una ficha que se lee *)
  indice: array [1..1000] of pFicha;  (* Punteros a 1000 fichas *)
  contador: integer;                  (* Nº de fichas que se lee *)

begin
  assign( fichero, 'Datos.Dat' );     (* Asigna el fichero *)
  reset( fichero );                   (* Lo abre *)
  for contador := 1 to 1000 do        (* Va a leer 1000 fichas *)
    begin
    read( fichero, datoleido );       (* Lee cada una de ellas *)
    new( indice[contador] );          (* Le reserva espacio *)
    indice[contador]^ := datoLeido;   (* Y lo guarda en memoria *)
    end;
  close( fichero );                   (* Cierra el fichero *)
  writeln('El nombre de la ficha 500 es: ');
  writeln(indice[500]^.nombre);
  for contador := 1 to 1000 do        (* Liberamos memoria usada *)
    dispose( indice[contador] );
end.
```

El **acento circunflejo** (^) quiere decir "que apunta a" o "apuntado por". Así,

```
pFicha = ^Ficha;
```

indica que pFicha va a "apuntar a" datos del tipo Ficha, y

```
indice[500]^.nombre
```

será el campo nombre del dato al que apunta la dirección 500 del índice. El manejo es muy parecido al de un array que contenga records, como ya habíamos visto, con la diferencia de el carácter ^, que indica que se trata de punteros.

Antes de asignar un valor a una variable dinámica, hemos de **reservarle espacio** con "**new**", porque si no estaríamos escribiendo en una posición de

memoria que el compilador no nos ha asegurado que esté vacía, y eso puede hacer que "machaquemos" otros datos, o parte del propio programa, o del sistema operativo... esto es muy peligroso, y puede provocar desde simples errores muy difíciles de localizar hasta un "cuelgue" en el ordenador o cosas más peligrosas...

Cuando terminamos de utilizar una variable dinámica, debemos **liberar** la memoria que habíamos reservado. Para ello empleamos la orden "**dispose**", que tiene una sintaxis igual que la de new:

```
new( variable );             { Reserva espacio }
dispose( variable );         { Libera el espacio reservado }
```

Bueno, ya está bien por ahora. Hemos visto una forma de tener arrays de más de 64K de tamaño, pero seguimos con la limitación en el número de fichas. En el próximo apartado veremos cómo evitar también esto... :-)

Experimentad, experimentad... };-)

Tema 13.2: Variables dinámicas (2).

Vimos una introducción a los punteros y comentamos cómo se manejarían combinados con arrays. Antes de pasar a estructuras más complejas, vamos a hacer un **ejemplo** práctico (que realmente funcione).

Tomando la base que vimos, vamos a hacer un lector. Algo parecido al README.COM que incluyen Borland y otras casas en muchos de sus programas.

```
 Lector de ficheros de texto. Nacho Cabanes, 95.          Pulse ESC para salir
PROGRAM ARBOLES_FRACTALES;

USES
    Graph,Crt;


CONST
   rap_ac = 0.5;
   rap_cd = 0.8;
   rap_ce = 0.8;


TYPE
    point = record
              x : integer;
              y : integer;
            end;


VAR
   controlgraf, modograf : integer;
   xmax, ymax, nombre    : integer;
 Use las flechas y AvPag, RePag para moverse.             Líneas:7-29/114
```

Es un programa al que le decimos el nombre de un **fichero de texto**, lo lee y lo va mostrando por pantalla. Podremos desplazarnos hacia arriba y hacia abajo, de línea en línea o de pantalla en pantalla. Esta vez, en vez de leer un registro "record", leeremos "strings", y por comodidad los limitaremos a la anchura de la pantalla, 80 caracteres. Tendremos una capacidad, por ejemplo, de 2000 líneas, de modo que gastaremos como mucho 80*2000 = 160 K aprox.

Hay cosas que se podrían hacer mejor, pero me he centrado en procurar que sea lo más legible posible. Espero haberlo conseguido...

Eso sí: un comentario obligado: eso que aparece en el fuente de #27 es lo mismo que escribir "chr(27)", es decir, corresponde al carácter cuyo código ASCII es el 27.

Vamos allá...

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Lector de ficheros    }
{    de texto              }
{    LECTOR.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program Lector;              { Lee ficheros de texto }

uses                                    { Unidades externas: }
   crt;                                 { Pantalla de texto y teclado }

const
   MaxLineas = 2000;                    { Para modificarlo facilmente }

   kbEsc = #27;                         { Código ASCII de la tecla ESC
}

   kbFuncion = #0;                      { Las teclas de función
devuelven
                                          0 + otro código }
   kbArr = #72;                         { Código de Flecha Arriba }
   kbPgArr = #73;                       { Página Arriba }
   kbAbj = #80;                         { Flecha Abajo }
   kbPgAbj = #81;                       { Página Abajo }


type
  LineaTxt = string [80];               { Una línea de texto }
```

```
  PLineaTxt = ^LineaTxt;                 { Puntero a lína de texto }
  lineas = array[1..maxLineas]           { Nuestro array de líneas }
      of PLineaTxt;

var
  nomFich: string;                       { El nombre del fichero }
  fichero: text;                         { El fichero en sí }
  datos: lineas;                         { Los datos, claro }
  lineaActual: string;                   { Cada línea que lee del
fichero }
  TotLineas: word;                       { El número total de líneas }
  Primera: word;                         { La primera línea en pantalla
}


Procedure Inicio;                        { Abre el fichero }
begin
  textbackground(black);                 { Colores de comienzo: fondo
negro }
  textcolor(lightgray);                  {   y texto gris }
  clrscr;                                { Borramos la pantalla }
  writeln('Lector de ficheros de texto.');
  writeln;
  write('Introduzca el nombre del fichero: ');
  readln(nomFich);
end;


Procedure Pantalla;                      { Pantalla del lector }
begin
  textbackground(red);                   { Bordes de la pantalla }
  textcolor(yellow);                     { Amarillo sobre rojo }
  clrscr;                                { ... }
  gotoxy(2,1);
  write('Lector de ficheros de texto. Nacho Cabanes, 95.'
   +'           Pulse ESC para salir');
  gotoxy(2,25);
  write('Use las flechas y AvPag, RePag para moverse.');
  window(1,2,80,24);                     { Define una ventana interior }
  textbackground(black);                 { Con distintos colores }
  textcolor(white);
  clrscr;
end;


Procedure EscribeAbajo(mensaje: string);     { Escribe en la línea
inferior }
begin
  window(1,1,80,25);                     { Restaura la ventana }
  textbackground(red);                   { Colores de los bordes: }
  textcolor(yellow);                     { Amarillo sobre rojo }
  gotoxy(60,25);                         { Se sitúa }
  write(mensaje);                        { y escribe }
  window(1,2,80,24);                     { Redefine la ventana interior
}
  textbackground(black);                 { y cambia los colores }
  textcolor(white);
end;
```

```
procedure salir;                      { Antes de abandonar el
programa }
var i: word;
begin
  for i := 1 to TotLineas             { Para cada línea leída, }
    do dispose(datos[i]);             {  libera la memoria ocupada }
  window(1,1,80,25);                  { Restablece la ventana de
texto, }
  textbackground(black);              { el color de fondo, }
  textcolor(white);                   { el de primer plano, }
  clrscr;                             { borra la pantalla }
  writeln('Hasta otra...');           { y se despide }
end;


Procedure Pausa;                      { Espera a que se pulse una
tecla }
var tecla:char;
begin
  tecla:=readkey;
end;


Function strs(valor:word):string;     { Convierte word a string }
var cadena: string;
begin
  str(valor,cadena);
  strs := cadena;
end;

function min(a,b: word): word;        { Halla el mínimo de dos
números }
begin
  if a<b then min := a else min := b;
end;


procedure Lee;
begin;
  clrscr;
  TotLineas := 0;                     { Inicializa variables }
  Primera := 0;
  while (not eof(fichero))            { Mientras quede fichero }
      and (TotLineas < MaxLineas) do  { y espacio en el array }
    begin
    readln( fichero, LineaActual );     { Lee una línea }
    TotLineas := TotLineas + 1 ;        { Aumenta el contador }
    new(datos[TotLineas]);              { Reserva memoria }
    datos[TotLineas]^ := LineaActual;   { y guarda la línea }
    end;
  if TotLineas > 0                    { Si realmente se han leido
líneas }
    then Primera := 1;                {   empezaremos en la primera }
  close(fichero);                     { Al final, cierra el fichero }
end;



procedure Muestra;                    { Muestra el fichero en
pantalla }
var
```

```
  i: word;                            { Para bucles }
  tecla: char;                        { La tecla que se pulsa }
begin;
  repeat
    for i := Primera to Primera+22 do
      begin
      gotoxy(1, i+1-Primera );        { A partir de la primera línea
}
      if datos[i] <> nil then         { Si existe dato
correspondiente, }
        write(datos[i]^);             {   lo escribe }
      clreol;                         { Y borra hasta fin de línea }
      end;
    EscribeAbajo('Líneas:'+strs(Primera)+'-'+
      strs(Primera+22)+'/'+strs(TotLineas)+' ');
    tecla := readkey ;
    if tecla = kbFuncion then begin   { Si es tecla de función }
      tecla := readkey;               { Mira el segundo código }
      case tecla of
        kbArr:                        { Flecha arriba }
          if Primera>1 then Primera := Primera -1;
        kbAbj:                        { Flecha abajo }
          if Primera<TotLineas-22 then Primera := Primera + 1;
        kbPgArr:                      { Página arriba }
          if Primera>22 then Primera := Primera - 22
            else Primera := 1;
        kbPgAbj:                      { Página Abajo }
          if Primera< (TotLineas-22) then
            Primera := Primera + min(22, TotLineas-23)
            else Primera := TotLineas-22;
        end;
      end;
  until tecla = kbEsc;
end;

begin
  Inicio;                             { Pantalla inicial }
  assign(fichero, nomFich);           { Asigna el fichero }
  {$I-}                               { desactiva errores de E/S }
  reset(fichero);                     { e intenta abrirlo }
  {$I+}                               { Vuelve a activar errores }
  if IOresult = 0 then               { Si no ha habido error }
    begin
    Pantalla;                           { Dibuja la pantalla }
    Lee;                                { Lee el fichero }
    Muestra;                            { Y lo muestra }
    end
  else                                { Si hubo error }
    begin
    writeln(' ¡ No se ha podido abrir el fichero ! ');   { Avisa }
    pausa;
    end;
  salir                               { En cualq. caso, sale al final
}
end.
```

Pues eso es todo por hoy... :-)

(Si aparece alguna cosa "rara", como la palabra **NIL**, no te preocupes: en el próximo apartado está explicada).

Tema 13.3: Variables dinámicas (3).

Habíamos comentado cómo podíamos evitar las limitaciones de 64K para datos y de tener que dar un tamaño fijo a las variables del programa.

Después vimos con más detalle como podíamos hacer arrays de más de 64K. Aprovechábamos mejor la memoria y a la vez seguíamos teniendo acceso directo a cada dato. Como inconveniente: no podíamos añadir más datos que los que hubiéramos previsto al principio (2000 líneas en el caso del lector de ficheros que vimos como ejemplo).

Pues ahora vamos a ver dos tipos de estructuras **totalmente dinámicas** (frente a los arrays, que eran estáticos). En esta lección serán **las listas**, y en la próxima trataremos los árboles binarios. Hay otras muchas estructuras, pero no son difíciles de desarrollar si se entienden bien estas dos.

Ahora "el truco" consistirá en que dentro de cada dato almacenaremos todo lo que nos interesa, pero también una referencia que nos dirá dónde tenemos que ir a buscar el siguiente.

Sería algo así como:

```
(Posición: 1023).
Nombre : 'Nacho Cabanes'
DireccionFido : '2:346/3.30'
SiguienteDato : 1430
```

Este dato está almacenado en la posición de memoria número 1023. En esa posición guardamos el nombre y la dirección (o lo que nos interese) de esta persona, pero también una información extra: la siguiente ficha se encuentra en la posición 1430.

Así, es muy cómodo recorrer la lista de forma **secuencial**, porque en todo momento sabemos dónde está almacenado el siguiente dato. Cuando lleguemos a uno para el que no esté definido cual es el siguiente, quiere decir que se ha acabado la lista.

Hemos perdido la ventaja del acceso directo: ya no podemos saltar directamente a la ficha número 500. Pero, por contra, podemos tener tantas fichas como la memoria nos permita.

Para añadir un ficha, no tendríamos más que reservar la memoria para ella, y el Turbo Pascal nos diría "le he encontrado sitio en la posición 4079". Así que nosotros iríamos a la última ficha y le diríamos "tu siguiente dato va a estar en la posición 4079".

Esa es la idea "intuitiva". Espero que a nadie le resulte complicado. Así que vamos a empezar a concretar cosas en forma de programa en **Pascal**.

Primero cómo sería ahora cada una de nuestras fichas:

```
type
  pFicha = ^Ficha;               { Puntero a la ficha }
  Ficha = record                 { Estos son los datos que guardamos: }
     nombre: string[30];         {   Nombre, hasta 30 letras }
     direccion: string[50];      {  Direccion, hasta 50 }
     edad: byte;                 {   Edad, un numero < 255 }
  siguiente: pFicha;             {   Y dirección de la siguiente }
end;
```

La nomenclatura ^Ficha ya la habíamos visto. Se refiere a que eso es un "**puntero** al tipo Ficha". Es decir, la variable "pFicha" va a tener como valor una dirección de memoria, en la que se encuentra un dato del tipo Ficha.

La **diferencia** está en el campo "siguiente" de nuestro registro, que es el que indica donde se encuentra la ficha que va después de la actual.

Un puntero que "no apunta a ningún sitio" tiene el valor **NIL**, que nos servirá después para comprobar si se trata del final de la lista: todas las fichas "apuntarán" a la siguiente, menos la última, que "no tiene siguiente" ;-)

Entonces la primera ficha la definiríamos con

```
var dato1: pFicha;                  { Va a ser un puntero a ficha
}</tt>
```

y la crearíamos con

```
new (dato1);                     { Reservamos memoria }
dato1^.nombre := 'Pepe';         { Guardamos el nombre, }
dato1^.direccion := 'Su casa';   { la dirección }
dato1^.edad := 102;              {   la edad :-o }
dato1^.siguiente := nil;         { y no hay ninguna más }
```

Ahora podríamos añadir una ficha detrás de ella. Primero guardamos espacio para la nueva ficha, como antes:

```
var dato2: pFicha;                  { Va a ser otro puntero a ficha }
new (dato2);                        { Reservamos memoria }
dato2^.nombre := 'Juan';            { Guardamos el nombre, }
dato2^.direccion := 'No lo sé';     {  la dirección }
dato2^.edad := 35;                  {   la edad }
dato2^.siguiente := nil;            { y no hay ninguna detrás }
```

y ahora enlazamos la anterior con ella:

```
dato1^.siguiente := dato2;
```

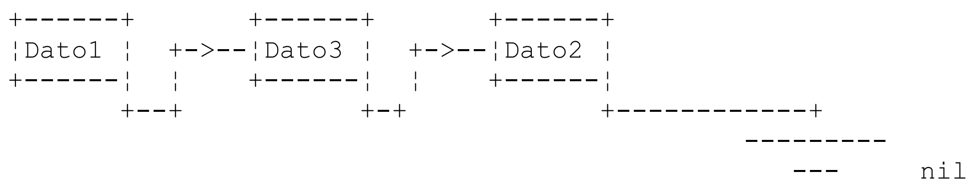
Si quisieramos introducir los datos **ordenados alfabéticamente**, basta con ir comparando cada nuevo dato con los de la lista, e insertarlo donde corresponda. Por ejemplo, para insertar un nuevo dato entre los dos anteriores, haríamos:

```
var dato3: pFicha;                { Va a ser otro puntero a ficha }
new (dato3);
dato3^.nombre := 'Carlos';
dato3^.direccion := 'Por ahí';
dato3^.edad := 14;
dato3^.siguiente := dato2;         { enlazamos con la siguiente }
dato1^.siguiente := dato3;         { y con la anterior }
```

La estructura que hemos obtenido es la siguiente

```
Dato1 - Dato3 - Dato2 - nil
```

o gráficamente:



Es decir: cada ficha está enlazada con la siguiente, salvo la última, que no está enlazada con ninguna (apunta a NIL).

Si ahora quisiéramos **borrar** Dato3, tendríamos que seguir dos pasos:

1.- Enlazar Dato1 con Dato2, para no perder información.
2.- Liberar la memoria ocupada por Dato3.

Esto, escrito en "paskalero" sería:

```
    dato1^.siguiente := dato2;          { Enlaza Dato1 y Dato2 }
    dispose(dato3);                     { Libera lo que ocupó Dato3 }
```

Hemos empleado tres variables para guardar tres datos. Si tenemos 20 datos, ¿necesitaremos 20 variables? ¿Y 3000 variables para 3000 datos?

Sería tremendamente ineficiente, y no tendría mucho sentido. Es de suponer que no sea así. En la práctica, basta con dos variables, que nos indicarán el principio de la lista y la posición actual, o incluso sólo una para el principio de la lista.

Por ejemplo, un procedimiento que **muestre en pantalla** toda la lista se podría hacer de forma recursiva así:

```
procedure MuestraLista ( inicial: pFicha );
begin
  if inicial <> nil then          { Si realmente hay lista }
  begin
    writeln('Nombre: ', inicial^.nombre);
    writeln('Dirección: ', inicial^.direccion);
    writeln('Edad: ', inicial^.edad);
    MuestraLista ( inicial^.siguiente );     { Y mira el siguiente }
  end;
end;
```

Lo llamaríamos con "MuestraLista(Dato1)", y a partir de ahí el propio procedimiento se encarga de ir mirando y mostrando los siguientes elementos hasta llegar a NIL, que indica el final.

Aquí va un programilla de **ejemplo**, que **ordena** los elementos que va insertando... y poco más }:-) :

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de listas     }
{    dinámicas enlazadas   }
{    LISTAS.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program EjemploDeListas;

type
```

```
  puntero = ^TipoDatos;
  TipoDatos = record
    numero: integer;
    sig:    puntero
  end;


function CrearLista(valor: integer): puntero;  {Crea la lista, claro}
var
  r: puntero;                 { Variable auxiliar }
begin
  new(r);                     { Reserva memoria }
  r^.numero := valor;         { Guarda el valor }
  r^.sig := nil;              { No hay siguiente }
  CrearLista := r             { Crea el puntero }
end;


procedure MuestraLista ( lista: puntero );
begin
  if lista <> nil then                { Si realmente hay lista }
    begin
    writeln(lista^.numero);           { Escribe el valor }
    MuestraLista (lista^.sig )        { Y mira el siguiente }
    end;
end;


procedure InsertaLista( var lista: puntero; valor: integer);
var
  r: puntero;                          { Variable auxiliar }
begin
  if lista <> nil then                 { Si hay lista }
    if lista^.numero<valor             {   y todavía no es su sitio }
      then                             {   hace una llamada recursiva:
}
      InsertaLista(lista^.sig,valor)  {   mira la siguiente posición }
    else                               { Caso contrario: si hay lista }
      begin                            {   pero hay que insertar ya: }
      new(r);                          {   Reserva espacio, }
      r^.numero := valor;              {   guarda el dato }
      r^.sig := lista;                 {   pone la lista a continuac. }
      lista := r                       {   Y hace que comience en }
      end                              {     el nuevo dato: r }
  else                                 { Si no hay lista }
    begin                              {   deberá crearla }
    new(r);                            {   reserva espacio }
    r^.numero := valor;                {   guarda el dato }
    r^.sig := nil;                     {   no hay nada detrás y }
    lista := r                         {   hace que la lista comience }
    end                                {     en el dato: r }
end;


var
  l: puntero;               { Variables globales: la lista }

begin
  l := CrearLista(5);       { Crea una lista e introduce un 5 }
  InsertaLista(l, 3);       { Inserta un 3 }
  InsertaLista(l, 2);       { Inserta un 2 }
```

```
  InsertaLista(l, 6);      { Inserta un 6 }
  MuestraLista(l)          { Muestra la lista resultante }
end.
```

Ejercicio propuesto: *¿Se podría quitar de alguna forma el segundo "else" de InsertaLista?*
Ejercicio propuesto: *¿Cómo sería un procedimiento que borrase toda la lista?*
Ejercicio propuesto: *¿Cómo sería un procedimiento de búsqueda, que devolviera la posición en la que está un dato, o NIL si el dato no existe?*
Ejercicio propuesto: *¿Cómo se haría una lista "doblemente enlazada", que se pueda recorrer hacia adelante y hacia atrás?*

Pues eso es todo por hoy... ;-)

Tema 13.4: Variables dinámicas (4).

El último día vimos cómo hacer listas dinámicas enlazadas, y cómo podíamos ir insertando los elementos en ellas de forma que siempre estuviesen ordenadas.

Hay varios casos particulares. Sólo comentaré algunos de ellos de pasada:

- Una **pila** es un caso particular de lista, en la que los elementos siempre se introducen y se sacan por el mismo extremo (se apilan o se desapilan). Es como una pila de libros, en la que para coger el tercero deberemos apartar los dos primeros (excluyendo malabaristas, que los hay). Este tipo de estructura se llama **LIFO** (Last In, First Out: el último en entrar es el primero en salir).
- Una **cola** es otro caso particular, en el que los elementos se introducen por un extremo y se sacan por el otro. Es como se supone que debería ser la cola del cine: los que llegan, se ponen al final, y se atiende primero a los que están al principio. Esta es una estructura **FIFO** (First In, First Out).

Estas dos son estructuras más sencillas de programar de lo que sería una lista en su caso general, pero que son también útiles en muchos casos. De momento no incluyo ejemplos de ninguna de ellas, y me lo reservo para los ejercicios y para cuando lleguemos a Programación Orientada a Objetos, y será entonces cuando creemos nuestro objeto Pila y nuestro objeto Cola (recordádmelo si se me pasa). Aun así, si alguien tiene dudas ahora, que no se corte en decirlo, o que se espere un poco, hasta ver las soluciones de los "deberes"... }:-)

Finalmente, antes de pasar con los "**árboles**", comentaré una mejora a estas listas enlazadas que hemos visto. Tal y como las hemos tratado, tienen la ventaja de que no hay limitaciones tan rígidas en cuanto a tamaño como en

las variables estáticas, ni hay por qué saber el número de elementos desde el principio. Pero siempre hay que recorrerlas desde DELANTE hacia ATRAS, lo que puede resultar lento. Una mejora relativamente evidente es lo que se llama una **lista doble** o lista doblemente enlazada: si guardamos punteros al dato anterior y al siguiente, en vez de sólo al siguiente, podremos avanzar y retroceder con comodidad. Pero tampoco me enrollo más con ello, lo dejo como ejercicio para quien tenga inquietudes.

ARBOLES.

Vamos allá. En primer lugar, veamos de donde viene el nombrecito. En las listas, después de cada elemento venía otro (o ninguno, si habíamos llegado al final). Pero también nos puede interesar tener varias posibilidades después de cada elemento, 3 por ejemplo. De cada uno de estos 3 saldrían otros 3, y así sucesivamente. Obtendríamos algo que recuerda a un árbol: un tronco del que nacen 3 ramas, que a su veces se subdividen en otras 3 de menor tamaño, y así sucesivamente hasta llegar a las hojas.

Pues eso será un árbol: una estructura dinámica en la que cada nodo (elemento) puede tener más de un "siguiente". Nos centraremos en los árboles binarios, en los que cada nodo puede tener un hijo izquierdo, un hijo derecho, ambos o ninguno (dos hijos como máximo).

Para puntualizar aun más, aviso que trataremos los árboles binarios de búsqueda, en los que tenemos prefijado un cierto orden, que nos ayudará a encontrar un cierto dato dentro de un árbol con mucha rapidez.

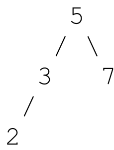
¿Y como es este "orden prefijado"? Sencillo: para cada nodo tendremos que:

- la rama de la izquierda contendrá elementos menores.
- la rama de la derecha contendrá elementos mayores.

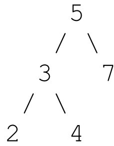
¿Asusta? Con un ejemplo seguro que no: Vamos a introducir en un árbol binario de búsqueda los datos 5,3,7,2,4,8,9

```
Primer número:  5 (directo)

        5

Segundo número: 3 (menor que 5)

        5
       /
      3

Tercer número: 7 (mayor que 5)

        5
       / \
      3   7
```

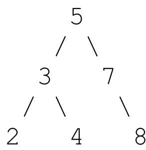
```
Cuarto: 2 (menor que 5, menor que 3)
```



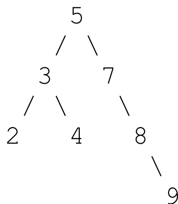
```
Quinto: 4 (menor que 5, mayor que 3)
```



```
Sexto: 8 (mayor que 5, mayor que 7)
```



```
Séptimo: 9 (mayor que 5, mayor que 7, mayor que 8)
```



¿Y qué **ventajas** tiene esto? Pues la rapidez: tenemos 7 elementos, lo que en una lista supone que si buscamos un dato que casualmente está al final, haremos 7 comparaciones; en este árbol, tenemos 4 alturas => 4 comparaciones como máximo.

Y si además hubiéramos "**equilibrado**" el árbol (irlo recolocando, de modo que siempre tenga la menor altura posible), serían 3 alturas.

Esto es lo que se hace en la práctica cuando en el árbol se va a hacer muchas más lecturas que escrituras: se reordena internamente después de añadir cada nuevo dato, de modo que la altura sea mínima en cada caso.

De este modo, el número máximo de comparaciones que tendríamos que hacer sería log2(n), lo que supone que si tenemos 1000 datos, en una lista podríamos llegar a tener que hacer 1000 comparaciones, y en un arbol binario, log2(1000) => 10 comparaciones como máximo. La ganancia es clara, ¿verdad?

No vamos a ver cómo se hace eso de los "equilibrados", que considero que sería propio de un curso de programación más avanzado, o incluso de uno de

"Tipos Abstractos de Datos" o de "Algorítmica", y vamos a empezar a ver rutinas para manejar estos árboles binarios de búsqueda.

Recordemos que la idea importante es todo dato menor estará a la izquierda del nodo que miramos, y los datos mayores estarán a su derecha.

Ahora la estructura de cada **nodo** (dato) será:

```
type
  TipoDato = string[10];  { Vamos a guardar texto, por ejemplo }
  Puntero = ^TipoBase;    { El puntero al tipo base }
  TipoBase = record       { El tipo base en sí: }
    dato: TipoDato;       { - un dato }
    hijoIzq: Puntero;     { - puntero a su hijo izquierdo }
    hijoDer: Puntero;     { - puntero a su hijo derecho }
  end;
```

Y las rutinas de inserción, búsqueda, escritura, borrado, etc., podrán ser recursivas. Como primer ejemplo, la de **escritura** de todo el árbol (la más sencilla) sería:

```
procedure Escribir(punt: puntero);
  begin
  if punt <> nil then { Si no hemos llegado a una hoja }
  begin
    Escribir(punt^.hijoIzq); { Mira la izqda recursivamente }
    write(punt^.dato); { Escribe el dato del nodo }
    Escribir(punt^.hijoDer); { Y luego mira por la derecha }
  end;
end;
```

Si alguien no se cree que funciona, que coja lápiz y papel y lo compruebe con el árbol que hemos puesto antes como ejemplo. Es MUY IMPORTANTE que este procedimiento quede claro antes de seguir leyendo, porque los demás serán muy parecidos.

La rutina de **inserción** sería:

```
procedure Insertar(var punt: puntero; valor: TipoDato);
begin
  if punt = nil then        { Si hemos llegado a una hoja }
  begin
    new(punt);              { Reservamos memoria }
    punt^.dato := valor;    { Guardamos el dato }
    punt^.hijoIzq := nil;   { No tiene hijo izquierdo }
    punt^.hijoDer := nil;   { Ni derecho }
  end
  else { Si no es hoja }
    if punt^.dato > valor              { Y encuentra un dato mayor }
      Insertar(punt^.hijoIzq, valor)   { Mira por la izquierda }
```

```
    else                            { En caso contrario (menor) }
      Insertar(punt^.hijoDer, valor) { Mira por la derecha }
end;
```

Y finalmente, la de **borrado** de todo el árbol, casi igual que la de escritura:

```
procedure BorrarArbol(punt: puntero);
begin
  if punt <> nil then { Si queda algo que borrar }
  begin
    BorrarArbol(punt^.hijoIzq); { Borra la izqda recursivamente }
    dispose(punt); { Libera lo que ocupaba el nodo }
    BorrarArbol(punt^.hijoDer); { Y luego va por la derecha }
  end;
end;
```

Sólo un comentario: esta última rutina es **peligrosa**, porque indicamos que "punt" está libre y después miramos cual es su hijo izquierdo (después de haber borrado la variable). Esto funciona en Turbo Pascal porque marca esa zona de memoria como disponible pero no la borra físicamente, pero puede dar problemas con otros compiladores o si se adapta esta rutina a otros lenguajes (como C). Una forma más **segura** de hacer lo anterior sería memorizando lo que hay a la derecha antes de borrar el nodo central:

```
procedure BorrarArbol2(punt: puntero);
var
  derecha: puntero; { Aquí guardaremos el hijo derecho }
begin
  if punt <> nil then { Si queda algo que borrar }
  begin
    BorrarArbol2(punt^.hijoIzq); { Borra la izqda recursivamente }
    derecha := punt^.hijoDer; { Guardamos el hijo derecho <=== }
    dispose(punt); { Libera lo que ocupaba el nodo }
    BorrarArbol2(derecha); { Y luego va hacia por la derecha }
  end;
end;
```

O bien, simplemente, se pueden borrar recursivamente los dos hijos antes que el padre (ahora ya no hace falta ir "en orden", porque no estamos leyendo, sino borrando todo):

```
procedure BorrarArbol(punt: puntero);
begin
  if punt <> nil then { Si queda algo que borrar }
  begin
    BorrarArbol(punt^.hijoIzq); { Borra la izqda recursivamente }
    BorrarArbol(punt^.hijoDer); { Y luego va hacia la derecha }
    dispose(punt); { Libera lo que ocupaba el nodo }
  end;
end;
```

Finalmente, vamos a juntar casi todo esto en un ejemplo "que funcione":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de árboles    }
{    binarios de búsqueda  }
{    ARBOL.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

type

  TipoDato = integer;     { Vamos a guardar texto, por ejemplo }

  Puntero = ^TipoBase;       { El puntero al tipo base }
  TipoBase = record          { El tipo base en sí: }
    dato:    TipoDato;       {   - un dato }
    hijoIzq: Puntero;        {   - puntero a su hijo izquierdo }
    hijoDer: Puntero;        {   - puntero a su hijo derecho }
  end;

procedure Escribir(punt: puntero);
begin
  if punt <> nil then           { Si no hemos llegado a una hoja }
    begin
    Escribir(punt^.hijoIzq);    { Mira la izqda recursivamente }
    write(punt^.dato, ' ');     { Escribe el dato del nodo }
    Escribir(punt^.hijoDer);    { Y luego mira por la derecha }
    end;
end;

procedure Insertar(var punt: puntero; valor: TipoDato);
begin
  if punt = nil then           { Si hemos llegado a una hoja }
    begin
    new(punt);                 { Reservamos memoria }
    punt^.dato := valor;       { Guardamos el dato }
    punt^.hijoIzq := nil;      { No tiene hijo izquierdo }
    punt^.hijoDer := nil;      { Ni derecho }
    end
  else                                  { Si no es hoja }
    if punt^.dato > valor               { Y encuentra un dato mayor }
    then
      Insertar(punt^.hijoIzq, valor)    { Mira por la izquierda }
    else                                { En caso contrario (menor) }
      Insertar(punt^.hijoDer, valor)    { Mira por la derecha }
end;

{ Cuerpo del programa }
```

```
var
  arbol1: Puntero;

begin
  arbol1 := nil;
  Insertar(arbol1, 5);
  Insertar(arbol1, 3);
  Insertar(arbol1, 7);
  Insertar(arbol1, 2);
  Insertar(arbol1, 4);
  Insertar(arbol1, 8);
  Insertar(arbol1, 9);
  Escribir(arbol1);
end.
```

Nota: en versiones anteriores de este fuente, la variable se llamaba "arbol". En la versión 3.5.1 del curso, he cambiado esta variable por "arbol1", dado que Tmt Pascal Lite protesta si usamos alguna variable que se llame igual que el nombre del programa (avisa de que estamos usando dos veces un identificador: "duplicate identifier").

Tema 13.5: Ejercicios.

Ejercicio propuesto: *Implementar una pila de strings[20]*
Ejercicio propuesto: *Implementar una cola de enteros*
Ejercicio propuesto: *Implementar una lista doblemente enlazada que almacene los datos leídos de un fichero de texto (mejorando el lector de ficheros que vimos)*
Ejercicio propuesto: *Implementar una lista simple que almacene los datos leídos de un fichero de texto, pero cuyos elementos sean otras listas de caracteres, en vez de strings de tamaño fijo*
Ejercicio propuesto: *Añadir la función "buscar" a nuestro árbol binario, que diga si un dato que nos interesa pertenece o no al árbol (TRUE cuando sí pertenece; FALSE cuando no).*
Ejercicio propuesto: *¿Cómo se borraría un único elemento del árbol?*

N.

Curso de Pascal. Tema 14: Creación de gráficos.

Primero vamos a ver una introducción a lo que son los conceptos fundamentales sobre eso de los gráficos.

Texto frente a gráficos. Hasta ahora hemos trabajado en modo texto: escribíamos letras en posiciones concretas de la pantalla, para formar textos que el usuario de nuestros programas pudiera leer. Esto de escribir letras está muy bien, y es algo que se puede hacer casi desde la prehistoria de la informática...

¿Pero que pasa si queremos mostrar los resultados de la empresa en forma de barras? Podemos salir del paso haciendo

```
XX
XX XX
XX XX XX
XX XX XX
XX XX XX
```

Como somos unos chicos avispados, hemos cubierto el expediente. Pero ahora resulta que nuestro incansable jefe ahora nos dice que quiere que le mostremos las tendencias de ventas...

¡Vaya, ahora tendríamos que dibujar líneas! Podemos hacerlo a base de puntos (.) en vez de X. Nuevamente nuestro ingenio nos habría salvado, pero el resultado sería muy pobre.

Para mejorar esto entran en juego los modos gráficos. En ellos, en vez de dibujar símbolos que corresponden a letras, podemos acceder a cada uno de los puntos que forman la pantalla.

¿Y cómo se hace esto? Pues la idea es casi evidente ;-) : en algunos ordenadores antiguos, las letras estaban formadas por una serie de puntos, con un tamaño uniforme, por ejemplo 5 puntos de ancho y 7 de alto:

```
XXXX. Las X indican los puntos que se dibujaban y los . los
X..X. que no se dibujaban. O sea, que el ordenador sí que
XXXX. accedía a los puntos individuales de la pantalla.
X..X.
X..X. Entonces apenas hacía falta que mejorase un poco la
..... tecnología para que nos dejase hacerlo a nosotros.
.....
```

En los tiempos de ordenadores como el famoso Spectrum, el Commodore 64, los MSX, Dragon, Oric, Amstrad CPC, etc., ya era habitual que se nos permitira acceder a puntos individuales de la pantalla.

Muchos de estos ya tenían caracteres de 8x8, más nítidos que el de 5x7 del ejemplo anterior, y que a su vez suponen para nosotros más puntos en los que poder dibujar.

Y más puntos en una pantalla del mismo tamaño quiere decir que los puntos serán más pequeños. Y si los puntos que forman una imagen son más pequeños, esta imagen resultará más nítida.

Por ejemplo, en los Amstrad CPC, el modo habitual de trabajo era de 25 filas y 40 columnas, con caracteres de 8x8, y nos permitía acceder individualmente a cada uno de esos puntos, es decir, 25x8=200 puntos verticales y 40x8=320 puntos horizontales. La diferencia es abismal: en modo texto accedemos a 25x40 = 1000 posiciones distintas de la pantalla, mientras que en modo gráfico a 320x200 = 64000 posiciones. Ahora nuestro jefe quedaría bastante más contento...

¿Y si se ve mejor, por qué no trabajamos siempre en modo gráfico? Por cuestiones de **velocidad** y **memoria**, principalmente: hemos visto que en un modo gráfico sencillo como éste hay 64 veces más puntos que letras, lo que supone gastar más memoria para guardar la información que estamos mostrando, necesitar más velocidad para representar los resultados sin que el usuario se aburra (especialmente si son imágenes en movimiento), etc. Por ello, se van viendo más cosas en modo gráfico a medida que los ordenadores van siendo más **potentes**.

De hecho, en algunos de estos primeros ordenadores personales (incluso en los primeros PC) debíamos elegir entre modo gráfico o de texto, según nuestros intereses, ya que no podían ser **simultáneos**.

Por ejemplo, en un PC con tarjeta **CGA** (ya veremos después lo que es esto), teníamos que elegir entre:

- Modo texto, en 80x25, con 16 colores (y la posibilidad de tener varias páginas guardadas en memoria).
- Modo texto, en 40x25, con 16 colores y el doble de páginas.
- Modo gráfico, con 640x200 puntos, pero en blanco y negro.
- Modo gráfico, con 320x200 puntos en 4 colores.

¿Y si no son simultáneos quiere decir que cuando estamos en modo gráfico no podemos escribir texto? No, el texto se puede "dibujar" también, y el ordenador lo suele hacer automáticamente (según casos) cuando estamos en un modo gráfico y le pedimos que escriba texto. Pero entonces se escribe más despacio, y con menos colores para elegir.

¿Y qué es un **pixel**? Seguro que habeis oído esa palabreja más de una vez. Pues no es más que un punto en la pantalla gráfica. El nombre viene de la abreviatura de dos palabras en inglés: "picture cell" o "picture element" (según el autor que se consulte).

¿Y eso de una **tarjeta gráfica**? Pues la idea tampoco es complicada: en los primeros ordenadores domésticos, como el Spectrum o el CPC, la "parte" encargada de mostrar el texto o los gráficos en la pantalla era una parte más del ordenador, que no se podía cambiar.

En cambio los PC son ordenadores modulares, muy fáciles de ampliar, que sí que nos permiten cambiar la "parte" que se encarga de gráficos o texto (y otras "partes" -casi todas- también se pueden cambiar) para poner otra más potente: más rápida, o que represente más colores, o más puntos.

Esta "parte" que se puede quitar tiene la forma de una placa de circuito impreso, con sus "chips" y sus cosas raras de esas Y esta placa se inserta en otra grande, que es la que realmente "piensa" en el ordenador (la placa madre).

Pues eso es la tarjeta gráfica: esa placa que podemos quitar o poner con una relativa facilidad, y que es la que se encarga de controlar todo el acceso a la pantalla.

En la historia del PC ha habido varios estándares en tarjetas gráficas o de texto. Los más importantes han sido:

- MDA: 80x25 caracteres en texto, blanco y negro (con negrita y subrayado). Sin gráficos.
- CGA: 80x25 o 40x25 en texto, con 16 colores. 640x200 en monocromo o 320x200 en 4 colores, para gráficos.
- Hercules: Como MDA en texto. 720x350 en gráficos, blanco y negro.
- EGA: Compatible con CGA, pero añade los modo gráficos: 320x200, 640x200 y 640x350 en 16 colores.
- MCGA: Compatible con EGA, pero añade: 320x200 en 256 colores y 640x480 monocromo.
- VGA: Compatible con MCGA, y añade: 640x480 en 16 colores.
- SuperVGA: Compatible con VCGA, y añaden (según cada modelo): Modos de 256 colores en 640x480, 800x600, 1024x768, 1280x1024..Modos de 32000, 64000 o 16 millones de colores.

¿Y qué es eso de un **driver**? Es un intermediario entre nosotros y la tarjeta gráfica: hay una gran cantidad de tarjetas gráficas (las anteriores son sólo las más importantes), y no todas se programan igual. Así, que para nosotros no necesitemos aprender qué hay que hacer para dibujar un punto en azul en cualquier pantalla, solemos tener "ayuda".

Por ejemplo, Borland distribuye con Turbo Pascal (y Turbo C/C++) unos drivers, los **BGI** (Borland Graphics Interface), que son los que se encargan de acceder a la pantalla. Así, nosotros no tenemos más que decir "dibuja un punto", y el BGI sabrá como hacerlo, en función de la tarjeta gráfica que haya

instalada, del modo de pantalla en el que estemos trabajando (cantidad de puntos y de colores disponibles), etc.

Pero todo eso lo veremos en el apartado siguiente...

14.2: Nuestro primer gráfico con Turbo Pascal.

Vamos a comentar cómo crear gráficos con **Turbo Pascal**, centrándonos en la versión 5.0 y superiores. Lo que veremos se podrá aplicar a otros compiladores posteriores, como **Free Pascal**.

Vamos a hacerlo empleando los "**BGI**" (los drivers originales de Bodland). Otra opción, que veremos como ampliación, es "saltarnoslos" y acceder nosotros mismos a la memoria de pantalla, con lo que podemos conseguir una mayor velocidad y un menor tamaño del programa ejecutable (.EXE) resultante, pero a cambio de una mayor dificultad a la hora de programarlo, y de menor portabilidad: nuestro programa no se podrá llevar con tanta facilidad a otro sistema.

Volviendo a lo que nos interesa hoy, los BGI nos permiten realizar muchísimas funciones, como puede ser dibujar puntos en la pantalla, líneas, rectángulos, círculos, elipses, arcos, polígonos, rellenar superficies, cambiar la paleta de colores, usar distintos estilos de texto, etc.

Para acceder a todo ello, necesitamos en primer lugar hacer una llamada a la **unidad graph** desde nuestro programa ("uses graph").

Ya en el cuerpo del programa, debemos comenzar por inicializar el sistema de gráficos mediante el procedimiento **InitGraph**. Este tiene como parámetros los siguientes:

- El código del driver concreto que vamos a usar:
    - HercMono si es una tarjeta Hercules.
    - Cga si es CGA, claro.
    - EgaMono si es EGA con monitor monocromo.
    - Ega si es EGA con monitor color.
    - Vga si es MCGA o VGA.
    - (otras menos habituales).
    - Detect si queremos que detecte cual está instalada.

- El modo de pantalla que vamos a usar para ese driver:
    - VgaHi es el modo de alta resolución de VGA: 640x480, 16 colores.

  - EgaHi es el modo de alta resolución de EGA: 640x350, 16 colores.
  - CgaHi es el modo de alta resolución de CGA: 640x200, 2 colores.
  - EgaMonoHi es el único modo de Ega monocromo: 640x350, b/n.
  - HercMonoHi es el único modo de Hercules: 720x350, b/n.
  - .. (distintos modos para cada tarjeta).

- El directorio en el que están los drivers.

Se debe indicar el directorio, porque los drivers están en forma de unos ficheros de extensión BGI: HERC.BGI, CGA,BGI, EGAVGA.BGI, etc., que el programa buscará en el momento de ejecutarlo.

No he puesto toda la lista de drivers ni de modos porque creo que tampoco tiene demasiado sentido: uno siempre acaba usando uno o dos nada más, y se los aprende. Sólo he dejado los más habituales. Si alguien quiere cotillear más, la mejor opción es recurrir a la ayuda on-line del compilador. Basta escribir Graph y pulsar Ctrl+F1 para que nos cuente montones y montones de cosas sobre la unidad Graph.

Si usamos el código "**detect**" para que el compilador intente detectar la tarjeta que hay instalada en el ordenador en el cual esté funcionando el programa, no hace falta indicar el modo, sino que él tomará el de mayor resolución que permita esa tarjeta.

En este caso, no tendremos claro el número de puntos que puede haber en la pantalla, así que podemos preguntarle al BGI. Tenemos las funciones GetMaxX y GetMaxY, que nos dicen el número de puntos horizontales y verticales que hay en el modo de pantalla actual. También podemos saber el número de colores con GetMaxColor.

Después de terminar de usar la pantalla en modo gráfico, deberemos usar la orden **CloseGraph**. ¿Parece que aún no sabemos nada y que esto es lioso? Todo lo contrario. Con esto y poco más ya podemos hacer nuestra primera prueba en modo gráfico:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer ejemplo de     }
{    gráficos usando BGI   }
```

```
{    GRAF1.PAS               }
{                            }
{  Este fuente procede de    }
{  CUPAS, curso de Pascal    }
{  por Nacho Cabanes         }
{                            }
{  Comprobado con:           }
{    - Turbo Pascal 7.0      }
{    - Free Pascal 2.2.0w    }
{--------------------------}

program PrimerGrafico;

uses Graph;                       { Va a usar la librería gráfica de TP
}

var
  Driver, Modo: Integer;              { Pues el driver y el modo, claro
}

begin
  Driver := Vga;                                      { Para pantalla VGA
}
  Modo := VgaHi;                              { Modo 640x480, 16 colores
}
  InitGraph(Driver, Modo, 'c:\tp\bgi');                   { Inicializamos
}
  Line(0, 0, 320, 240);                             { Dibujamos una línea
}
  Readln;                                                      { Esperamos
}
  CloseGraph                                                  { Y se acabó
}
end.
```

Este ejemplo funciona sin ningún cambio también con Free Pascal, incluso si compilamos para Windows (probado con la versión 2.0.4 para Windows). Eso sí, en este caso, no haría falta indicar la ruta 'c:\tp\bgi', porque no buscará ningún fichero BGI en ninguna parte de nuestro disco, sino que los enlaza directamente cuando se compila.

14.3: Más órdenes gráficas.

Las **órdenes gráficas** más habituales son:

- **PutPixel**(x,y,color): dibuja un punto en un color dado.
- **Line**(x1,y1,x2,y2): dibuja una linea con el color y estilo predefinido.
- **SetColor**(color): escoge el color predefinido.
- **SetLineStyle**(estilo, patron, color): cambia el estilo con que se dibujarán las líneas. Para elegir el estilo de línea, tenemos predefinidas constantes como SolidLn (línea continua), DottedLn (de puntos), etc. Para el grosor también tenemos constantes: NormWidth (normal), ThickWidth (gruesa).
- **Circle**(x,y,radio): dibuja un circulo con el color predefinido.

- **Ellipse**(x,y,anguloIni, anguloFin, radioX, radioY): dibuja una elipse o un arco de elipse.
- **Arc**(x,y,anguloIni, anguloFin, radio): dibuja un arco circular.
- **Rectangle**(x1,y2,x2,y2): dibuja un rectángulo con el color y el tipo de línea predefinido. Los puntos son las esquinas superior izquierda e inferior derecha
- **Bar**(x1,y1,x2,y2): dibuja un rectángulo relleno con el color de relleno (y el patrón de relleno, si es el caso) dados por SetFillStyle y SetFillPattern.
- **SetFillStyle**(patron, color): elige el patrón y color de relleno.
- **SetFillPattern**(patron, color): permite redefinir el patrón con que se rellenarán las figuras.
- **Bar3D**(x1,y1,x2,y2,profund,tapa): dibuja un rectángulo relleno, con una cierta profundidad en 3D (si la profundidad es 0, equivale a "Bar" pero como una barra exterior.
- **FloodFill**(x,y,borde): rellena una zona cerrada, cuyo borde sea de un cierto color. Usa el color y patrón de relleno actual.
- **OutText**(texto): escribe un texto en la posición actual del cursor.
- **OutTextXY**(x,y,texto): escribe un texto en una posición dada.
- **SetTextStyle**(fuente,direccion,tamaño): elige la fuente (tipo de letra), la direccion y el tamaño con que se escribirán los textos.
- (etc)

Creo que no tiene sentido que me enrolle con ellas. Voy a poner sólo un **ejemplo** que use las más frecuentes:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Funciones gráficas    }
{    más frecuentes        }
{    DEMOGRAF.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program DemoGraf;

uses graph;

var
  driver, modo: integer;
  x, y: integer;
  i: integer;

begin
  driver := detect;
  initgraph(driver, modo, '');
```

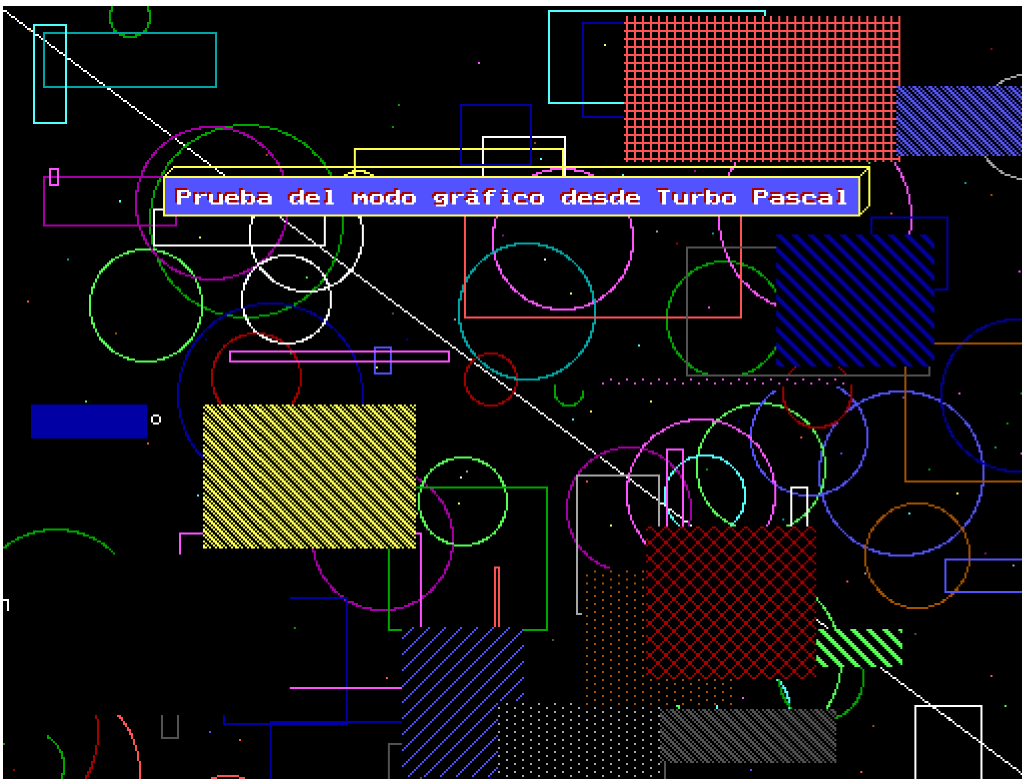
```
    randomize;                    { Comienzo a generar números al azar }
    line(0,0,639,479);            { Línea fija }
    for i := 1 to 200 do          { 200 puntos }
      putpixel(random(640), random(480), random(15)+1);
    for i := 1 to 50 do           { 50 círculos }
      begin
      setcolor(random(15)+1);
      circle(random(640), random(480), random(50));
      end;
    for i := 1 to 30 do           { 30 rectángulos }
      begin
      x := random(500);
      y := random(400);
      setcolor(random(15)+1);
      rectangle(x, y, x+random(140), y+random(80));
      end;
    for i := 1 to 15 do           { 15 rectángulos }
      begin
      x := random(500);
      y := random(400);
      setcolor(random(15)+1);
      setfillstyle(random(11), random(15)+1);
      bar(x, y, x+random(140), y+random(80));
      end;
                                  { Recuadro con texto en su interior }
    setfillstyle(SolidFill, LightBlue);
    SetColor(Yellow);
    Bar3D(93,93,440,112, 5, true);
    setcolor(Red);
    OutTextXY(99,99,'Prueba del modo gráfico desde Turbo Pascal');
    setcolor(White);
    OutTextXY(100,100,'Prueba del modo gráfico desde Turbo Pascal');
    readln;
    closegraph;
  end.
```

Espero que todo se entienda por si solo. Los principales cambios (además del hecho de emplear algunas órdenes nuevas para nosotros) son los siguientes:

- En vez de decirle que quiero usar una tarjeta VGA y el modo de alta resolución VgaHi, le pido que detecte qué tarjeta gráfica tengo instalada, con la orden **driver := detect;**
- Aun así, en todo el programa estoy suponiendo que el tamaño de mi pantalla es de 640x480 puntos, y puede que esto no sea cierto. Sería preferible usar **GetMaxX** en vez de 640 y GetMaxY en vez de 480.
- Con **Randomize** comienzo a generar números aleatorios (al azar). **Random(640)** es una función que me devuelve un número al azar entre 0 y 639. Ambas se ven con más detalle en la ampliación 1.
- Para los colores, he supuesto que tengo 16 colores disponibles (del 0 al 15) y nuevamente habría sido más correcto usar **GetMaxColor** para saber con certeza cuantos puedo emplear. En cualquier caso, estos colores irán del 0 al 15; como el 0 es el negro, que no se ve, dibujo los puntos y las líneas tomando como color random(15)+1, que me

dará un color entre 0 y 14, al que luego sumo 1: un color entre 1 y 15, con lo que consigo "esquivar" el negro.

Un **comentario** obligado: Nuevamente, este programa funciona sin cambios en TMT y FPK/Free Pascal, pero hay un posible peligro: estamos detectando la tarjeta gráfica que existe en nuestro ordenador y dejando que el propio compilador elija el modo gráfico dentro de ese tipo de tarjeta. Esto es peligroso con estos compiladores, que reconocen tarjetas más modernas y permiten acceder a los modos VESA, con más puntos y/o más colores. Así, este programa probado con mi ordenador y con TMT cambia a un modo que no es el de 640x480 puntos con 16 colores que usaría Turbo Pascal, sino a otro con mayor número de colores, mientras que FPK deja mi pantalla "en negro", posiblemente porque esté cambiando a un modo de pantalla de mi alta resolución, que sí permita mi tarjeta gráfica pero no mi pantalla. Con Free Pascal sí funciona correctamente, aunque reserva un tamaño de pantalla mayor de 640x480. Su apariencia es ésta:



Podemos evitar este tipo de posibles problemas no dejando que el compilador detecte cual es nuestra tarjeta gráfica, sino obligando nosotros, como hicimos en el ejemplo anterior, cambiando esta línea

```
driver := detect;
```

por estas dos

```
Driver := Vga; { Para pantalla VGA }
Modo := VgaHi; { Modo 640x480, 16 colores }
```

14.4: Un sencillo programa de dibujo.

Otro ejemplo de programa, en este caso más sencillo, pero espero que también más útil, sería un programita sencillo de **dibujo**:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Programa sencillo de  }
{    dibujo                }
{    DIB.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program dib;

uses graph, crt;

var
  driver, modo: integer;
  tecla: char;
  x, y: integer;

begin
  driver := detect;
  initgraph(driver, modo, '');
  x := 320;    { Comienzo en el centro }
  y := 240;
  repeat
    putpixel(x,y, 15);
    tecla := readkey;
    case tecla of
      '8': y := y-1;
      '4': x := x-1;
      '6': x := x+1;
      '2': y := y+1;
    end;
  until tecla = 'f';
  closegraph;
end.
```

Creo que la idea en sí es muy sencilla: si se pulsan las teclas 2, 4, 6, 8 (del teclado numérico), el puntito que dibuja se va desplazando por la pantalla, y dibujando un punto en el color 15 (que normalmente es el blanco). Cuando se pulse "f", acaba el programa.

14.5: Mejorando el programa de dibujo.

Por supuesto, este programa es muy muy mejorable. La primera mejora casi evidente es poder pintar en más de un **color**. Podemos retocar el programa para que lo haga, y de paso comprobamos el tamaño real de la pantalla y el número de colores disponible:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Programa sencillo de  }
{    dibujo (en varios     }
{    colores)              }
{    DIB2.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program dib2;

uses graph, crt;

var
  driver, modo: integer;
  tecla: char;
  x, y, c: integer;

begin
  driver := detect;
  initgraph(driver, modo, '');
  x := getMaxX div 2;    { Comienzo en el centro }
  y := getMaxY div 2;
  c := getMaxColor;      { Y con el máximo color permitido en este
modo }
  repeat
    putpixel(x,y, c);
    tecla := readkey;
    case tecla of
      '8': y := y-1;
      '4': x := x-1;
      '6': x := x+1;
      '2': y := y+1;
      'c': begin
           c := c + 1;
           if c>getMaxColor then c := 1;
           end;
    end;
  until tecla = 'f';
  closegraph;
end.
```

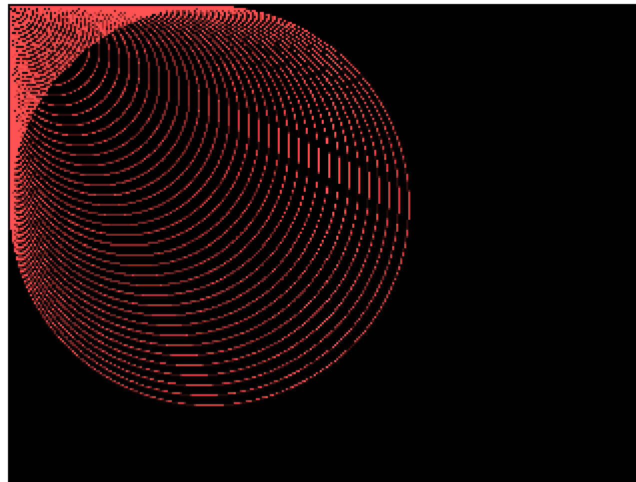
¿Otras mejoras posibles? Que se pueda borrar un punto o la pantalla entera, que se puedan dibujar líneas, rectángulos y otras figuras... eso ya queda a la imaginación de cada uno.

Y todo esto podría seguir y seguir... Lo mejor es que, si os interesa, miréis la ayuda y practiquéis con los ejemplos que incluye, o bien que miréis el **BGIDEMO** que incluye Turbo Pascal, que pone a prueba casi todas las posibilidades gráficas de este compilador..

Lo que sí voy a hacer es poner yo algún ejemplo que YO considere interesante porque aporte algo más o menos curioso... }:-)

14.6: Un efecto vistoso utilizando círculos.

Para empezar, vamos a dibujar algo parecido a un **cono**, formado por varios círculos. Buscamos un resultado similar a éste:



Y lo podemos conseguir así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Cono formado por      }
{    varios círculos       }
{    GRAF2.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.2.0w  }
{--------------------------}
```

```
program SegundoGrafico;

uses Graph;                    { Va a usar la librería gráfica de TP
}

var
  Driver, Modo: Integer;          { Pues el driver y el modo, claro
}
  bucle: word;                    { Para bucles, ya se verá por qué
}

begin
  Driver := Vga;                             { Para pantalla VGA
}
  Modo := VgaHi;                       { Modo 640x480, 16 colores
}
  InitGraph(Driver, Modo, 'c:\tp\bgi');            { Inicializamos
}
  SetColor(LightRed);                                { Color rojo
}
  for bucle := 1 to 40 do               { Dibujaremos 40 círculos
}
    circle ( bucle*5, bucle*5, bucle*5 );
  Readln;                                             { Esperamos
}
  CloseGraph                                         { Y se acabó
}
end.
```

Hemos dibujado 40 círculos, cuyas coordenadas x e y, y cuyo radio son iguales en cada círculo. El efecto es vistoso, pero se puede **mejorar**...

14.7. Jugando con la paleta de colores.

El efecto anterior se puede mejorar. Podemos cambiar la **paleta de colores** para que quede más vistoso aún. Con la orden **SetRGBPalette** podemos fijar los componentes de rojo, verde y azul de un cierto color base. El formato es SetRGBPalette(Color, R,G,B) donde

- Color va de 0 a 15, en este caso, y de 0 a GetMaxColor en el caso general (como ya hemos comentado, GetMaxColor es una función que nos dice el número de colores disponibles según el driver y modo que estemos usando.
- R,G,B van de 0 (negro) a 63 (intensidad máxima de cada uno de ellos).

Así, nuestro "cono" retocado quedaría:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Cono de círculos con  }
```

```
{    los colores cambiados }
{    GRAF2B.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program SegundoGraficoRetocado;

uses
  Graph,                       { Va a usar la librería gráfica de TP
}
  Crt;                              { y la CRT para tiempo y teclado
}

var
 Driver, Modo: Integer;             { Pues el driver y el modo, claro
}
 bucle: word;                       { Para bucles, ya se verá por qué
}
 Intensidad: byte;                            { Intensidad del color
}
 Incremento: ShortInt;                                { Será +1 ó -1
}
 Tecla: char;                                { La tecla que se pulse
}

begin
 Driver := Vga;                                 { Para pantalla VGA
}
 Modo := VgaHi;                          { Modo 640x480, 16 colores
}
 InitGraph(Driver, Modo, 'c:\tp\bgi');                { Inicializamos
}
 SetColor(Red);                                          { Color rojo
}
 for bucle := 1 to 40 do                    { Dibujaremos 40 círculos
}
   circle ( bucle*5, bucle*5, bucle*5 );
 Intensidad := 63;                          { Empezaremos desde rojo
}
 Incremento := -1;                                  { y disminuyendo
}
 while not keypressed do               { Mientras no se pulse tecla
}
   begin
   SetRGBPalette(Red, Intensidad, 0, 0);            { Cambia la paleta
}
   Intensidad := Intensidad + Incremento;       { Y la próxima intens.
}
   if Intensidad = 0 then                       { Si llegamos a int. 0
}
     Incremento := 1;                             { Deberemos aumentar
}
   if Intensidad = 63 then                           { Si es la máxima
}
```

```
      Incremento := -1;                                  { Disminuiremos
}
    delay(20);                                         { Pausa de 20 mseg
}
    end;
  Tecla := ReadKey;                        { Abosorbemos la tecla pulsada
}
  CloseGraph                                                { Y se acabó
}
end.
```

14.8: Dibujando líneas.

Otro efecto curioso es el que ocurre cuando cambiamos la forma de dibujar los puntos. El siguiente ejemplo dibuja **líneas** desde el origen (0,0) a cada uno de los puntos de la línea inferior.

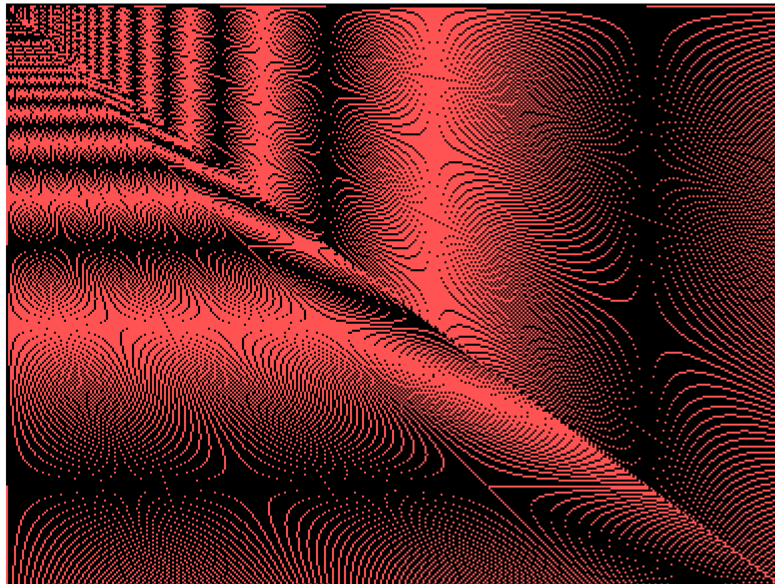
```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Líneas desde el       }
{    punto (0,0)           }
{    GRAF3.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.2.0w  }
{--------------------------}

program TercerGrafico;

uses Graph;                       { Va a usar la librería gráfica de TP
}

var
  Driver, Modo: Integer;            { Pues el driver y el modo, claro
}
  bucle: word;                      { Para bucles, ya se verá por qué
}

begin
  Driver := Vga;                                 { Para pantalla VGA
}
  Modo := VgaHi;                          { Modo 640x480, 16 colores
}
  InitGraph(Driver, Modo, 'c:\tp\bgi');                 { Inicializamos
}
  SetColor(LightRed);                                      { Color rojo
}
  for bucle := 0 to 639 do                         { Dibujaremos lineas
}
    line (0,0,bucle,439);
```

```
  Readln;                                                     { Esperamos
}
  CloseGraph                                                  { Y se acabó
}
 end.
```

Sencillo, ¿no?

14.9. Otro efecto más vistoso.

Ahora veamos lo que ocurre si dibujamos los puntos de cada línea haciendo una operación **XOR** con los que ya existían, en vez de dibujarlos encima simplemente. El efecto es mucho más llamativo:

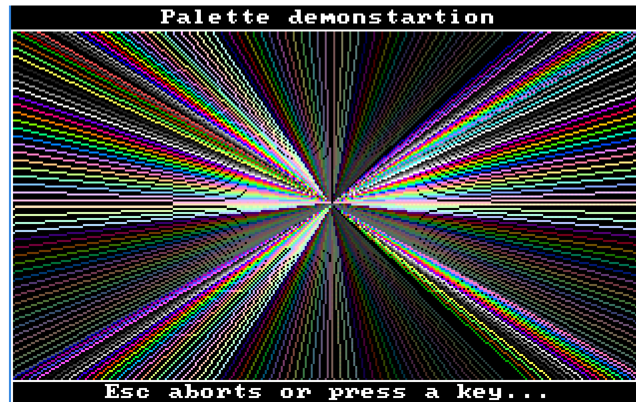


```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Líneas dibujadas con  }
{    XOR                   }
{    GRAF3B.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
```

```
{     - Turbo Pascal 7.0     }
{     - Free Pascal 2.2.0w   }
{--------------------------}

program TercerGraficoConXor;

uses Graph;                      { Va a usar la librería gráfica de TP
}

var
  Driver, Modo: Integer;            { Pues el driver y el modo, claro
}
  bucle: word;                      { Para bucles, ya se verá por qué
}

begin
  Driver := Vga;                                  { Para pantalla VGA
}
  Modo := VgaHi;                            { Modo 640x480, 16 colores
}
  InitGraph(Driver, Modo, 'c:\tp\bgi');                { Inicializamos
}
  SetColor(LightRed);                                     { Color rojo
}
  SetWriteMode(XORPut);                         { <-- Este es el cambio
}
  for bucle := 0 to 639 do                          { Dibujaremos lineas
}
    line (0,0,bucle,479);
  for bucle := 0 to 479 do                                      { y más
}
    line (0,0,639,bucle);
  Readln;                                                  { Esperamos
}
  CloseGraph                                             { Y se acabó
}
end.
```

Pues lo voy a dejar aquí. Espero que todo esto os haya dado una base para empezar a cotillear y, sobre todo, que haya hecho que os pique la curiosidad.

Así que ya sabeis, si os llama la atención esto de los gráficos, pues a experimentar. Insisto en que un ejemplo del que se puede aprender mucho es del BGIDEMO.PAS que trae Turbo Pascal: allí podreis ver con detalle cómo manejar los tipos de letra, copiar zonas de la pantalla, definir patrones para rellenar zonas, etc.:



14.10. Gráficos con FPK y TMT.

La gran mayoría de los ejemplos que hemos visto para Turbo Pascal deberían funcionar sin cambios en FPK Pascal y en TMT Pascal, pero estos no necesitan que los ficheros BGI estén en un cierto directorio, y nos permitirán acceder a modos gráficos con más puntos en pantalla y más colores, propios de tarjetas gráficas más avanzadas (siguiendo el estándar VESA).

Por ejemplo, el primer programa gráfico que hemos creado se podría reescribir para Tmt con ligeras modificaciones, así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer ejemplo de     }
{    graficos en TMT       }
{    GRAF1T.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - TMT Pascal 3.30     }
{--------------------------}

program PrimerGrafico;

uses Graph;                    { Usaremos la libreria grafica de TMT }

begin
 SetGraphMode($101);              { Inicializamos: modo 640x480, 256c
}
 Line(0, 0, 320, 240);                            { Dibujamos una linea
}
 Readln;                                                    { Esperamos
}
 CloseGraph                                                { Y se acabo
}
end.
```

Aquí hemos cambiado la orden "InitGraph" por la que se recomienda en TMT, que es "SetGraphMode", y que sólo espera el número de modo gráfico que nos interesa (insisto, no necesitamos indicar ningún directorio donde buscar los drivers). Los números de modos gráficos, definidos en el estándar VESA, pueden ser, por ejemplo:

```
 100h 640x400x256
 101h 640x480x256
 102h 800x600x16
 103h 800x600x256
 104h 1024x768x16
 105h 1024x768x256
 106h 1280x1024x16
 107h 1280x1024x256
```

para tarjetas que sigan el estandar VESA 1.0, y si siguen el estandar VESA 1.2, hay todavía más modos disponibles, con mayor número de colores:

```
 10Dh 320x200x32K
 10Eh 320x200x64K
 10Fh 320x200x16M
 110h 640x480x32K
 111h 640x480x64K
 112h 640x480x16M
 113h 800x600x32K
 114h 800x600x64K
 115h 800x600x16M
 116h 1024x768x32K
 117h 1024x768x64K
 118h 1024x768x16M
 119h 1280x1024x32K
 11Ah 1280x1024x64K
 11Bh 1280x1024x16M
```

Podríamos crearnos nosotros muestras constantes si nos interesa, con algo parecido a:

```
const
 VESA640x400x256 = $100;
 VESA640x480x256 = $101;
 (...)
```

En **FPK Pascal** sí se usa la misma orden InitGraph, pero la lista de drivers y de modos es bastante distinta: los "drivers" se referirán al número de colores que deseamos, y el "modo" se referirá a la resolución (número) de puntos en

pantalla. Tenemos definidas las siguientes constantes (tomado directamente de la documentación de FPK):

```
-- "Drivers" de pantalla:
D1bit = 11;
D2bit = 12;
D4bit = 13;
D6bit = 14; { 64 colors Half-brite mode - Amiga }
D8bit = 15;
D12bit = 16; { 4096 color modes HAM mode - Amiga }
D15bit = 17;
D16bit = 18;
D24bit = 19; { not yet supported }
D32bit = 20; { not yet supported }
D64bit = 21; { not yet supported }

-- "Modos" de pantalla:
detectMode = 30000;
m320x200 = 30001;
m320x256 = 30002; { amiga resolution (PAL) }
m320x400 = 30003; { amiga/atari resolution }
m512x384 = 30004; { mac resolution }
m640x200 = 30005; { vga resolution }
m640x256 = 30006; { amiga resolution (PAL) }
m640x350 = 30007; { vga resolution }
m640x400 = 30008;
m640x480 = 30009;
m800x600 = 30010;
m832x624 = 30011; { mac resolution }
m1024x768 = 30012;
m1280x1024 = 30013;
m1600x1200 = 30014;
m2048x1536 = 30015;
```

De modo que para entrar al modo de 640x480 con 256 colores haríamos:

```
driver : = D8bit;
modo : = m640x480;
initgraph ( driver, modo, '');
```

14.11. Incluir los BGI en el EXE.

Un último comentario: si alguien usa Turbo Pascal, pero no le gusta eso de que estén los ficheros con extensión .BGI por ahí siempre y depender de ellos, puede **incluirlos en el fichero EXE**. Demos un repasito rápido a la forma de hacerlo:

Primero debemos convertir el BGI en OBJ, que es algo que ya sí se podrá incluir dentro del EXE. Para conseguirlo, tecleamos:

```
binobj egavga.bgi egavga EgaVgaDriver
```

(BINOBJ es una utilidad que se incluye con las últimas versiones de Turbo Pascal). El formato en general es

```
BINOBJ  NombreBgi  NombreObj  NombrePúblico
```

Y para incluir este OBJ en el EXE, un programa sencillo sería:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Incluye BGI en el EXE }
{    (antes se debe usar   }
{    Binobj)               }
{    BGIEXE.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program BGIEXE;  { Introducir los BGI en el .EXE }

uses Graph;        { Las funciones gráficas, claro }

var
  GraphDriver, GraphMode, Error : integer;

procedure EgaVgaDriver; external;   {  El driver, en el EXE }
{$L EGAVGA.OBJ }
begin
                                         { Compruebo que el BGI está
bien }
  if RegisterBGIdriver(@EgaVgaDriver) < 0 then
    begin
    writeln('No se pudo inicializar el modo VGA');
    halt(1);
    end;
  GraphDriver := VGA;                       { Paso a VGA }
  GraphMode := VGAHi;                       { Modo 640x480x16 }
  InitGraph(GraphDriver, GraphMode, '');
  if GraphResult <> grOk then               { Si hay error, paro }
  begin
    writeln('No se pudo pasar a modo gr fico');
    halt(1);
  end;
  Line(0, 0, 639, 479);
  Readln;
  CloseGraph;
end.
```

Y si quereis ver cómo hacer gráficos desde Turbo Pascal sin necesidad de usar los BGI, mirad la "Ampliación 2" del curso, orientada principalmente al modo MCGA/VGA 320x200, 256 colores.

Por cierto, que también tenemos la posibilidad de usar otros BGI distintos de los que proporciona Borland. Por ejemplo, es fácil encontrar en alguna BBS o en CdRoms de shareware algún BGI para modos de 256 colores, o de mayor resolución en una SuperVGA. Si os llega a interesar, mirad la ayuda sobre la orden "InstallUserDriver".

Curso de Pascal. Tema 15: Servicios del DOS.

La unidad DOS nos permite acceder a muchos de los servicios que el sistema operativo pone a nuestra disposición, como son:

- Acceso a la fecha y la hora del sistema.
- Lectura de los ficheros que existen en un cierto directorio.
- Acceso a la fecha, hora y atributos de un archivo.
- Ejecución de otros programas.
- Llamada a interrupciones del DOS.
- Lectura de las variables de entorno.
- ...

Vamos a ver algunos **ejemplos** de su uso...

15.1: Espacio libre en una unidad de disco.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Espacio libre en la   }
{    unid. de disco actual }
{    ESPACIO.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program Espacio;  { Mira el espacio libre en la unidad de disco actual
}

uses Dos;
```

```
begin
  write(' Hay disponibles: ');
  write( DiskFree(0) div 1024 );         { Muestra el espacio libre }
  write(' K de ');
  write( DiskSize(0) div 1024 );         { y el espacio total }
  writeln(' K totales.');
end.
```

El 0 de DiskFree y DiskSize indica que se trata de la unidad actual. Un 1 se referiría a la A, un 2 a la B, un 3 a la C, y así sucesivamente. Si una unidad no existe, DiskSize devuelve -1.

15.2. Fecha del sistema.

Un segundo ejemplo, que muestre la **fecha** del sistema, sería:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Muestra la fecha      }
{    actual                }
{    FECHA.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program Fecha;

uses Dos;

const
  nombre : array [0..6] of String[15] =
    ('Domingo','Lunes','Martes','Miércoles',
     'Jueves','Viernes','Sábado');
var
  anyo, mes, dia, diaSem : Word;

begin
  GetDate( anyo, mes, dia, diaSem );
  writeln('Hoy es ', nombre[diaSem], ' ',
    dia, '/', mes, '/', anyo);
end.
```

15.3. Fecha usando interrupciones.

Esto mismo lo podemos hacer usando **interrupciones** del DOS (os aconsejo no experimentar con esto si no sabeis lo que estais haciendo; una buena fuente de información es la lista de interrupciones recopilada por Ralf Brown). Eso sí, esta forma no funcionará en compiladores de Pascal no basados en MsDos, como FreePascal:

```
 {--------------------------}
 {  Ejemplo en Pascal:      }
 {                          }
 {    Lee la fecha usando   }
 {    interrupciones        }
{    FECHA2.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

program Fecha2;

uses Dos;

var
  anyo, mes, dia: string;           { Aquí guardaremos cada dato }
  regs: Registers;                  { Necesaria para Intr }
             { El tipo Registers está definido en la unidad Dos }

begin
  regs.ah := $2a;            { Llamamos a la funcion 2Ah }
  with regs do
    intr($21,regs);          { De la interrupción 21h }
  with regs do
    begin
      str(cx, anyo);         { El año está en regs.CX, el mes en DH }
      str(dh, mes);          { y el día en DL.  Pasamos todo a cadena }
      str(dl, dia);          { con Str }
    end;
  writeln('Hoy es ', dia+'/'+mes+'/'+anyo);
end.
```

15.4. Lista de ficheros.

Podemos ver la **lista de los ficheros** que hay en un directorio mediante FindFirst y FindNext. Por ejemplo, para ver los que tienen extensión .PAS:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Lista de ficheros en  }
{    un directorio         }
{    LISTADIR.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
```

```
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{--------------------------}
program ListaDir;

uses Dos;

var
  Hallado: SearchRec;            { Información sobre el directorio }
                  { El tipo SearchRec está definido en la unidad Dos }

begin
  FindFirst( '*.PAS', AnyFile, Hallado );       { Los busca }
  while DosError = 0 do                         { Mientras existan }
    begin
    Writeln( Hallado.Name );       { Escribe el nombre hallado }
    FindNext( Hallado );           { y busca el siguiente }
    end;
end.
```

("AnyFile" es una constante, definida también en la unidad DOS y se refiere a que tome cualquier fichero, tenga los atributos que tenga).

15.5. Ejecutar otros programas.

Finalmente, vamos a ver un ejemplo de cómo **ejecutar** otros programas desde uno nuestro:

```
{--------------------------}
{  Ejemplo en Pascal:       }
{                           }
{    Ejecuta otro prog.     }
{    desde el nuestro       }
{    EJECUTA.PAS            }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Turbo Pascal 7.0     }
{    - Tmt Pascal Lt 1.20   }
{--------------------------}

{$M 2000,0,0 }   { 2000 bytes pila, sin heap }
{ Habrá que suprimir la línea anterior si se usa Tmt Pascal }

program EjecutaOrden;

uses dos;

begin
  SwapVectors;
```

```
  Exec('C:\WP51\WP.EXE', 'MITEXTO.TXT');
  SwapVectors;
end.
```

Lo del principio es una "**directiva de compilación**" (se tratan con más detalle en una de las ampliaciones del curso), que indica al compilador cuanta memoria queremos reservar. En este caso, 2000 bytes para la pila (donde se guardarán las llamadas recursivas, por ejemplo) y nada (0 como mínimo y 0 como máximo) para el Heap o "montón", donde se guardan las variables dinámicas, que no tiene este programa.

Después guardamos los **vectores de interrupción** con **SwapVectors**, y ejecutamos el programa, indicando su nombre (camino completo) y los parámetros. Finalmente, restauramos los vectores de interrupciones.

15.6: Ejecutar órdenes del DOS.

Podemos **ejecutar ordenes** del DOS como DIR y otros programas cuyo nombre completo (incluyendo directorios) no conozcamos, pero para eso hemos de llamar a COMMAND.COM con la opción /C:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejecuta una orden     }
{    interna de MsDos      }
{    EJECUTA2.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

{$M 2000,0,0 }   { 2000 bytes pila, sin heap }
{ Habrá que suprimir la línea anterior si se usa Tmt Pascal }

program EjecutaOrden2;

uses dos;

begin
  SwapVectors;
  Exec('C:\COMMAND.COM', '/C DIR *.PAS');
  SwapVectors;
end.
```

¿Y qué ocurre si nuestro intérprete de comandos no es el COMMAND.COM? Esto puede ocurrir, por ejemplo, a quienes usen 4DOS o NDOS, o versiones recientes de Windos, que usen CMD.EXE. Pues entonces podemos recurrir a la variable de entorno COMSPEC, que nos dice cual es el intérprete de comandos que estamos usando. Así, nuestra orden Exec quedaría:

```
Exec( GetEnv('COMSPEC'), '/C DIR *.PAS');
```

Como se ve en el ejemplo, **GetEnv** es la función que nos devuelve el valor de una variable de entorno. El fuente completo quedaría:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejecuta una orden     }
{    interna de MsDos/Win  }
{    EJECUTA3.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.20  }
{--------------------------}

{$M 2000,0,0 }   { 2000 bytes pila, sin heap }
{ Habrá que suprimir la línea anterior si se usa Tmt Pascal }

program EjecutaOrden3;

uses dos;

begin
  SwapVectors;
  Exec( GetEnv('COMSPEC'), '/C DIR *.PAS');
  SwapVectors;
end.
```

Pues se acabó por hoy. Como siempre, os recuerdo que queda mucho jugo por sacar a esta unidad, así que experimentad. Recordad que la ayuda on-line os puede sacar de más de un apuro y os puede descubrir más de una cosa.

Os propongo un par de **ejercicios**:

- A ver quien es capaz de hacer un programa que busque un determinado archivo (o archivos) en todo el disco duro, como hacen muchas herramientas tipo PcTools, o el propio DIR /S. Como "pista", habrá que usar FindFirst y FindNext.
- Un programita que muestre la hora en la parte superior de la pantalla mientras que nosotros escribimos en la parte inferior (lógicamente, la

hora se tiene que actualizar continuamente, no vale que sea lea al principio y nada más). En este caso, la "pista" es que habrá que emplear ReadKey y GetTime, que (con toda mi mala idea) no he contado, así que a tirar de la ayuda... }:-)

(Puedes ver una propuesta de solución -todavía no disponible-, así como un ejemplo -todavía no disponible- que muestra el espacio libre en todas las unidades de disco).

15.6: Principales posibilidades de la unidad DOS.

En el apartado anterior hemos visto algunas de las posibilidades de la unidad DOS. Ahora vamos a ver una lista de lo que nos permite (en el caso de Turbo Pascal 7.0), para que tengais más base para investigar y experimentar, sin tener que pelearos demasiado con la ayuda... :-)

Indicaré el nombre de cada procedimiento (proc) o función (func), su cometido y el formato de uso... a veces... :-)

Funciones de fecha y hora:

```
Nombre                      Cometido
------------------------    ----------------------------------------------
--
GetDate (proc)              Devuelve la fecha del sistema operativo.
                            Formato: GetDate(anyo, mes, dia, diaSemana)
                            donde anyo, mes, dia, diaSemana son de tipo
                            "word".
SetDate (proc)              Fija la fecha del sistema operativo.
                            Formato: SetDate(anyo, mes, dia)
GetTime (proc)              Devuelve la hora del sistema operativo.
                            Formato: GetTime(hora, min, seg, seg100)
                            con hora, mis, seg, seg100 de tipo "word".
SetTime (proc)              Fija la hora del sistema operativo.
                            Formato: SetTime(hora, min, seg, seg100)
GetFTime (proc)             Fecha y hora de la última modificación de
un
                            fichero.  El fichero debe estar asignado y
                            abierto.  La fecha y hora están en formato
                            empaquetado, que se puede descomprimir con
                            UnpackTime.
UnpackTime (proc)           Convierte la fecha y hora del formato que
```

```
                          devuelven GetFTime, FindFirst y FindNext en
                          un "record" de formato "DateTime":
                          type
                           TDateTime = record
                              Year,Month,Day,Hour,Min,Sec: Word;
                            end;
SetFTime (proc)           Fija la fecha y hora de un fichero.  La
fecha
                          y la hora se pasan como un valor comprimido
                          (longint), que se crea con PackTime.
PackTime (proc)           Convierte un registro DateTime a un valor
                          empaquetado para poder usar SetFTime.
```

Funciones de estado del disco.

```
Nombre                 Cometido
--------------------   -----------------------------------------------
--

DiskFree (func)        Devuelve el número de bytes libres en una
unidad
                       de disco.
DiskSize (func)        Devuelve el tamaño total (bytes) de una unidad
de
                       disco.
GetVerify (proc)       Devuelve el estado del flag de verificación
                       (verify) del DOS: si es TRUE, las escrituras
                       en disco son verificadas posteriormente; si es
                       FALSE, no se comprueban.
SetVerify (proc)       Fija el flag de verificación a TRUE o FALSE.
```

Funciones de manejo de ficheros.

```
Nombre                    Cometido
-----------------------   --------------------------------------------
-

FindFirst (proc)          Busca en el directorio que se indique el
primer
                          fichero que tenga unos ciertos atributos y
un
                          cierto nombre (se permiten comodines * y
?).
                          (Ver un ejemplo en el apartado anterior).
FindNext (proc)           Busca el siguiente fichero que cumpla las
                          condiciones fijadas con FindFirst.  Se
                          comprueba si aún quedan ficheros mirando el
```

```
                          valor de la variable DosError.
GetFAttr (proc)           Devuelve los atributos de un fichero.
Estos
                          se encuentran en forma de un "word", en el
                          que cada bit tiene un significado:
                          ReadOnly        $01
                          Hidden          $02
                          SysFile         $04
                          VolumeID        $08
                          Directory       $10
                          Archive         $20
                          AnyFile         $3F
                          Por tratarse de bits, lo habitual será
                          comprobar su valor utilizando el producto
                          lógico "and":
                          if atributos and Hidden <> 0 then ...
SetFAttr (proc)           Fija los atributos de un fichero.
FSearch (func)            Busca un fichero en una lista de
directorios.
FExpand (func)            Expande un nombre de fichero a
                          unidad+directorio+nombre+extensión.
FSplit (func)             Parte un nombre de fichero completo en
                          (directorio, nombre y extensión).
```

Soporte de interrupciones.

```
Nombre                 Cometido
--------------------   ----------------------------------------------
MsDos (proc)           Ejecuta una llamada a una función del DOS.
                       Para indicar los parámetros de la llamada
                       (valores de los registros del 80x86), se
                       debe usar una variable de tipo "Registers"
                       (ver ejemplo en el apartado anterior).
Intr (proc)            Llama a una cierta interrupción software.
GetIntVec (proc)       Devuelve la dirección a la que apunta un
                       cierto vector de interrupción.  Junto con
                       SetIntVec nos permite modificar las interrup-
                       ciones.  El uso de estas dos ordenes es
                       avanzado, y muy peligroso si no se domina lo
                       que se está haciendo.
SetIntVec (proc)       Fija la dirección a la que apuntará un
                       cierto vector de interrupción.
```

Manejo de procesos.

```
Nombre                     Cometido
------------------------   ---------------------------------------------
--
Exec (proc)                Ejecuta un cierto programa externo,
pasándole
                           una serie de parámetros.
SwapVectors (proc)         Intercambia los vectores de interrupción
                           salvados con los actuales.  Será necesario
                           hacerlo antes y después de Exec,
especialmente
                           si hemos redirigido alguna interrupción.
Keep (proc)                Hace que el programa termine y se quede
                           residente (TSR).  El problema está en cómo
                           acceder a él después...  :-)
                           (Se puede conseguir usando GetIntVec y
                           SetIntVec, pero esto cae muyyyy por encima
                           del nivel del curso).
DosExitCode (func)         Devuelve el código de salida de un
subproceso,
                           el "errorlevel" que usan los ficheros BAT,
                           y que nosotros podemos fijar haciendo
halt(n).
```

Misceláneas.

```
Nombre                     Cometido
------------------------   ---------------------------------------------
--
DosVersion (func)          Devuelve el número de versión del DOS.  Es
                           un valor de tipo "word", cuyo byte bajo nos
                           indica el número principal de versión (6 en
                           MsDos 6.22, p.ej. ) y cuyo byte alto nos
dice
                           el número secundario de versión (22 en el
                           ejemplo anterior).
GetCBreak (proc)           Dice si el DOS comprobará la pulsación de
                           Ctrl-Break (Ctrl+Inter en los teclados
                           españoles).
SetCBreak (proc)           Hace que el DOS compruebe o no la pulsación
                           de Ctrl-Break.
```

Curso de Pascal. Tema 16: Programación Orientada a Objetos.

Antes de empezar, un consejo: si la Programación Orientada a Objetos es nueva para ti, lee todo este **texto seguido**: Si algo no lo entiendes en un principio, es muy fácil que algún párrafo posterior te lo aclare... espero... O:-)

Hasta ahora estamos estado "**cuadriculando**" todo para obtener algoritmos: tratábamos de convertir cualquier cosa en un procedimiento o una función que pudiéramos emplear en nuestros programas.

Pero no todo lo que nos rodea es tan fácil de cuadricular. Supongamos por ejemplo que tenemos que introducir datos sobre una puerta en nuestro programa. ¿Nos limitamos a programar los procedimientos AbrirPuerta y CerrarPuerta? Al menos, deberíamos ir a la zona de declaración de variables, y allí guardaríamos otras datos como su tamaño, color, etc.

No está mal, pero es **antinatural**. ¿A qué me refiero? Pues a que una puerta es un **conjunto**: no podemos separar su color de su tamaño, o de la forma en que debemos abrirla o cerrarla. Sus características son tanto las físicas (lo que hasta ahora llamábamos variables) como sus comportamientos en distintas circunstancias (lo que para nosotros eran los procedimientos). Todo ello va unido, formando un **OBJETO**.

Por otra parte, si tenemos que explicar a alguien lo que es el portón de un garaje, y ese alguien no lo ha visto nunca, pero conoce cómo es la puerta de su casa, le podemos decir "se parece a una puerta de una casa, pero es más grande para que quepan los coches, está hecha de metal en vez de madera..."

Finalmente, conviene recordar que "abrir" no se refiere sólo a una puerta. También podemos hablar de abrir una ventana o un libro, por ejemplo.

Pues con esta discusión filosófica ;-) hemos comentado casi sin saberlo las tres **características** más importantes de la Programación Orientada a Objetos (OOP):

- **ENCAPSULACION**: No podemos separar los comportamientos de las características de un objeto. Los comportamientos serán procedimientos o funciones, que en OOP llamaremos METODOS. Las características serán variables, como las que hemos usado siempre. La apariencia de un objeto en Pascal, como veremos un poco más adelante, recordará a un registro o "record".
- **HERENCIA**: Unos objetos pueden heredar métodos y datos de otros. Esto hace más fácil definir objetos nuevos a partir de otros que ya teníamos anteriormente (como ocurría con el portón y la puerta) y facilitará la reescritura de los programas, pudiendo aprovechar buena parte de los anteriores... si están bien diseñados.
- **POLIMORFISMO**: Un mismo nombre de un método puede hacer referencia a comportamientos distintos (como abrir una puerta o un

libro). Igual ocurre para los datos: el peso de una puerta y el de un portón los podemos llamar de igual forma, pero obviamente no valdrán lo mismo.

Comentado esto, vamos a empezar a ver ejemplos en Pascal para tratar de fijar conceptos. Para todo lo que viene a continuación, será necesario emplear la versión **5.5 o superior** de Turbo Pascal.

Los **objetos en Pascal** se definen de forma parecida a los registros (record), sólo que ahora también incluirán procedimientos y funciones. Vamos a escribir un mensaje en una cierta posición de la pantalla. Este será nuestro objeto "título".

```
type                                              { Def. tipo }
  titulo = object
    texto: string;                                { Variables }
    x,y : byte;

    procedure FijaCoords(nuevoX, nuevoY: byte);   { Métodos }
    procedure FijaTexto(mensaje: string);
    procedure Escribe;
  end;

var
  miTitulo: titulo;                        { Variable de tipo objeto }
```

Accederemos a los métodos y a los datos precediendo el nombre de cada uno por el nombre de la variable y por un punto, como hacíamos con los registros (record):

```
miTitulo.x := 23;
miTitulo.y := 12;
miTitulo.FijaTexto('Hola');
miTitulo.Escribe;
```

Lo anterior se entiende, ¿verdad? Sin embargo, hay algo que se puede pero no se debe hacer: NO SE DEBE ACCEDER DIRECTAMENTE A LOS DATOS. Esto es otra de las máximas de la OOP (la **ocultación de datos**). Para modificarlos, lo haremos siempre a través de algún procedimiento (método) del objeto.

Por eso es por lo que hemos definido los procedimientos FijaCoords y FijaTexto... ¿¡definido!? ¡Pero si sólo está la primera línea!... }:-)

Don't panic! Esto no es nuevo para nosotros: cuando creábamos una unidad (unit), teníamos un encabezamiento (interface) en el que sólo poníamos las cabeceras de las funciones y procedimientos, y luego un desarrollo

(implementation), donde realmente iba todo el código. ¿Verdad? Pues aquí ocurrirá algo parecido.

Vamos a ir **completando** el programa para que funcione, y así irá quedando todo más claro:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primer ejemplo de     }
{    Prog. Orientada Obj.  }
{    OBJETOS1.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program Objetos1;                    { Nuestro primer programa en OOP }
uses crt;                                         { Usaremos "GotoXY" }
type                                 { Aquí definimos nuestro objeto }
  titulo = object
    texto: string;                           { El texto que se escribirá }
    x,y : byte;                                    { En qué posición }
    procedure FijaCoords(nuevoX, nuevoY: byte);            { Pues eso }
    procedure FijaTexto(mensaje: string);                      { Idem }
    procedure Escribe;                             { Lo escribe, claro }
  end;
var
  miTitulo: titulo;                       { Una variable de ese tipo }
procedure titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                                   { Definimos el procedimiento: }
  x := nuevoX;                          { Actualiza las coordenadas }
  y := nuevoY;
end;
procedure titulo.FijaTexto(mensaje: string);
begin                                           { Actualiza el texto }
  Texto := Mensaje;
end;
procedure titulo.Escribe;
begin                                             { Muestra el título }
  Gotoxy(X,Y);
  Write(Texto);
end;
begin                                    { -- Cuerpo del programa --}
  ClrScr;  { Que se vea qué hacemos }
  miTitulo.FijaCoords(37,12);
  miTitulo.FijaTexto('Hola');
  miTitulo.Escribe;
end.
```

¿Y todo este rollazo de programa para escribir "Hola"? Nooooo.... Ya iremos viendo las ventajas, pero eso el próximo día, para que no resulte demasiado denso...

16.2: Herencia y polimorfismo.

En el apartado anterior del tema vimos una introducción a lo que eran los objetos y a cómo podíamos utilizarlos desde Turbo Pascal. Hoy vamos a ver qué es eso de la **herencia** y el **polimorfismo**.

Habíamos definido un objeto "título": un cierto texto que se escribía en unas coordenadas de la pantalla que nosotros fijásemos.

```
type
 titulo = object
   texto: string;                      { El texto que se escribirá }
   x,y : byte;                                   { En qué posición }

   procedure FijaCoords(nuevoX, nuevoY: byte);          { Pues eso }
   procedure FijaTexto(mensaje: string);                    { Idem }
   procedure Escribe;                          { Lo escribe, claro }
 end;
```

Funciona, pero hemos tecleado mucho para hacer muy poco. Si de verdad queremos que se vaya pareciendo a un título, lo menos que deberíamos hacer es poder cambiar el color para que resalte un poco más.

Para conseguirlo, podemos modificar nuestro objeto o crear otro. Supongamos que nos interesa conservar ese tal y como está porque lo hemos usado en muchos programas, etc, etc.

Pues con el Pascal "de toda la vida" la opción que nos queda sería crear otro objeto. Con los editores de texto que tenemos a nuestro alcance, como los que incluye el Entorno de Desarrollo de TP6 y TP7 esto no es mucho problema porque no hay que teclear demasiado: marcamos un bloque, lo copiamos y modificamos lo que nos interese.

Pero vamos teniendo nuevas versiones de nuestros objetos por ahí desperdigadas. Si un día descubrimos una orden más rápida o más adecuada que Write para usarla en el procedimiento "escribe", tendremos que buscar cada versión del objeto en cada programa, modificarla, etc...

La **herencia** nos evita todo esto. Podemos definir un nuevo objeto partiendo del que ya teníamos. En nuestro caso, conservaremos la base del objeto "Titulo" pero añadiremos el manejo del color y retocaremos "escribe" para que lo contemple.

El nuevo objeto quedaría:

```
type
 TituloColor = object( titulo )
```

```
    color: byte;                                    { El color, claro }
    procedure FijaColores(pluma, fondo: byte);             { Pues eso }
    procedure Escribe;                   { Lo escribe de distinta forma }
  end;
```

Aunque no lo parezca a primera vista, nuestro objeto sigue teniendo los métodos "FijaCoords" y "FijaTexto". ¿Donde están? Pues en la primera línea:

```
object ( titulo )
```

quiere decir que es un objeto **descendiente** de "titulo". Tendrá todos sus métodos y variables más los nuevos que nosotros indiquemos (en este caso, "color" y "FijaColores"). Además podemos redefinir el comportamiento de algún método, como hemos hecho con Escribe.

Veamos cómo quedaría nuestro programa ampliado

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Segundo ejemplo de    }
{    Prog. Orientada Obj.  }
{    OBJETOS2.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program Objetos2;                  { Nuestro segundo programa en OOP }
uses crt;                             { Usaremos "GotoXY" y TextAttr }
type                                 { Aquí definimos nuestro objeto }
  titulo = object
    texto: string;                         { El texto que se escribirá }
    x,y : byte;                                     { En qué posición }
    procedure FijaCoords(nuevoX, nuevoY: byte);            { Pues eso }
    procedure FijaTexto(mensaje: string);                      { Idem }
    procedure Escribe;                             { Lo escribe, claro }
  end;
 type
  TituloColor = object( titulo )
    color: byte;                                    { El color, claro }
    procedure FijaColores(pluma, fondo: byte);             { Pues eso }
    procedure Escribe;                   { Lo escribe de distinta forma }
  end;
var
  T1: titulo;                          { Una variable de ese tipo }
  T2: tituloColor;                     { Y otra del otro ;-)   }
{ --- Desarrollo del objeto Titulo --- }
procedure titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                                   { Definimos el procedimiento: }
  x := nuevoX;                             { Actualiza las coordenadas }
  y := nuevoY;
end;
```

```
procedure titulo.FijaTexto(mensaje: string);
begin                                            { Actualiza el texto }
  Texto := Mensaje;
end;
procedure titulo.Escribe;
begin                                             { Muestra el título }
  Gotoxy(X,Y);
  Write(Texto);
end;
{ --- Métodos específicos de TituloColor --- }
procedure tituloColor.FijaColores(pluma,fondo: byte);
begin                                   { Definimos el procedimiento: }
  color := pluma + fondo*16;                     { Actualiza el color }
end;
procedure tituloColor.Escribe;
begin                                             { Muestra el título }
  textAttr := color;
  Gotoxy(X,Y);
  Write(Texto);
end;
{ -- Cuerpo del programa --}
begin
  ClrScr;
  T1.FijaCoords(37,12);
  T1.FijaTexto('Hola');
  T1.Escribe;
  T2.FijaCoords(37,13);
  T2.FijaColores(14,2);
  T2.FijaTexto('Adiós');
  T2.Escribe;
end.
```

En el mismo programa, como quien no quiere la cosa ;-) , tenemos un ejemplo de **polimorfismo**: no es sólo que las variables "texto", "x" e "y" esten definidas en los dos objetos de igual forma y tengan valores diferentes, sino que incluso el método "Escribe" se llama igual pero no actúa de la misma forma.

Antes de dar este apartado por "sabido" para pasar a ver qué son los constructores, los destructores, los métodos virtuales, etc... un par de **comentarios**:

- ¿Verdad que la definición de "Escribe" se parece mucho en "Titulo" y en "TituloColor"? Hay una parte que es exactamente igual. ¡Pues vaya asco de herencia si tenemos que volver a copiar partes que son iguales! X-D No, hay un truquillo en Turbo Pascal 7: tenemos la palabra clave "**inherited**" (heredado), con la que podríamos hacer:

```
procedure tituloColor.Escribe;
begin
```

```
  textAttr := color;
  inherited escribe;
end;
```

Es decir: cambiamos el color y luego todo es igual que el Escribe que hemos heredado del objeto padre ("titulo"). En otras versiones anteriores de Turbo Pascal (5.5 y 6) no existe la palabra inherited, y deberíamos haber hecho

```
procedure tituloColor.Escribe;
begin
  textAttr := color;
  titulo.escribe;
end;
```

que es equivalente. El inconveniente es que tenemos que recordar el nombre del padre.

Los problemas que puede haber con herencia de este tipo los veremos cuando digamos qué son métodos virtuales...

- Segundo comentario: ¿Qué ocurre si ejecutamos "T1.Escribe" sin haber dado **valores** a las coordenadas ni al texto? Cualquier cosa... }:-)

Me explico: no hemos inicializado las variables, de modo que "x" valdrá lo que hubiera en la posición de memoria que el compilador le ha asignado a esta variable. Si este valor fuera mayor de 80, estaríamos intentando escribir fuera de la pantalla. Igual con "y", y a saber lo que nos aparecería en "texto"...

¿Cómo lo solucionamos? Pues por ejemplo creando un procedimiento "inicializar" o similar, que sea lo primero que ejecutemos al usar nuestro objeto. Por ejemplo:

```
procedure titulo.init;
begin
  x := 1;
  y := 1;
  texto := '';
end;
[...]

procedure tituloColor.init;
begin
  inherited init;
```

```
  color := 0;
end;

[...]

begin
  titulo.Init;
  tituloColor.Init;
  [...]
end.
```

Antes de dar por terminada esta lección, un comentario sobre OOP en general, no centrado en Pascal: puede que alguien oiga por ahí el término "**sobrecarga**". Es un tipo de polimorfismo: sobrecarga es cuando tenemos varios métodos que se llaman igual pero cuyo cuerpo es distinto, y polimorfismo puro sería cuando tenemos un solo método que se aplica a argumentos de distinto tipo.

En C++ se habla incluso de la sobrecarga de operadores: podemos redefinir operadores como "+" (y muchos más) para sumar (en este caso) objetos que hayamos creado, de forma más cómoda y legible:

```
 matriz3 = matriz1 + matriz2
```

en vez de hacerlo mediante una función:

```
matriz3 = suma( matriz1, matriz2 )
```

Continuará... :-)

16.3: Problemas con la herencia.

Hemos visto una introducción a los objetos, y hemos empezado a manejar la herencia y el polimorfismo. Ahora vamos a ver algo que puede parecer desconcertante y que nos ayudará a entender qué son los **métodos virtuales** (espero) ;-)

Por cierto, este tema es "denso". No intentes "dominarlo" a la primera lectura. Si te pierdes, prueba a releerlo entero. Si te siguen quedando dudas, pregunta... :-)

Habíamos creado nuestro objeto "Titulo". Despues introdujimos otro llamado "TituloColor" que aprovechaba parte de las definiciones del primero, le añadimos cosas nuevas y retocamos las que nos interesaron.

Ahora vamos a crear un "TituloParpadeo", que será exactamente igual que "TituloColor", con la diferencia de que además el texto parpadeará.

Si alguien ha trabajado un poco con los colores en modo texto, sabrá que el único cambio que tenemos que hacer es sumar 128 al color que vamos a usar (o si lo preferís ver así: fijamos el primer bit de los atributos, que es el que indica el parpadeo).

Entonces crearemos un nuevo objeto que heredará todo de "TituloColor" y sólo redefinirá "FijaColores". El programa queda así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Tercer ejemplo de     }
{    Prog. Orientada Obj.  }
{    OBJETOS3.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program Objetos3;                 { Nuestro tercer programa en OOP }
uses crt;                           { Usaremos "GotoXY" y TextAttr }
type                               { Aquí definimos nuestro objeto }
  titulo = object
    texto: string;                     { El texto que se escribirá }
    x,y : byte;                                  { En qué posición }
    procedure FijaCoords(nuevoX, nuevoY: byte);         { Pues eso }
    procedure FijaTexto(mensaje: string);                   { Idem }
    procedure Escribe;                         { Lo escribe, claro }
  end;
 type
  TituloColor = object( titulo )
    color: byte;                                 { El color, claro }
    procedure FijaColores(pluma, fondo: byte);          { Pues eso }
    procedure Escribe;               { Lo escribe de distinta forma }
  end;
 type
  TituloParpadeo = object( tituloColor )
    procedure FijaColores(pluma, fondo: byte);         { Pues eso }
  end;
var
  T1: titulo;                     { Una variable de ese tipo }
  T2: tituloColor;                { Y otra del otro ;-)   }
  T3: tituloParpadeo;             { Y el que queda }
{ --- Desarrollo del objeto Titulo --- }
procedure titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                               { Definimos el procedimiento: }
  x := nuevoX;                        { Actualiza las coordenadas }
  y := nuevoY;
end;
procedure titulo.FijaTexto(mensaje: string);
begin                                          { Actualiza el texto }
```

```
  Texto := Mensaje;
end;
procedure titulo.Escribe;
begin                                      { Muestra el título }
  Gotoxy(X,Y);
  Write(Texto);
end;
{ --- Métodos específicos de TituloColor --- }
procedure tituloColor.FijaColores(pluma,fondo: byte);
begin                            { Definimos el procedimiento: }
  color := pluma + fondo*16;               { Actualiza el color }
end;
procedure tituloColor.Escribe;
begin                                      { Muestra el título }
  textAttr := color;
  Gotoxy(X,Y);
  Write(Texto);
end;
{ --- Métodos específicos de TituloParpadeo --- }
procedure tituloParpadeo.FijaColores(pluma,fondo: byte);
begin                            { Definimos el procedimiento: }
  color := pluma + fondo*16 + 128;         { Actualiza el color }
end;
{ -- Cuerpo del programa --}
begin
  ClrScr;
  T1.FijaCoords(37,12);
  T1.FijaTexto('Hola');
  T1.Escribe;
  T2.FijaCoords(37,13);
  T2.FijaColores(14,2);
  T2.FijaTexto('Adiós');
  T2.Escribe;
  T3.FijaCoords(35,14);
  T3.FijaColores(15,3);
  T3.FijaTexto('Y una más');
  T3.Escribe;
end.
```

No hemos definido Escribe para TituloParpadeo y sin embargo lo hemos usado. Claro, es que tampoco hacía falta redefinirlo. Recordemos la definición de TituloColor.Escribe:

```
procedure tituloColor.Escribe;
begin
  textAttr := color;    { Asignamos el color }
  Gotoxy(X,Y);          { Vamos a la posición adecuada }
 Write(Texto);          { Y escribimos el texto }
end;
```

Todo nos sirve: queremos hacer eso mismo, pero con "TituloParpadeo" en vez de con "TituloColor". No hay problema, el compilador se da cuenta de que es T3 quien intenta escribir, y toma las variables X, Y, Color y Texto apropiadas.

Así tenemos el primer rótulo "Hola", con letras grises y fondo negro. El segundo, "Adiós", tiene letras amarillas y fondo verde. El tercero, "Y una más", tiene fondo azul claro y letras blancas parpadeantes.

Inciso momentáneo: En OOP se suele llamar **CLASE** a una definición genérica de un objeto, como en nuestro caso "TituloColor" e **INSTANCIA** a una variable concreta que pertenece a ese clase, como T2 en nuestro ejemplo. Como ejemplo más general, "persona" sería una clase, formada por muchos individuos con unas características comunes, mientras que "Pepe" sería un **objeto** (sin ánimo de ofender) de la clase persona, una instancia de esa clase.

Así me puedo permitir el lujo de usar las palabras "clase" e "instancia" cuando quiera y ya nadie tiene excusa para protestar... O:-)

Curso de Pascal. Tema 16.4: Más detalles...

Volviendo a lo que nos interesa. Vamos a ver lo que ocurre si en vez de que el compilador determine a quién pertenecen X e Y (datos) le hacemos que tenga que determinar a quién pertenece un método como Escribe.

Para ello vamos a crear un nuevo método que va a asignar y escribir el texto en un sólo paso. Nuestra nueva versión del programa queda:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Cuarto ejemplo de     }
{    Prog. Orientada Obj.  }
{    OBJETOS4.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program Objetos4;                  { Nuestro cuarto programa en OOP }
uses crt;                            { Usaremos "GotoXY" y TextAttr }
type                                { Aquí definimos nuestro objeto }
  titulo = object
    texto: string;                     { El texto que se escribirá }
    x,y : byte;                                 { En qué posición }
    procedure FijaCoords(nuevoX, nuevoY: byte);          { Pues eso }
    procedure FijaTexto(mensaje: string);                   { Idem }
    procedure Escribe;                          { Lo escribe, claro }
    procedure AsignaYEscribe(nuevoTexto:string);
  end;
 type
  TituloColor = object( titulo )
```

```
    color: byte;                                  { El color, claro }
    procedure FijaColores(pluma, fondo: byte);          { Pues eso }
    procedure Escribe;                { Lo escribe de distinta forma }
  end;
 type
  TituloParpadeo = object( tituloColor )
    procedure FijaColores(pluma, fondo: byte);         { Pues eso }
  end;
var
  T1: titulo;                       { Una variable de ese tipo }
  T2: tituloColor;                  { Y otra del otro ;-)   }
  T3: tituloParpadeo;               { Y el que queda }
{ --- Desarrollo del objeto Titulo --- }
procedure titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                                 { Definimos el procedimiento: }
  x := nuevoX;                          { Actualiza las coordenadas }
  y := nuevoY;
end;
procedure titulo.FijaTexto(mensaje: string);
begin                                            { Actualiza el texto }
  Texto := Mensaje;
end;
procedure titulo.Escribe;
begin                                            { Muestra el título }
  Gotoxy(X,Y);
  Write(Texto);
end;
procedure titulo.AsignaYEscribe(nuevoTexto:string);
begin
  FijaTexto(NuevoTexto);
  Escribe;
end;
{ --- Métodos específicos de TituloColor --- }
procedure tituloColor.FijaColores(pluma,fondo: byte);
begin                                 { Definimos el procedimiento: }
  color := pluma + fondo*16;                     { Actualiza el color }
end;
procedure tituloColor.Escribe;
begin                                            { Muestra el título }
  textAttr := color;
  Gotoxy(X,Y);
  Write(Texto);
end;
{ --- Métodos específicos de TituloParpadeo --- }
procedure tituloParpadeo.FijaColores(pluma,fondo: byte);
begin                                 { Definimos el procedimiento: }
  color := pluma + fondo*16 + 128;               { Actualiza el color }
end;
{ -- Cuerpo del programa --}
begin
  ClrScr;
  T1.FijaCoords(37,12);
  T1.AsignaYEscribe('Hola');
  T2.FijaCoords(37,13);
  T2.FijaColores(14,2);
  T2.AsignaYEscribe('Adiós');
  T2.Escribe;
  T3.FijaCoords(35,14);
  T3.FijaColores(15,3);
  T3.AsignaYEscribe('Y una más');
end.
```

Aparentemente, todo bien, ¿verdad? Hemos hecho lo mismo que antes: no hemos redefinido "AsignaYEscribe" porque no nos hace falta:

```
procedure titulo.AsignaYEscribe(nuevoTexto:string);
begin
  FijaTexto(NuevoTexto);
  Escribe;
end;
```

Queremos dar ese valor al texto y escribir el título. Pues sí, todo bien... ¿o no?

Si lo ejecutais vereis que no: el título que corresponde a T3 no ha salido con los mismos colores de antes (blanco parpadeante sobre azul claro) sino con los que corresponden a T2 (amarillo sobre verde).

¿Por qué?

Veamos: ¿dónde está el Escribe de "TituloParpadeo"? ¡No está! Claro, si lo hemos heredado... pero por ahí vienen los **fallos**...

Cuando el compilador ve que el método AsignaYEscribe llama a Escribe, enlaza cada AsignaEscribe con su Escribe correspondiente. Pero para "TituloParpadeo" no existe Escribe, así que ¿qué hace? Pues toma el más cercano, el de su padre, TítuloColor.

Como comprobación, podeis hacer que el programa termine así:

```
  T3.AsignaYEscribe('Y una más');    { Con métodos anidados }
  readln;                            { Esperamos que se pulse INTRO }
  T3.Escribe;                        { Y lo hacemos directamente }
end.
```

En cuanto pulsamos INTRO, ejecuta T3.Escribe, y éste ya sí que funciona bien.

Curso de Pascal. Tema 16.5: Métodos virtuales.

Entonces está claro que el **error** aparece cuando un método llama a otro que no hemos redefinido, porque puede **no saber** con quien enlazarlo. Lo intentará con su padre o ascendiente más cercano, y los resultados quizá no sean los adecuados.

Deberíamos regañar a nuestro compilador y decirle...

"Mira, no me presupongas cosas. Cuando alguien te diga Escribe, haz el favor de mirar quién es, y hacerlo como sabes que él quiere." ;-)

¿Ah, pero se puede hacer eso? Pues resulta que sí: lo que el compilador hace si no le decimos nada es un **enlace estático** o "temprano" (en inglés, **early binding**), en el momento de compilar el programa. Pero también podemos pedirle que haga un **enlace dinámico** o "tardío" (**late binding**) justo en el momento de la ejecución, que es lo que nos interesa en estos casos.

De modo que con el enlace dinámico damos una mayor flexibilidad y potencia a la Herencia. Por contra, como los enlaces se calculan en tiempo de ejecución, el programa resultante será ligeramente más lento.

¿Y cómo se hace eso del enlace dinámico? Pues indicando que se trata de un método **VIRTUAL**. ¿Y eso otro cómo se hace? Fácil:, en la cabecera del método añadimos esa palabra después del punto y coma:

```
type                                    { Aquí definimos nuestro objeto }
  titulo = object
    [...]
    procedure Escribe; virtual;
    [...]
  end;
```

Antes de ver cómo quedaría nuestro programa, unos **comentarios**:

- Si un método es **virtual** en un cierto objeto, deberá serlo también en todos sus descendientes (objetos definidos a partir de él mediante herencia).
- Antes de llamar al método virtual, el compilador debe hacer alguna inicialización que le asegure que va a encontrar a dónde apunta ese método virtual, etc. Para ello deberemos definir un método constructor . El simple hecho de definir un método (el primero que vayamos a utilizar) como "**constructor**" en vez de "procedure" hace que el compilador nos asegure que todo va a ir bien. Ese contructor podemos llamarlo, por ejemplo, "**init**" (inicializar, en inglés; es el nombre que usa Turbo Pascal en todos los ejemplos que incluye).

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Quinto ejemplo de     }
{    Prog. Orientada Obj.  }
{    OBJETOS5.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
```

```
{     - Free Pascal 2.2.0w   }
{     - Turbo Pascal 7.0     }
{--------------------------}
program Objetos5;                     { Nuestro quinto programa en OOP }
uses crt;                               { Usaremos "GotoXY" y TextAttr }
type                                   { Aquí definimos nuestro objeto }
  titulo = object
    texto: string;                         { El texto que se escribirá }
    x,y : byte;                                   { En qué posición }
    constructor FijaCoords(nuevoX, nuevoY: byte);          { <======= }
    procedure FijaTexto(mensaje: string);                      { Idem }
    procedure Escribe; virtual;                   { <===== Lo escribe }
    procedure AsignaYEscribe(nuevoTexto:string);
  end;
 type
  TituloColor = object( titulo )
    color: byte;                                    { El color, claro }
    procedure FijaColores(pluma, fondo: byte);             { Pues eso }
    procedure Escribe; virtual;            { <====== Virtual también }
  end;
 type
  TituloParpadeo = object( tituloColor )
    procedure FijaColores(pluma, fondo: byte);            { Pues eso }
  end;
var
  T1: titulo;                         { Una variable de ese tipo }
  T2: tituloColor;                    { Y otra del otro ;-)   }
  T3: tituloParpadeo;                 { Y el que queda }
{ --- Desarrollo del objeto Titulo --- }
constructor titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                                  { Definimos el procedimiento: }
  x := nuevoX;                           { Actualiza las coordenadas }
  y := nuevoY;
end;
procedure titulo.FijaTexto(mensaje: string);
begin                                             { Actualiza el texto }
  Texto := Mensaje;
end;
procedure titulo.Escribe;
begin                                              { Muestra el título }
  Gotoxy(X,Y);
  Write(Texto);
end;
procedure titulo.AsignaYEscribe(nuevoTexto:string);
begin
  FijaTexto(NuevoTexto);
  Escribe;
end;
{ --- Métodos específicos de TituloColor --- }
procedure tituloColor.FijaColores(pluma,fondo: byte);
begin                                  { Definimos el procedimiento: }
  color := pluma + fondo*16;                      { Actualiza el color }
end;
procedure tituloColor.Escribe;
begin                                              { Muestra el título }
  textAttr := color;
  Gotoxy(X,Y);
  Write(Texto);
end;
{ --- Métodos específicos de TituloParpadeo --- }
procedure tituloParpadeo.FijaColores(pluma,fondo: byte);
```

```
begin                                   { Definimos el procedimiento: }
  color := pluma + fondo*16 + 128;               { Actualiza el color }
end;
{ -- Cuerpo del programa --}
begin
  ClrScr;
  T1.FijaCoords(37,12);
  T1.AsignaYEscribe('Hola');
  T2.FijaCoords(37,13);
  T2.FijaColores(14,2);
  T2.AsignaYEscribe('Adiós');
  T2.Escribe;
  T3.FijaCoords(35,14);
  T3.FijaColores(15,3);
  T3.AsignaYEscribe('Y una más');
end.
```

Continuará... :-)

Curso de Pascal. Tema 16.6: Programación Orientada a Objetos (6).

Se supone que ya entendemos la base de la OOP en Pascal. Hemos visto como definir y heredar datos o métodos de una "clase" (objeto genérico), cómo redefinirlos, y como crear y manejar "instancias" de esa clase (variables concretas).

También hemos tratado los métodos "virtuales", para que el compilador sepa siempre de qué clase "padre" estamos heredando.

En estos casos, necesitábamos emplear en primer lugar un método "**constructor**", que prepararía una tabla dinámica en la que se van a almacenar todas estas referencias que permiten que nuestro programa sepa a qué método o dato debe acceder en cada momento.

También habíamos comentado que los constructores tenían otro uso, y que existía algo llamado "**destructores**". Eso es lo que vamos a ver hoy.

Vamos a empezar por hacer una variante de nuestro objeto. En ella, el texto va a guardarse de forma **dinámica**, mediante un puntero.

¿Para qué? Esto es repaso de la lección sobre punteros: la primera ventaja es que no vamos a reservarle memoria del segmento de datos (64K) sino del "heap" (los 640K de memoria convencional). La segunda es que podríamos enlazar unos con otros para crear una lista de textos tan grande como quisiéramos (y la memoria nos permitiese, claro).

No vamos a ver lo de la lista, que ya tratamos en su día, y que no nos interesa hoy especialmente, y vamos a centrarnos en las diferencias que introduciría este puntero en nuestro objeto.

Antes de empezar a usar nuestro objeto, tendremos que reservar memoria para el texto, como hacíamos con cualquier otro puntero:

```
new( texto );
```

y cuando terminemos de usar el objeto, debemos liberar la memoria que habíamos reservado:

```
dispose( texto );
```

Hasta ahora todo claro, ¿no? Pues vamos a ir fijando conceptos. La orden "new" la debemos usar antes que nada, al inicializar el objeto. Así que crearemos un método encargado de **inicializar** el objeto (reservar memoria, darle los valores iniciales que nos interesen, etc). Lo llamaré "**init**", que es el nombre que suele usar Borland, pero se puede elegir cualquier otro nombre.

Entonces, nuestro "init" podría quedar simplemente así:

```
procedure titulo.init(TextoInicial: string);
begin
  new(texto);                         { Reservamos memoria }
  texto^ := TextoInicial;             { Damos valor al texto }
end;
```

o mejor podemos aprovechar para dar valores iniciales a las demás variables:

```
procedure titulo.init(TextoInicial: string);
begin
  new(texto);                         { Reservamos memoria }
  texto^ := TextoInicial;             { Damos valor al texto }
  x :=1 ; y := 1;                     { Coordenadas por defecto }
  color := 7;                         { Gris sobre negro }
end;
```

A su vez, tendremos que **liberar** esta memoria al terminar, para lo que crearemos otro método, que llamaré "**done**" (hecho) también por seguir el esquema que suele usar Borland. En nuestro caso, que es sencillo, bastaría con

```
procedure titulo.done;
begin
```

```
  dispose(texto);                         { Liberamos la memoria }
end;
```

Vamos a ver cómo iría quedando nuestra clase con estos cambios:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Objetos y punteros-1  }
{    OBJETOP1.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program ObjetosConPunteros;
uses crt;                               { Usaremos "GotoXY" y TextAttr }
type                                   { Aquí definimos nuestro objeto }
  titulo = object
    texto: ^string;                       { El texto que se escribirá }
    x,y : byte;                                     { En qué posición }
    color: byte;                                     { El color, claro }
    procedure init(TextoInicial: string);                 { Inicializa }
    procedure FijaCoords(nuevoX, nuevoY: byte);             { Pues eso }
    procedure FijaTexto(mensaje: string);                       { Idem }
    procedure FijaColores(pluma,fondo: byte);              { y lo otro }
    procedure Escribe;                             { Lo escribe, claro }
    procedure done;                                  { Libera memoria }
  end;
{ --- Desarrollo del objeto Titulo --- }
procedure titulo.init(TextoInicial: string);
begin
  new(texto);                          { Reservamos memoria }
  texto^ := TextoInicial;              { Damos valor al texto }
  x :=1 ; y := 1;                      { Coordenadas por defecto }
  color := 7;                          { Gris sobre negro }
end;
procedure titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                                  { Definimos el procedimiento: }
  x := nuevoX;                           { Actualiza las coordenadas }
  y := nuevoY;
end;
procedure titulo.FijaTexto(mensaje: string);
begin                                           { Actualiza el texto }
  Texto^ := Mensaje;
end;
procedure titulo.FijaColores(pluma,fondo: byte);
begin                                  { Definimos el procedimiento: }
  color := pluma + fondo*16;                    { Actualiza el color }
end;
procedure titulo.Escribe;
begin
  TextAttr := color;                              { Muestra el título
}
  Gotoxy(X,Y);
  Write(Texto^);
```

```
end;
procedure titulo.done;
begin
  dispose(texto);                        { Liberamos la memoria }
end;
var
  t1: titulo;
{ -- Cuerpo del programa --}
begin
  ClrScr;
  T1.Init('Por defecto');
  T1.Escribe;
  T1.FijaCoords(37,12);
  T1.FijaColores(14,2);
  T1.FijaTexto('Modificado');
  T1.Escribe;
  T1.Done;
end.
```

¿Todo claro? Espero que sí. Hasta ahora no ha habido grandes cambios...

Curso de Pascal. Tema 16.7: Programación Orientada a Objetos (7).

Pero ¿qué ocurre si nuestra clase (objeto) tiene **métodos virtuales**? Entonces recordemos que era necesario un método **constructor**, que se ejecute antes que ningún otro. En nuestro caso es evidente que nos conviene que el constructor sea el "init".

El programa queda ahora

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Objetos y punteros-2  }
{    OBJETOP2.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program ObjetosConPunteros2;
uses crt;                               { Usaremos "GotoXY" y TextAttr }
type                                   { Aquí definimos nuestro objeto }
  titulo = object
    texto: ^string;                        { El texto que se escribirá }
    x,y : byte;                                      { En qué posición }
    color: byte;                                     { El color, claro }
    constructor init(TextoInicial: string);                 { Inicializa
}
    procedure FijaCoords(nuevoX, nuevoY: byte);             { Pues eso }
```

```
    procedure FijaTexto(mensaje: string);                  { Idem }
    procedure FijaColores(pluma,fondo: byte);          { y lo otro }
    procedure Escribe; virtual;                   { Lo escribe, claro }
    procedure done;                                  { Libera memoria }
  end;
{ --- Desarrollo del objeto Titulo --- }
constructor titulo.init(TextoInicial: string);
begin
  new(texto);                             { Reservamos memoria }
  texto^ := TextoInicial;                 { Damos valor al texto }
  x :=1 ; y := 1;                         { Coordenadas por defecto }
  color := 7;                             { Gris sobre negro }
end;

{ [...] }
{ (El resto no cambia). }
```

Tampoco hay problemas, ¿no? Pues ahora vamos a rizar el rizo, y vamos a hacer que el propio **objeto** a su vez sea un **puntero**. No voy a repetir el motivo por el que nos puede interesar, y voy a centrarme en los cambios.

Curso de Pascal. Tema 16.8: Programación Orientada a Objetos (8).

Ahora nos hará falta un nuevo "new" para el objeto y un nuevo "dispose":

```
new(t1)
t1^.init('Texto de ejemplo');
...
t1^.done;
dispose(t1);
```

Pero aún hay más: al liberar un objeto dinámico, existe una palabra clave que nos garantiza que se va a liberar la cantidad de memoria justa (especialmente si nuestros objetos dinámicos contienen a su vez datos dinámicos, etc). Es la palabra "**destructor**" que reemplaza a "procedure" (igual que hacía "constructor").

Entonces, nuestro último método quedaría

```
destructor titulo.done;
begin
  dispose(texto);                          { Liberamos la memoria }
end;
```

Se puede definir más de un destructor para una clase, y como es fácil que lo heredemos sin reescribirlo, puede resultar conveniente definirlo siempre como "virtual".

Vamos a ver cómo quedaría ahora nuestro programita:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Objetos y punteros-3  }
{    OBJETOP3.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program ObjetosConPunteros3;
uses crt;                                  { Usaremos "GotoXY" y TextAttr }
type                                      { Aquí definimos nuestro objeto }
  pTitulo = ^titulo;
  titulo = object
    texto: ^string;                          { El texto que se escribirá }
    x,y : byte;                                        { En qué posición }
    color: byte;                                        { El color, claro }
    constructor init(TextoInicial: string);                 { Inicializa }
    procedure FijaCoords(nuevoX, nuevoY: byte);               { Pues eso }
    procedure FijaTexto(mensaje: string);                         { Idem }
    procedure FijaColores(pluma,fondo: byte);                { y lo otro }
    procedure Escribe; virtual;                       { Lo escribe, claro }
    destructor done; virtual;                            { Libera memoria }
  end;
{ --- Desarrollo del objeto Titulo --- }
constructor titulo.init(TextoInicial: string);
begin
  new(texto);                            { Reservamos memoria }
  texto^ := TextoInicial;                { Damos valor al texto }
  x :=1 ; y := 1;                        { Coordenadas por defecto }
  color := 7;                            { Gris sobre negro }
end;
procedure titulo.FijaCoords(nuevoX, nuevoY: byte);
begin                                    { Definimos el procedimiento: }
  x := nuevoX;                             { Actualiza las coordenadas }
  y := nuevoY;
end;
procedure titulo.FijaTexto(mensaje: string);
begin                                              { Actualiza el texto }
  Texto^ := Mensaje;
end;
procedure titulo.FijaColores(pluma,fondo: byte);
begin                                    { Definimos el procedimiento: }
  color := pluma + fondo*16;                       { Actualiza el color }
end;
procedure titulo.Escribe;
begin
  TextAttr := color;                                { Muestra el título
}
  Gotoxy(X,Y);
  Write(Texto^);
end;
destructor titulo.done;
begin
```

```
    dispose(texto);                    { Liberamos la memoria }
  end;
  var
    t1: pTitulo;
  { -- Cuerpo del programa --}
  begin
    ClrScr;
    new(T1);
    T1^.Init('Por defecto');
    T1^.Escribe;
    T1^.FijaCoords(37,12);
    T1^.FijaColores(14,2);
    T1^.FijaTexto('Modificado');
    T1^.Escribe;
    T1^.Done;
    dispose(T1);
  end.
```

Finalmente, Turbo Pascal amplía también la **sintaxis** de la orden "**new**" para permitir reservar la memoria e inicializar el objeto en un solo paso, de modo que

```
new(T1);
T1^.Init('Por defecto');
```

se puede escribir como

```
new(T1, Init('Por defecto'));
```

es decir, ejecutamos "new" con dos parámetros: el objeto para el que queremos reservar la memoria, y el procedimiento que se va a encargar de inicializarlo. Lo mismo ocurre con "**dispose**". Así, el cuerpo de nuestro programa queda:

```
begin
  ClrScr;
  new(T1, Init('Por defecto'));
  T1^.Escribe;
  T1^.FijaCoords(37,12);
  T1^.FijaColores(14,2);
  T1^.FijaTexto('Modificado');
  T1^.Escribe;
  dispose(T1, Done);
end.
```

Pues esto es la OOP en Pascal. Este último apartado y el anterior (objetos dinámicos y métodos virtuales) son, a mi parecer, los más difíciles y los más

pesados. No todo el mundo va a utilizar objetos dinámicos, pero es raro no tener que usar nunca métodos virtuales.

Así que mi consejo es el de siempre: **experimentad**, que muchas veces es como más se aprende.

Curso de Pascal. Tema 17: El entorno Turbo Vision.

Turbo Vision es un **entorno**, el mismo que usa el IDE (entorno de desarrollo) de Turbo Pascal 6.0 y 7.0. Este entorno permite crear menús desplegables, ventanas de diálogo, visualizadores de listas y de textos, botones de elección múltiple, etc. Podemos manejarlo con el teclado o con el ratón.

(Para FreePascal, existe un "clon" de TurboVision, llamado FreeVision, de modo que estos fuentes deberían funcionar casi sin cambios. Hoy en día parece poco razonable usar FreePascal para crear "entornos avanzados en modo texto", existiendo Lazarus que permite crear entornos similares en modo gráfico, pero conservaremos este apartado por completitud y por si alguien quiere crear algo de este estilo para "resucitar" un ordenador basado en MsDos)

Cualquiera que haya manejado cualquiera de estas dos versiones de Turbo Pascal habrá visto que resulta bastante cómodo. Además, se incluyen las unidades necesarias para crear entornos como este. En eso nos vamos a centrar en esta lección.

No pretendo contarlo con toda la profundidad que sería deseable. Como siempre, prefiero despertar la curiosidad en aquellos que no lo conozcais, y si os interesa teneis un montón de **ejemplos** con Turbo Pascal, con un nivel de dificultad creciente. Estos ejemplos vienen además explicados en la "**Guía de Referencia de Turbo Vision**", un libro de ¡618 páginas! (en el caso de TP7.0 en inglés). Y si no se os da bien el inglés, teneis muchos libros traducidos en los que podreis aprender muchísimo, o bien preguntar aquí las dudas que os vayan surgiendo.

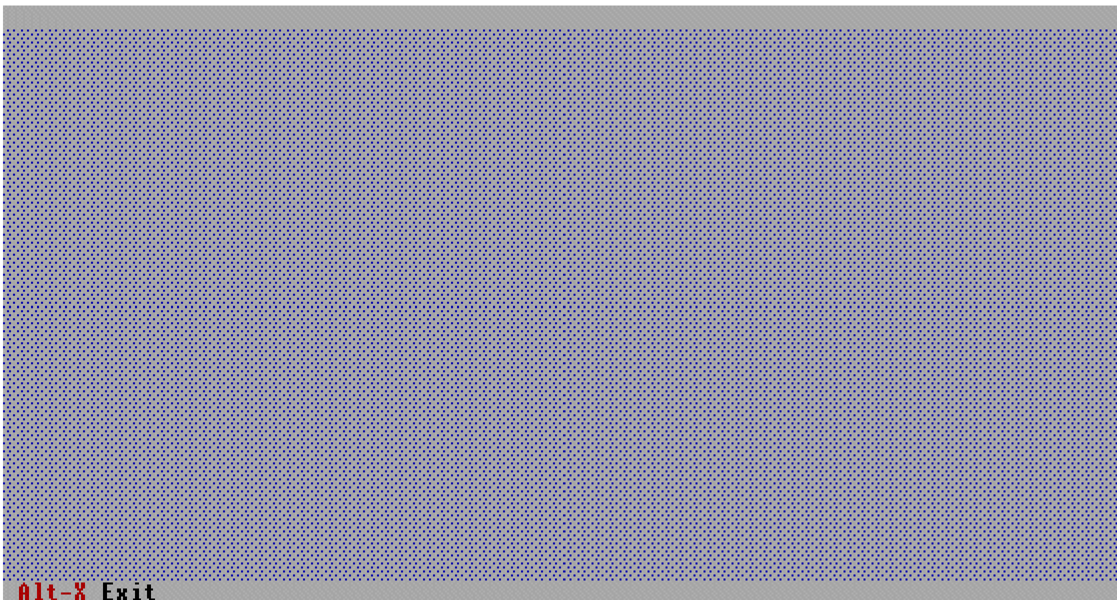
Además, ya vereis que el tema en sí no es sencillo. Si no entendeis algo, probad a seguir leyendo a ver si algún párrafo posterior lo deja claro, y si no, experimentad con el compilador o preguntad...

Vamos allá... El esqueleto de una aplicación básica se encuentra en la unidad "**App**". En ella tenemos definida la clase (objeto) **TApplication**, que vamos a usar para nuestro primer ejemplo. En éste sólo usaremos tres de los

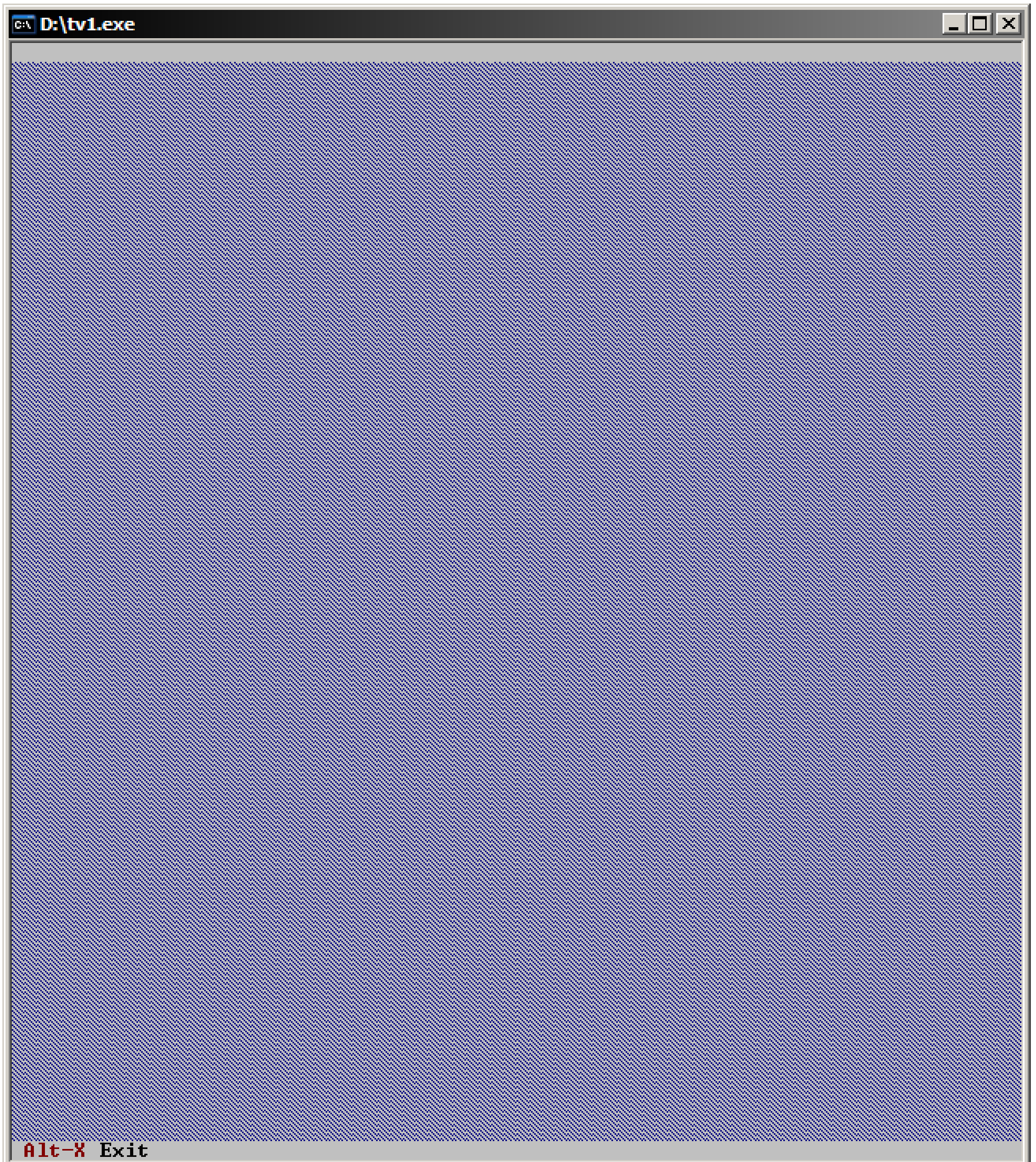
métodos: **Init** (el constructor), **Done** (el destructor) y **Run** (el cuerpo de la aplicación en sí).

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 1      }
{    TV1.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.2.2w  }
{--------------------------}

program TV1;
uses App;

var
  Prog: TApplication;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Con esta pequeñez ya nos aparece una pantalla parcialmente sombreada, por la que podemos pasear el ratón a nuestro gusto, y un letrero inferior que dice "Alt-X Exit". Pues sí, funciona: si pulsamos Alt-X o pinchamos encima de ese letrero con el ratón, sale (exit) del programa.



Si usamos FreePascal, la apariencia de la pantalla será muy similar, con la diferencia de que, si no indicamos lo contrario, se tomará la reslución de pantalla más alta posible, de modo que tendremos 50 líneas de texto en pantalla:



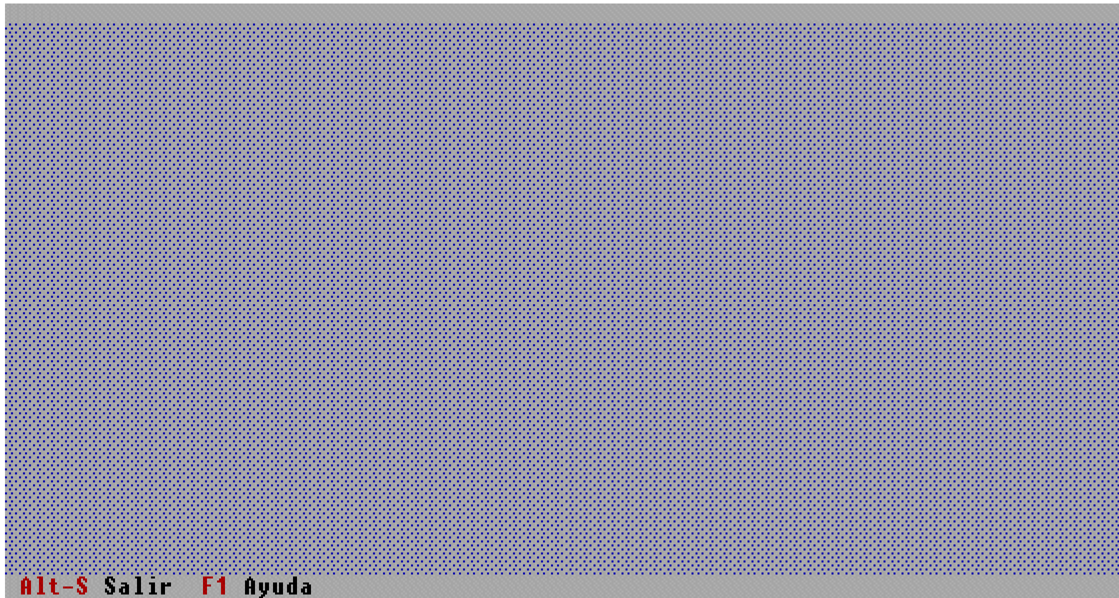
Y estamos en OOP, así que no sólo podemos heredar esta aplicación base, sino que además podemos modificarla y **ampliarla**...

Vamos a empezar por modificar esa línea inferior para que al menos aparezca en español... ;-)

Esta línea se llama "**línea de estado**" (en inglés Status Line) y sus características se fijan con el método "**InitStatusLine**". Ahora nuestro programa quedaría:

```
{--------------------------}
{  Ejemplo en Pascal:       }
{                           }
{    Turbo Vision - 2       }
{    TV2.PAS                }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{--------------------------}
program TV2;
uses App, Objects, Menus, Drivers, Views;
const
  cmAyuda = 100;    { Una orden que vamos a crear }
type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
  end;
procedure Programa.InitStatusLine;
var
  R: TRect;               { Rectángulo de pantalla }
begin
  GetExtent(R);           { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;     { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
      nil)),
    nil)));
end;
var Prog: Programa;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Esto es lo que obtendremos:



Comentarios generales:

InitStatusLine es un método **virtual** (espero que recordeis lo que era eso, de los temás de OOP), luego al heredarlo deberá seguir siéndolo.

TRect es el tipo "rectángulo" que usaremos para trabajar con partes de la pantalla. Está definido en la unidad Objects. Así, empezamos por ver el tamaño de la pantalla con GetExtent y lo guardamos en el rectángulo R. Luego modificamos este tamaño para que pase a ser el que nos interesa: queremos que la línea de estado sea la inferior, lo que hace orden

```
R.A.Y := R.B.Y - 1;
```

en la que A es la posición superior izquierda de la pantalla y B es la inferior derecha, cada una con sus coordenadas X e Y. Por tanto, lo que hace esta orden es que la coordenada Y de la parte superior de nuestro rectángulo sea un unidad menor que la de la parte inferior.

Por tanto, si nuestra pantalla terminaba en la línea 25, ahora nuestra línea de estado termina en la línea 25 y empieza en la 24. Esto no es del todo exacto, pero me niego a darle más vueltas porque esto es "lo fácil", así que quedaos simplemente con la idea de que estamos reservando una única línea para nuestra StatusLine y que es la inferior.

Después creamos la línea de estado, usando la sintaxis extendida de "**new**" que vimos el último día de OOP. Daos cuenta de como vamos encadenando las opciones: cada una aparece como último parámetro de la anterior, y terminamos con "nil".

Al crear la StatusLine le indicamos el rectángulo en el que queremos que aparezca (R) y la definición en sí (StatusDef, en la unidad Menus), que son los rangos de ayuda (desde 0 hasta $FFFF) y cada una de las opciones.

¿Rangos de ayuda? Sí, supongo que os habreis fijado que el IDE de TP 6 y 7 tiene una línea inferior de estado, y que se convierte en "ayuda" cuando os paseais por el menú superior. Eso se hace con esos "rangos de ayuda". En nuestro caso, siempre (desde 0 hasta $FFFF=65535) van a aparecer las mismas opciones abajo.

Y en estas opciones indicamos: el texto con una parte realzada, que se marca entre dos caracteres ASCII 126 (la tilde que va sobre la "ñ"), la tecla a la que queremos que responda (constantes cuyo nombre, por convenio, empieza por "kb" y que están definidas en la unidad Drivers), y el comando que queremos que ejecute (constantes también, pero que ahora empiezan por "cm" y que hemos definido nosotros).

Aun así, a lo mejor os habeis dado cuenta de un "pequeño" problema: hemos "creado" la orden cmAyuda, pero no hemos dicho lo que queremos que haga cuando le demos esas órdenes. Si ejecutais el programa, os dareis cuenta seguro }:-) porque no podreis hacer nada con ella: el teclado no hace nada si pulsais F1 (sí responde a Alt-S, porque el comando cmQuit está predefinido en la unidad Views), y si pulsais con el ratón sobre el letrero "F1 Ayuda" aparece un fondo verde que parece decir "si, ya te oigo, pero ¿qué quieres que haga?".

Pues vamos a corregir esto también. Necesitamos una rutina que "**maneje sucesos**" (Handle Events, en inglés), a la que le diremos "cuando lo que ocurra sea que te llegue la orden _cmAyuda_, escribe un mensaje que yo te voy a decir..."

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 3      }
{    TV3.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program TV3;
uses App, Objects, Menus, Drivers, Views, MsgBox;
const
  cmAyuda = 100;    { Una orden que vamos a crear }
```

```
type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
  end;
procedure Programa.InitStatusLine;
var
  R: TRect;                  { Rectángulo de pantalla }
begin
  GetExtent(R);              { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;        { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
      nil)),
    nil)));
end;
procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmAyuda:
         MessageBox('¡Aún no hay ayuda disponible!',
           nil, mfWarning or mfOKButton);
    end;
end;
var Prog: Programa;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Pues el **HandleEvent** primero hace lo que haya heredado (y así no nos preocupamos nosotros de órdenes como **cmQuit**). Luego mira si se trata de una orden (también podía ser una pulsación de tecla, un movimiento del ratón, etc), y si ese comando es el que nos interesa.

En caso de que sea la solicitud de ayuda escribe una ventana con un mensaje (**Message Box**), con un cierto texto, otros parámetros opcionales (en este caso ninguno: nil) y unos indicadores (flags: mf es abreviatura de**Message Flags**): mfWarning para que aparezca el título "Warning" y mfOKButton para que salga el boton OK. Si queremos traducir alguno de estos mensajes no tenemos más que modificar la unidad MsgBox, o bien crear nuestra propia versión a partir de ella (heredándola).

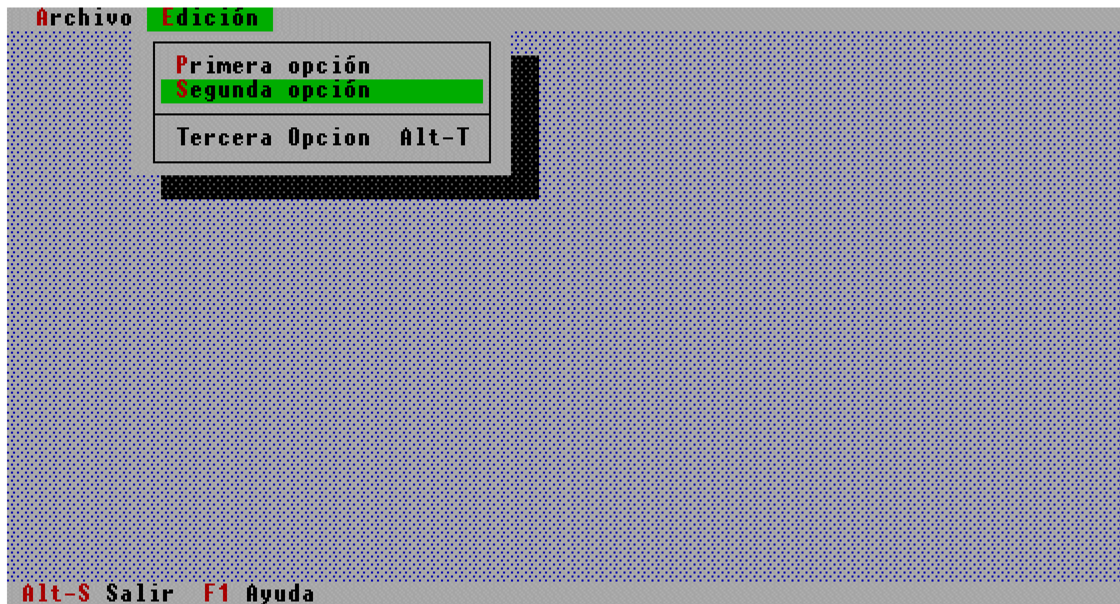
(De hecho, crearemos nuestro propio MessageBox en el apartado 3 de esta lección).

Finalmente, vamos a crear también un **menú desplegable** en la parte superior de la pantalla (una barra de menú, en inglés **Menu Bar**):

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 4      }
{    TV4.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program TV4;
uses App, Objects, Menus, Drivers, Views, MsgBox;
const
  cmAyuda = 100;    { Una orden que vamos a crear }
  cmAunNo = 101;    { Y otra }
type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure InitMenuBar; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
  end;
procedure Programa.InitStatusLine;
var
  R: TRect;                  { Rectángulo de pantalla }
begin
  GetExtent(R);              { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;        { Nos quedamos la línea inferior }
```

```
    New(StatusLine, Init(R,
      NewStatusDef(0, $FFFF,
        NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
        NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
        nil)),
      nil)));
  end;
  procedure Programa.InitMenuBar;
  var
    R: TRect;
  begin
    GetExtent(R);
    R.B.Y := R.A.Y + 1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
      NewSubMenu('~A~rchivo', hcNoContext, NewMenu(
        NewItem('~A~yuda', 'F1', kbF1, cmAyuda, hcNoContext,
        NewItem('~S~alir', 'Alt-S', kbAltS, cmQuit, hcNoContext,
        nil))),
      NewSubMenu('~E~dición', hcNoContext, NewMenu(
        NewItem('~P~rimera opción', '', kbnoKey, cmAunNo, hcNoContext,
        NewItem('~S~egunda opción', '', kbNoKey, cmAunNo, hcNoContext,
        NewLine(
        NewItem('Tercera Opcion', 'Alt-T', kbAltT, cmAunNo,
hcNoContext,
        nil))))),
      nil)))));
  end;
  procedure Programa.HandleEvent(var Suceso: TEvent);
  var
    R: TRect;
  begin
    inherited HandleEvent(Suceso);       { Primero que mire el "padre"
}
    if Suceso.What = evCommand then      { Si es una orden }
      case Suceso.Command of             { Miramos qué orden es }
        cmAyuda:
           MessageBox('¡Aún no hay ayuda disponible!',
             nil, mfWarning or mfOKButton);
        cmAunNo:
           MessageBox(#3+'Esta opción tampoco funciona pero al menos'
             +' está centrada en la ventana.  ;-)',
             nil, mfInformation or mfOkCancel);
      end;
  end;
  var Prog: Programa;
  begin
    Prog.Init;
    Prog.Run;
    Prog.Done;
  end.
```

Hay pocas cosas nuevas que comentar, porque el método InitMenuBar recuerda mucho al InitStatusLine, con la diferencia de que ahora hay **SubMenús** y elementos (**Items**), y que pueden tener un código de ayuda sensible al contexto cada uno de ellos (en este caso no se ha usado: todos son **hcNoContext**).



Un detalle que a lo mejor os ha extrañado es eso del "#3" que aparece delante del segundo mensaje. Vayamos por partes: escribir **#3** es lo mismo que escribir **chr(3)**. ¿Y por qué chr(3)? Pues es una cosa curiosa ;-) ... El carácter 3 es el que se devuelve al pulsar **Ctrl+C** en el teclado. Así que si un texto empieza por Ctrl+C, Turbo Vision considera que se trata del código de control de **centrado**, e intenta colocarlo en el centro de la ventana activa. Este "truquito" sirve también en otras ventanas, no sólo en MessageBox.

Por cierto, estos programas están comprobados con Turbo Pascal 7.0. Con la versión 6.0 deberían funcionar, con la única diferencia de que no existe la palabra clave "**inherited**", luego habría que cambiar

```
inherited HandleEvent;
```

por

```
TApplication.HandleEvent;
```

Curso de Pascal. Tema 17.2: Turbo Vision - Ventanas estándar.

Dentro de nuestra aplicación realizada con Turbo Vision podemos emplear fácilmente una serie de ventanas predefinidas. En la lección anterior vimos como crear "cajas de mensaje" (**MessageBox**). Ahora vamos a comenzar por usar escribir mensajes que ocupen un poco más:.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 5      }
{    TV5.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
```

```
{  por Nacho Cabanes         }
{                            }
{  Comprobado con:           }
{    - Free Pascal 2.2.0w    }
{    - Turbo Pascal 7.0      }
{--------------------------}

program TV5;

uses App, Objects, Menus, Drivers, Views, MsgBox;

const
  cmAyuda = 100;    { Una orden que vamos a crear }
  cmAunNo = 101;    { Y otra }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure InitMenuBar; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
  end;

procedure Programa.InitStatusLine;
var
  R: TRect;                 { Rectángulo de pantalla }
begin
  GetExtent(R);             { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;       { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
      nil)),
    nil)));
end;

procedure Programa.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~A~rchivo', hcNoContext, NewMenu(
      NewItem('~A~yuda', 'F1', kbF1, cmAyuda, hcNoContext,
      NewItem('~S~alir', 'Alt-S', kbAltS, cmQuit, hcNoContext,
      nil))),
    NewSubMenu('~E~dición', hcNoContext, NewMenu(
      NewItem('~P~rimera opción', '', kbnoKey, cmAunNo, hcNoContext,
      NewItem('~S~egunda opción', '', kbNoKey, cmAunNo, hcNoContext,
      NewLine(
      NewItem('Tercera Opcion', 'Alt-T', kbAltT, cmAunNo,
hcNoContext,
      nil))))),
    nil)))));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
```

```
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmAyuda:
         MessageBox('¡Aún no hay ayuda disponible!',
           nil, mfWarning or mfOKButton);
      cmAunNo:
         MessageBox('Esta opción no funciona'#13
           +'y además ni siquiera'#13
           +'cabe en la ventana.'#13
           +'Esta línea aún se ve'#13
           +'y esta ya no se ve.',
           nil, mfInformation or mfOkCancel);
    end;
end;

var Prog: Programa;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Hemos puesto 5 líneas de texto y hemos desbordado la capacidad de la ventana. Por supuesto, debe haber alguna forma de evitar esto. Pues sí, la hay, y además es sencilla (todavía ;-)

Vamos a crear un "**rectángulo**" del tamaño que nos interese:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 6      }
{    TV6.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program TV6;

uses App, Objects, Menus, Drivers, Views, MsgBox;

const
  cmAyuda = 100;    { Una orden que vamos a crear }
  cmAunNo = 101;    { Y otra }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure InitMenuBar; virtual;
```

```
    procedure HandleEvent(var Suceso: TEvent); virtual;
  end;

procedure Programa.InitStatusLine;
var
  R: TRect;               { Rectángulo de pantalla }
begin
  GetExtent(R);           { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;     { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
      nil)),
    nil)));
end;

procedure Programa.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~A~rchivo', hcNoContext, NewMenu(
      NewItem('~A~yuda', 'F1', kbF1, cmAyuda, hcNoContext,
      NewItem('~S~alir', 'Alt-S', kbAltS, cmQuit, hcNoContext,
      nil))),
    NewSubMenu('~E~dición', hcNoContext, NewMenu(
      NewItem('~P~rimera opción', '', kbnoKey, cmAunNo, hcNoContext,
      NewItem('~S~egunda opción', '', kbNoKey, cmAunNo, hcNoContext,
      NewLine(
      NewItem('Tercera Opcion', 'Alt-T', kbAltT, cmAunNo,
hcNoContext,
      nil))))),
    nil)))));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);      { Primero que mire el "padre"
}
  if Suceso.What = evCommand then     { Si es una orden }
    case Suceso.Command of            { Miramos qué orden es }
      cmAyuda:
         MessageBox('¡Aún no hay ayuda disponible!',
           nil, mfWarning or mfOKButton);
      cmAunNo:
         begin
         R.Assign(10,10,40,21);
         MessageBoxRect(R,'Esta opción no funciona'#13
           +'y además ni siquiera'#13
           +'cabe en la ventana.'#13
           +'Esta línea aún se ve'#13
           +'y esta... ¡también! :-)',
           nil, mfInformation or mfOkCancel);
         end;
    end;
end;
```

```
var Prog: Programa;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Esta vez no hemos usado cosas como R.A.Y := R.B.Y - 1 para cambiar el tamaño de la ventana (aunque no se trate de toda la pantalla, como vimos, sino sólo de "un trozo", R.A sigue representado la esquina superior izquierda y R.B la inferior derecha), sino que directamente le hemos dado un tamaño con **R.Assign(xMin,yMin,xMax,yMax)**: indicamos las coordenadas x e y de principio y de fin.

Nos puede parecer más intuitivo indicar el tamaño y la posición: pensar "quiero una ventana de 30x12, situada en la posición (5,5)". Pues entonces fijamos el tamaño con R.Assign, y la posición con **R.Move**. Nuestro programa quedaría:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 7      }
{    TV7.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
```

```
{--------------------------}

program TV7;

uses App, Objects, Menus, Drivers, Views, MsgBox;

const
  cmAyuda = 100;    { Una orden que vamos a crear }
  cmAunNo = 101;    { Y otra }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure InitMenuBar; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
  end;

procedure Programa.InitStatusLine;
var
  R: TRect;                  { Rectángulo de pantalla }
begin
  GetExtent(R);              { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;        { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
      nil)),
    nil)));
end;

procedure Programa.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~A~rchivo', hcNoContext, NewMenu(
      NewItem('~A~yuda', 'F1', kbF1, cmAyuda, hcNoContext,
      NewItem('~S~alir', 'Alt-S', kbAltS, cmQuit, hcNoContext,
      nil))),
    NewSubMenu('~E~dición', hcNoContext, NewMenu(
      NewItem('~P~rimera opción', '', kbnoKey, cmAunNo, hcNoContext,
      NewItem('~S~egunda opción', '', kbNoKey, cmAunNo, hcNoContext,
      NewLine(
      NewItem('Tercera Opcion', 'Alt-T', kbAltT, cmAunNo,
hcNoContext,
      nil))))),
    nil)))));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmAyuda:
         MessageBox('¡Aún no hay ayuda disponible!',
```

```
          nil, mfWarning or mfOKButton);
     cmAunNo:
        begin
        R.Assign(0,0,30,12);
        R.Move(5,5);
        MessageBoxRect(R,'Esta opción no funciona'#13
          +'y además ni siquiera'#13
          +'cabe en la ventana.'#13
          +'Esta línea aún se ve'#13
          +'y ésta... ¡también! :-)',
          nil, mfInformation or mfOkCancel);
        end;
   end;
end;

var Prog: Programa;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

La unit **MsgBox** es la que nos ha dado esta posibilidad de crear ventanas de aviso con facilidad, pero también nos permite hacer otras cosas muy frecuentes...

Vamos a ver cómo "**preguntar cosas**" a los usuarios de nuestros programas. Para ello usaremos la función **InputBox**, que tiene el formato:

```
function InputBox(Titulo: String; Etiqueta: String;
  var S: String; Limite: Byte): Word;
```

En nuestro caso, vamos a leer el nombre, de 40 letras, y a ver cómo comprobar si se ha cancelado la operación:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 8      }
{    TV8.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program TV8;
```

```
uses App, Objects, Menus, Drivers, Views, MsgBox;

const
  cmAyuda = 100;    { Una orden que vamos a crear }
  cmAunNo = 101;    { Y otra }
  cmNombre = 102;   { Y otra más }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure InitMenuBar; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure PideNombre;
  end;

procedure Programa.InitStatusLine;
var
  R: TRect;                 { Rectángulo de pantalla }
begin
  GetExtent(R);             { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;       { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F1~ Ayuda', kbF1, cmAyuda,
      nil)),
    nil)));
end;

procedure Programa.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~A~rchivo', hcNoContext, NewMenu(
      NewItem('~A~yuda', 'F1', kbF1, cmAyuda, hcNoContext,
      NewItem('~S~alir', 'Alt-S', kbAltS, cmQuit, hcNoContext,
      nil))),
    NewSubMenu('~E~dición', hcNoContext, NewMenu(
      NewItem('~P~rimera opción', '', kbnoKey, cmAunNo, hcNoContext,
      NewItem('~S~egunda opción', '', kbNoKey, cmAunNo, hcNoContext,
      NewLine(
      NewItem('Tercera Opcion', 'Alt-T', kbAltT, cmAunNo,
hcNoContext,
      nil))))),
    NewSubMenu('~N~ombre', hcNoContext, NewMenu(
      NewItem('~I~ntroducir Nombre', 'Alt-I', kbAltI, cmNombre,
        hcNoContext,
      nil)),
    nil))))));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmAyuda:
         MessageBox('¡Aún no hay ayuda disponible!',
```

```
            nil, mfWarning or mfOKButton);
        cmAunNo:
           MessageBox('Esta opción aún no está lista',
             nil, mfWarning or mfOKButton);
        cmNombre:
           PideNombre;
     end;
end;

procedure Programa.PideNombre;
var
  nombre: string;        { El nombre que leeremos }
  result: word;          { El resultado de la petición }
  result2: word;         { y el de la confirmación }
  correcto: boolean;     { Para repetir mientras queramos }
begin
  nombre := 'Nacho';                   { Valor por defecto }
  repeat
    correcto := true;                  { Suponemos que todo va bien }
    result := InputBox( 'Petición de datos',
      'Su nombre es:', nombre, 20);
    if result = cmCancel then          { Si cancelamos, pide confirmación
}
      begin
      result2 := MessageBox('Confirme que quiere cancelar',
            nil, mfWarning or mfYesButton or mfNoButton);
      if result2 = cmNo then correcto := false;
      end;
  until correcto;
end;

var Prog: Programa;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Con esto hemos visto de paso cómo interpretar las respuestas que nos devuelven las ventanas de avisos o de entrada de datos.

Eso sí, si probais a escribir mensajes largos, vereis que limita la longitud de los datos a 20 letras, como esperábamos, pero el recuadro azul es más grande, y eso desconcierta un poco. Lo ideal sería que el recuadro azul fuese también del tamaño adecuado, para darnos una idea de cuanto podemos escribir.

Pero eso, junto con la forma de traducir las ventanas, lo veremos en el siguiente apartado.

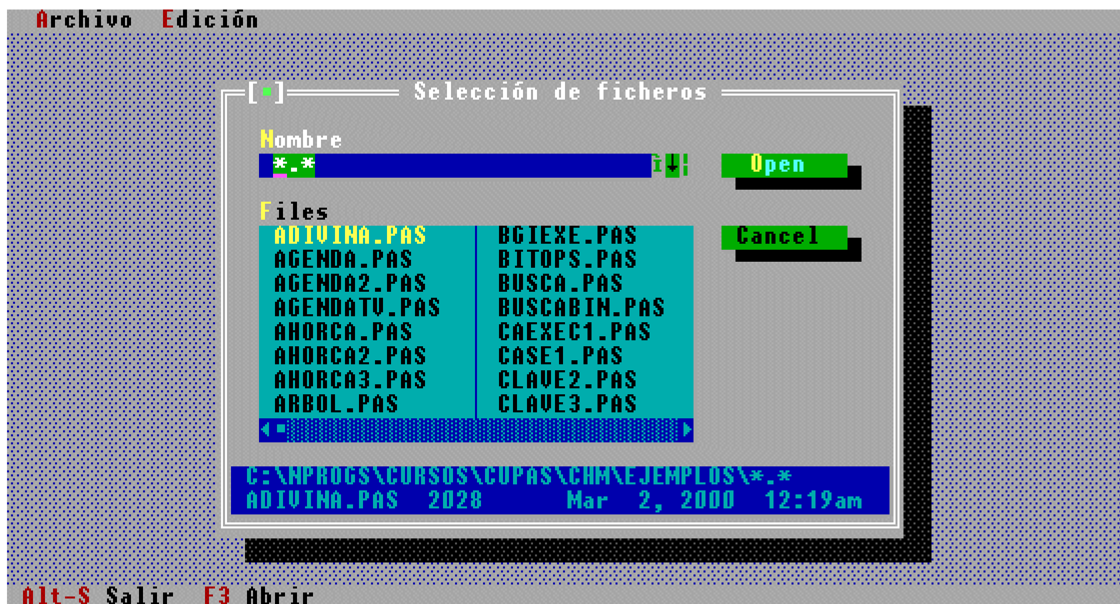
En último lugar, pero no por ello menos importante, vamos a ver otra ventana estándar, esta vez de la unidad StdDlg. No comento lo que es para obligaros a probarlo primero:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Turbo Vision - 9      }
{    TV9.PAS               }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program TV9;

uses App, Objects, Menus, Drivers, Views, Dialogs, MsgBox, StdDlg;

const
  cmAbrir = 100;    { Una orden que vamos a crear }
```

```
  cmAunNo = 101;    { Y otra }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure InitMenuBar; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Abre;
  end;

procedure Programa.InitStatusLine;
var
  R: TRect;                 { Rectángulo de pantalla }
begin
  GetExtent(R);             { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;       { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~F3~ Abrir', kbF3, cmAbrir,
      nil)),
    nil)));
end;

procedure Programa.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~A~rchivo', hcNoContext, NewMenu(
      NewItem('~A~brir', 'F3', kbF3, cmAbrir, hcNoContext,
      NewItem('~S~alir', 'Alt-S', kbAltS, cmQuit, hcNoContext,
      nil))),
    NewSubMenu('~E~dición', hcNoContext, NewMenu(
      NewItem('~P~rimera opción', '', kbnoKey, cmAunNo, hcNoContext,
      NewItem('~S~egunda opción', '', kbNoKey, cmAunNo, hcNoContext,
      nil))),
    nil)))));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmAbrir: Abre;
      cmAunNo:
         MessageBox('Esta opción aún no está lista',
           nil, mfWarning or mfOKButton);
    end;
end;

procedure Programa.Abre;
var
  Nombre: string;
  Dlg: PDialog;
begin
  nombre := '*.*';
  Dlg := New(PFileDialog, Init(nombre, 'Selección de ficheros',
```

```
     '~N~ombre', fdOpenButton, 100));
   ExecuteDialog( Dlg, @nombre);
   if nombre <> '*.*' then
     MessageBox(#3'Ha escogido '+nombre,
       nil, mfInformation or mfOKButton);
end;

var Prog: Programa;

begin
   Prog.Init;
   Prog.Run;
   Prog.Done;
end.
```

Esta vez hemos creado una ventana de diálogo predefinida, que es la que se encarga de mostrar la lista de ficheros, pero la hemos creado de una forma distinta, más general.



Como esa forma de crear ventanas la veremos en el próximo apartado, no cuento nada más... }:-)

Curso de Pascal. Tema 17.3: Turbo Vision - Ventanas de diálogo.

Simplificando un poco el asunto, podemos distinguir dos tipos de ventanas en un entorno gráfico (a los que intenta imitar Turbo Vision): modales y no modales.

- Una ventana **modal** es aquella que toma por completo el control de la aplicación, y no nos deja hacer nada más hasta que la cerramos.

- Una ventana **no modal** es la que no toma el control de la aplicación, sino que podemos trabajar con otras ventanas sin necesidad de cerrar ésta.

¿Suena raro? Con un par de ejemplos seguro que se verá claro del todo. Vamos a fijarnos en el lector del curso. podemos tener abiertas a la vez varias ventanas con textos de distintas lecciones, y saltar de una o otra, pasearnos por el menú, etc. Son ventanas **no modales**. En cambio, accedemos a alguna de las opciones de ayuda, pulsando F1 por ejemplo, no podemos acceder al menú ni a las demás ventanas que hayamos abierto, a no ser que cerremos la actual. Es una ventana **modal**.

Insisto en que todo esto está simplificado. Y ahora voy a hacer una simplificación más: esas ventanas no modales es lo que llamaré "ventanas de texto", y las veremos en el siguiente apartado; las ventanas modales como las de ayuda son las que llamaré "**ventanas de diálogo**".

Esto no es del todo cierto: podemos hacer que ventanas como los InputBox y MessageBox que hemos visto no sean modales, y que una ventana de texto sí lo sea, pero eso me lo salto... al menos de momento.

Volvamos a lo práctico. Hemos visto cómo usar algunas ventanas predefinidas de Turbo Vision, que casi "se creaban solas". Ahora vamos a ver cómo crearlas nosotros.

Deberemos **crear** una ventana de diálogo y ejecutarla. Primer ejemplillo, directamente:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de diálogo   }
{    en Turbo Vision - 1   }
{    DLG1.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Dlg1;

uses App, Objects, Menus, Drivers, Views, Dialogs;
                                      { ^ Incluimos esta Unit }

const
  cmDialogo = 100;    { La orden que vamos a crear }
```

```
 type Programa = object (TApplication)
     procedure InitStatusLine; virtual;
     procedure HandleEvent(var Suceso: TEvent); virtual;
     procedure Dialogo;
   end;

 procedure Programa.InitStatusLine;
 var
   R: TRect;                 { Rectángulo de pantalla }
 begin
   GetExtent(R);             { Miramos cuando ocupa }
   R.A.Y := R.B.Y - 1;       { Nos quedamos la línea inferior }
   New(StatusLine, Init(R,
     NewStatusDef(0, $FFFF,
       NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
       NewStatusKey('~Alt-D~ Diálogo', kbAltD, cmDialogo,
       nil)),
     nil)));
 end;

 procedure Programa.HandleEvent(var Suceso: TEvent);
 var
   R: TRect;
 begin
   inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
   if Suceso.What = evCommand then       { Si es una orden }
     case Suceso.Command of              { Miramos qué orden es }
       cmDialogo: Dialogo;
     end;
 end;

 procedure Programa.Dialogo;
 var
   R: TRect;                             { El "rectángulo" que ocupará
}
   D: PDialog;                           { La ventana en sí }
 begin
   R.Assign(10,10,30,20);                { Definimos la zona de
pantalla }
   D := New(PDialog,                     { Inicializamos: nuevo diálogo
}
     Init(R, 'Prueba'));                 {   en la zona R, y título
Prueba }
   ExecuteDialog(D,nil);                 { Ejecutamos, sin parámetros }
 end;

 var Prog: Programa;

 begin
   Prog.Init;
   Prog.Run;
   Prog.Done;
 end.
```

Hemos creado una ventana de diálogo, pero estaba vacía. Ahora vamos a hacer que tenga algo en su interior.

Para hacer que tenga algo en su **interior**, bastará con crear un nuevo rectángulo y añadirlo antes de ejecutar el diálogo:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de diálogo   }
{    en Turbo Vision - 2   }
{    DLG2.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Dlg2;

uses App, Objects, Menus, Drivers, Views, Dialogs;

const
  cmDialogo = 100;    { La orden que vamos a crear }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Dialogo;
  end;

procedure Programa.InitStatusLine;
```

```
var
  R: TRect;                { Rectángulo de pantalla }
begin
  GetExtent(R);            { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;      { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-D~ Diálogo', kbAltD, cmDialogo,
      nil)),
    nil)));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmDialogo: Dialogo;
    end;
end;

procedure Programa.Dialogo;
var
  R: TRect;                             { El "rectángulo" que ocupará
}
  D: PDialog;                           { La ventana en sí }
begin
  R.Assign(10,10,30,20);                { Definimos la zona de
pantalla }
  D := New(PDialog,                     { Inicializamos: nuevo diálogo
}
    Init(R, 'Prueba'));                 {   en la zona R, y título
Prueba }
  R.Assign(2,2,10,5);                   { Definimos una zona interior
}
  D^.Insert(New(PStaticText,            {   e insertamos texto en ella
}
    Init(R, 'Prueba de texto estático')));
  ExecuteDialog(D,nil);                 { Ejecutamos, sin parámetros }
end;

var Prog: Programa;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Se va complicando, ¿verdad? }:-)

Ahora que ya sabemos cómo "tener cosas" en un diálogo, vamos ver cómo poner varias cosas de interés ;-) , por ejemplo **botones** y líneas de **introducción de texto**:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de diálogo   }
{    en Turbo Vision - 3   }
{    DLG3.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Dlg3;

uses App, Objects, Menus, Drivers, Views, Dialogs;

const
  cmDialogo = 100;    { La orden que vamos a crear }

type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Dialogo;
  end;

procedure Programa.InitStatusLine;
```

```
var
  R: TRect;                  { Rectángulo de pantalla }
begin
  GetExtent(R);              { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;        { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-D~ Diálogo', kbAltD, cmDialogo,
      nil)),
    nil)));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);          { Primero que mire el "padre"
}
  if Suceso.What = evCommand then         { Si es una orden }
    case Suceso.Command of                { Miramos qué orden es }
      cmDialogo: Dialogo;
    end;
end;

procedure Programa.Dialogo;
var
  R: TRect;                               { El "rectángulo" que ocupará
}
  D: PDialog;                             { La ventana en sí }
begin
  R.Assign(0,0,50,15);                    { Definimos el tamaño }
  R.Move(15,5);                           {  y la posición }
  D := New(PDialog,                       { Inicializamos: nuevo diálogo
}
    Init(R, 'Más vistoso'));              {   en la zona R, y título
Prueba }
  R.Assign(2,2,20,3);                     { Definimos una zona interior
}
  D^.Insert(New(PStaticText,              {   e insertamos texto en ella
}
    Init(R, 'Esto es un texto estático')));
  R.Assign(5,11,20,13);                   { Igual para el botón }
  D^.Insert(New(PButton,
    Init(R, '~A~dios', cmQuit, bfNormal)));
  R.Assign(10,7,40,8);                    { Y para la línea de entrada }
  D^.Insert(New(PInputLine,
    Init(R, 50)));                        {   de longitud 50 }
  ExecuteDialog(D,nil);                   { Ejecutamos, sin parámetros }
end;

var Prog: Programa;

begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Comentarios sobre este programa:

- Antes hemos visto que si un texto estático no cabía en una línea, seguía en las sucesivas hasta el final de la ventana.
- Ahora sólo había una línea; en cualquier caso, si el texto no cabe entero, se parte en un espacio entre palabras, no se rompen las palabras.
- El botón se define de forma parecida, pero parece que el comando cmQuit "no se ejecuta". Esto no es del todo cierto. Se ejecutaría para el HandleEvent propio del diálogo, no el de todo el programa. De momento sólo responde a "Esc", que le manda la señal de cancelar (cmCancel), a la que sí sabe responder. bfNormal es uno de los "button flags", indicadores específicos para los botones. Cuando tengamos varios botones en una ventana, podremos usar bfDefault en uno de ellos para indicar que ése será el botón por defecto (el que se pulsará con Intro).
- La línea de entrada de datos permite 50 caracteres de longitud, pero esta creada en un rectángulo de 30 (40-10) caracteres de anchura. Eso no es problema, porque no hace falta ver todo el texto a la vez, como podeis comprobar si tecleais un texto largo.

Finalmente, vamos a ver cómo añadir una **etiqueta** que identifique a esa línea de entrada de datos, y como trabajar con los **datos** de dicha línea:

```
{--------------------------}
{  Ejemplo en Pascal:      }
```

```
{                          }
{    Ventanas de diálogo   }
{    en Turbo Vision - 4   }
{    DLG4.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Dlg4;

uses App, Objects, Menus, Drivers, Views, Dialogs, MsgBox;

const
  cmDialogo = 100;    { La orden que vamos a crear }

type Programa = object (TApplication)
    texto: string;
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Dialogo;
  end;

procedure Programa.InitStatusLine;
var
  R: TRect;                { Rectángulo de pantalla }
begin
  GetExtent(R);            { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;      { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-D~ Diálogo', kbAltD, cmDialogo,
      nil)),
    nil)));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmDialogo: Dialogo;
    end;
end;

procedure Programa.Dialogo;
var
  R: TRect;                             { El "rectángulo" que ocupará
}
  D: PDialog;                           { La ventana en sí }
  Enlace: PView;                        { Para la enlazar la etiqueta
                                          con la InputLine }
```

```
begin
   Texto := 'Texto por defecto';         { Texto inicial }
   R.Assign(0,0,50,15);                  { Definimos el tamaño }
   R.Move(15,5);                         {  y la posición }
   D := New(PDialog,                     { Inicializamos: nuevo diálogo
}
     Init(R, 'Más vistoso'));            {   en la zona R, y título
Prueba }
   R.Assign(2,2,20,3);                   { Definimos una zona interior
}
   D^.Insert(New(PStaticText,            {   e insertamos texto en ella
}
     Init(R, 'Esto es un texto estático')));
   R.Assign(5,11,20,13);                 { Igual para el primer botón }
   D^.Insert(New(PButton,
     Init(R, '~C~ancelar', cmCancel, bfNormal)));
   R.Assign(25,11,40,13);                 { Y para el segundo }
   D^.Insert(New(PButton,
     Init(R, '~A~ceptar', cmOK, bfDefault)));
   R.Assign(10,7,40,8);                  { Y para la línea de entrada }
   Enlace := New(PInputLine, Init(R, 50));
   D^.Insert(Enlace);
   R.Assign(9,6,30,7);                  { Y la etiqueta asociada }
   D^.Insert(New(PLabel,
     Init(R, 'Texto:', Enlace)));
   ExecuteDialog(D,@Texto);             { Ejecutamos, CON parámetros }
   MessageBox('Ha tecleado: '+ Texto,   { Y mostramos el resultado }
            nil, mfInformation or mfOKButton);
end;

var Prog: Programa;

begin
   Prog.Init;
   Prog.Run;
   Prog.Done;
end.
```

En un diálogo también podemos poner casillas de verificación de opciones, visores de listas, etc., pero eso lo dejo para que lo investigue quien tenga muchas inquietudes.

Diseñar diálogos "a pelo" puede ser algo muy pesado. En muchas BBS y en Internet podremos encontrar utilidades que nos permiten hacerlo de forma "visual", como DLGDSN (Dialog Design, de L. David Baldwin)

*(**Nota:** todo esto está realizado con TP7; no lo he probado en TP6).*

Curso de Pascal. Tema 17.4: Turbo Vision - Ventanas de texto.

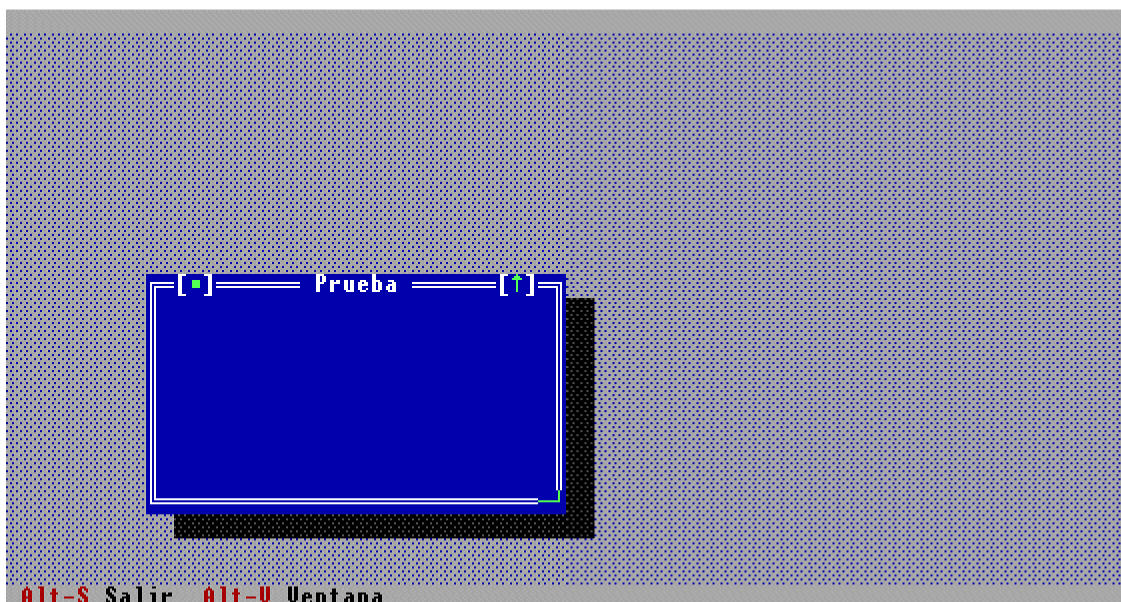
Ya hemos visto cómo crear ventanas de diálogo, que "paralizan" al resto de la aplicación mientras estén activas.

Ahora veremos cómo crear ventanas más generales, no modales (podremos hacer otras cosas mientras estén visibles), que usaremos para mostrar textos en pantalla.

Tomando la base de "Dlg1", la aplicación con la que creamos la primera ventana de diálogo, podemos hacer otra muy parecida, pero que ahora use ventanas "normales":

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de texto     }
{    en Turbo Vision - 1   }
{    VENT1.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program Vent1;
uses App, Objects, Menus, Drivers, Views;
const
  cmVentana = 100;    { La orden que vamos a crear }
type Programa = object (TApplication)
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Ventana;
  end;
procedure Programa.InitStatusLine;
var
  R: TRect;                 { Rectángulo de pantalla }
begin
  GetExtent(R);             { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;       { Nos quedamos la línea inferior }
```

```
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-V~ Ventana', kbAltV, cmVentana,
      nil)),
    nil)));
end;
procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmVentana: Ventana;
    end;
end;
procedure Programa.Ventana;
var
  R: TRect;                             { El "rectángulo" que ocupará
}
  W: PWindow;                           { La ventana en si }
begin
  R.Assign(10,10,40,20);                { Definimos el tamaño }
  W := New(PWindow,                     { Inicializamos: nueva ventana
}
    Init(R, 'Prueba',                   {   en la zona R, título
Prueba }
    wnNoNumber));                       {   y sin número de ventana }
  InsertWindow(W);                  { La insertamos en nuestro programa
}
end;
var Prog: Programa;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```



Esta vez hemos creado una ventana de texto, que recuerda a las que usa el entorno de Turbo Pascal cuando editamos nuestros programas. Como hicimos en el apartado anterior, hemos empezado por una ventana vacía, pero antes de añadir algo en su interior quiero remarcar un par de cosas:

- Esta ventana se comporta como las del IDE: si pinchamos con el ratón en la línea superior, podremos moverla; con la esquina inferior derecha podemos cambiar su tamaño; tiene un "botón" de zoom y otro de cierre...
- Así que pulsa Alt+V varias veces y mueve la ventana con el ratón. ¡Hay más ventanas! Y si pinchas en una de ellas, pasa a ser esa la "importante", a pesar de que hay otras abiertas, como queríamos.
- Pero ¿qué eso eso de wnNoNumber? Pues podemos hacer que las primeras 9 ventanas tengan un número asignado, con lo que podemos saltar de una a otra también pulsando Alt+1, Alt+2, etc. En este caso, con la constante "wnNoNumber" indicamos que no queremos tener números en las ventanas.

Así que antes de "llenar" la ventana, vamos a ver cómo darle un número. Lo haré de una forma más o menos estricta: podemos llevar cuenta del número de ventanas abiertas en cualquier parte de nuestro programa, pero lo más correcto es que sea nuestro objeto "Programa" quien lo gestione, así que le voy a añadir un campo y voy a redefinir el constructor Init. También haré que las ventanas aparezcan en posiciones aleatorias de la pantalla, para que se vean mejor:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de texto     }
{    en Turbo Vision - 2   }
{    VENT2.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}
program Vent2;
uses App, Objects, Menus, Drivers, Views;
const
  cmVentana = 100;    { La orden que vamos a crear }
type Programa = object (TApplication)
    NumeroVent: byte;              { Número de ventana }
    constructor Init;              { Vamos a ampliar el Init }
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Ventana;
  end;
```

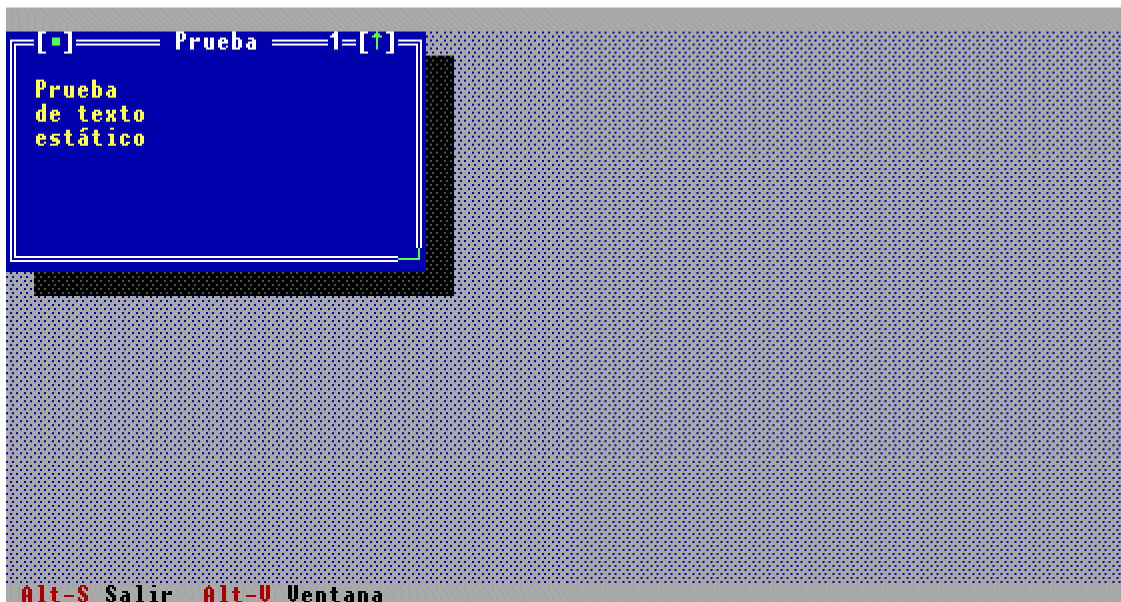
```
constructor Programa.Init;    { Nuestra propia versión del Init }
begin
  Inherited Init;             { Que hará lo mismo que la inicial }
  NumeroVent := 0;            { Pero además prepara la variable }
end;
procedure Programa.InitStatusLine;
var
  R: TRect;                 { Rectángulo de pantalla }
begin
  GetExtent(R);             { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;       { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-V~ Ventana', kbAltV, cmVentana,
      nil)),
    nil)));
end;
procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);          { Primero que mire el "padre"
}
  if Suceso.What = evCommand then         { Si es una orden }
    case Suceso.Command of                { Miramos qué orden es }
      cmVentana: Ventana;
    end;
end;
procedure Programa.Ventana;
var
  R: TRect;                               { El "rectángulo" que ocupará
}
  W: PWindow;                             { La ventana en si }
begin
  Inc(NumeroVent);                        { Una ventana más }
  R.Assign(0,0,30,10);                    { Definimos el tamaño }
  R.Move(random(50), random(13));
  W := New(PWindow,                       { Inicializamos: nuevo diálogo
}
    Init(R, 'Prueba',NumeroVent));                    {   en la zona R, y
título Prueba }
  InsertWindow(W);                  { Ejecutamos, sin parámetros }
end;
var Prog: Programa;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done;
end.
```

Ahora sí, ya vamos a escribir algo dentro de la ventana. Lo haremos exactamente igual que vimos para los diálogos, creando un rectángulo interior, que rellenaremos con "Static Text".

Como el tipo TStaticText está definido en la unidad Dialogs, tendremos que añadirla en nuestra sección de "uses".

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de texto     }
{    en Turbo Vision - 3   }
{    VENT3.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Vent3;

uses App, Objects, Menus, Drivers, Views, Dialogs;

const
  cmVentana = 100;   { La orden que vamos a crear }

type Programa = object (TApplication)
    NumeroVent: byte;              { Número de ventana }
    constructor Init;              { Vamos a ampliar el Init }
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
    procedure Ventana;
  end;

constructor Programa.Init;   { Nuestra propia versión del Init }
begin
  Inherited Init;            { Que hará lo mismo que la inicial }
  NumeroVent := 0;           { Pero además prepara la variable }
end;

procedure Programa.InitStatusLine;
var
  R: TRect;               { Rectángulo de pantalla }
begin
  GetExtent(R);           { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;     { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-V~ Ventana', kbAltV, cmVentana,
      nil)),
    nil)));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmVentana: Ventana;
    end;
```

```
end;

procedure Programa.Ventana;
var
   R: TRect;                           { El "rectángulo" que ocupará
}
   W: PWindow;                         { La ventana en si }
begin
   Inc(NumeroVent);                    { Una ventana más }
   R.Assign(0,0,30,10);                { Definimos el tamaño }
   R.Move(random(50), random(13));
   W := New(PWindow,                   { Inicializamos: nuevo diálogo
}
      Init(R, 'Prueba',NumeroVent));                 {   en la zona R, y
título Prueba }
   R.Assign(2,2,10,5);                 { Definimos una zona interior
}
   W^.Insert(New(PStaticText,          {   e insertamos texto en ella
}
      Init(R, 'Prueba de texto estático')));
   InsertWindow(W);                 { Ejecutamos, sin parámetros }
end;

var Prog: Programa;
begin
   Prog.Init;
   Prog.Run;
   Prog.Done;
end.
```



También podremos insertar botones, líneas de estado, etc. en una ventana de texto como ésta, pero el manejo se complica mucho comparado con los diálogos que vimos en el apartado anterior.

Finalmente, vamos a ver cómo cambiar la apariencia y propiedades de una ventana de texto.

Las ventanas tienen una serie de "flags", cuyo nombre empieza por "wf" y que nos permiten modificar su comportamiento:

- wfMove: Indica si queremos permitir que se mueva.
- wfGrow: Si puede cambiar de tamaño
- wfClose: Si se puede cerrar
- wfZoom: Si se puede hacer un zoom (ampliar hasta toda la ventana)

Recordemos que las ventanas tenían "botones" para hacer muchas de estas cosas. Pero Turbo Vision ya lo tiene en cuenta: si decimos que no se pueda cerrar una ventana, el botón de cierre no aparecerá siquiera.

Un último detalle antes de pasar al ejemplo: estos flags se almacenan como bits de un byte, que es el campo "flags". Por ello, deberemos modificarlos con "and" u "or". Por ejemplo, para fijar el bit wfClose asegurarnos de que una se podrá cerrar, podemos usar

```
flags := flags or wfClose;
```

y para que no se pueda cerrar sería

```
flags := flags and not wfClose;
```

Ahora ya, vamos a ver cómo quedaría el programita:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ventanas de texto     }
{    en Turbo Vision - 4   }
{    VENT4.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program Vent4;

uses App, Objects, Menus, Drivers, Views, Dialogs;

const
  cmVentana = 100;    { La orden que vamos a crear }

type Programa = object (TApplication)
    NumeroVent: byte;               { Número de ventana }
    constructor Init;               { Vamos a ampliar el Init }
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Suceso: TEvent); virtual;
```

```
    procedure Ventana;
  end;

constructor Programa.Init;   { Nuestra propia versión del Init }
begin
  Inherited Init;            { Que hará lo mismo que la inicial }
  NumeroVent := 0;           { Pero además prepara la variable }
end;

procedure Programa.InitStatusLine;
var
  R: TRect;               { Rectángulo de pantalla }
begin
  GetExtent(R);           { Miramos cuando ocupa }
  R.A.Y := R.B.Y - 1;     { Nos quedamos la línea inferior }
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-S~ Salir', kbAltS, cmQuit,
      NewStatusKey('~Alt-V~ Ventana', kbAltV, cmVentana,
      nil)),
    nil)));
end;

procedure Programa.HandleEvent(var Suceso: TEvent);
var
  R: TRect;
begin
  inherited HandleEvent(Suceso);        { Primero que mire el "padre"
}
  if Suceso.What = evCommand then       { Si es una orden }
    case Suceso.Command of              { Miramos qué orden es }
      cmVentana: Ventana;
    end;
end;

procedure Programa.Ventana;
var
  R: TRect;                             { El "rectángulo" que ocupará
}
  W: PWindow;                           { La ventana en si }
begin
  Inc(NumeroVent);                      { Una ventana más }
  R.Assign(0,0,30,10);                  { Definimos el tamaño }
  R.Move(random(50), random(13));
  W := New(PWindow,                     { Inicializamos: nuevo diálogo
}
    Init(R, 'Prueba',NumeroVent));                 {   en la zona R, y
título Prueba }
  R.Assign(2,2,10,5);                   { Definimos una zona interior
}
  W^.Insert(New(PStaticText,            {   e insertamos texto en ella
}
    Init(R, 'Prueba de texto estático')));
  W^.flags := W^.flags and not (wfClose or WfGrow);
  InsertWindow(W);                  { Ejecutamos, sin parámetros }
end;

var Prog: Programa;

begin
  Prog.Init;
```

```
   Prog.Run;
   Prog.Done;
end.
```

Como se ve, la ventana ya no tiene botón de cierre ni la esquina inferior de redimensionado, pero sí que se la puede mover o ampliar totalmente.

Pues con esto doy por finalizada la introducción a Turbo Vision. Con lo que hemos visto, uno ya puede hacer con una cierta facilidad que sus programas queden más "bonitos"... como mínimo.

El avanzar a partir de aquí, ya lo dejo a quien quiera. Con esta base confío en que asuste menos acercarse a fuentes que empleen Turbo Vision, o al manual de TP, o incluso a la ayuda en línea.

Incluyo un ejemplo, que sería nuestra super Agenda ;-), pasada a Turbo Vision. En ella se puede observar que he dibujado la ventana de texto de otra forma menos rígida que la que acabo de comentar: en vez de "insertar" texto estático, defino su interior como View y luego ya lo incluyo en la ventana. Esto es más versátil porque nos permite usar barras de desplazamiento, por ejemplo, pero también es bastante más complicado, y no sé si alguien lo llegará a usar, así que quien quiera trastear con ello, no tiene más que basarse en el ejemplo.

Si alguien quiere avanzar más y saber cómo poner también barras de desplazamiento y demás, le doy la pista clave: VIEWTEXT.PAS es un visor de texto, en el que me he basado yo para el lector de las primeras versiones del curso. Es uno de los ejemplos que trae Turbo Pascal 7, dentro del directorio

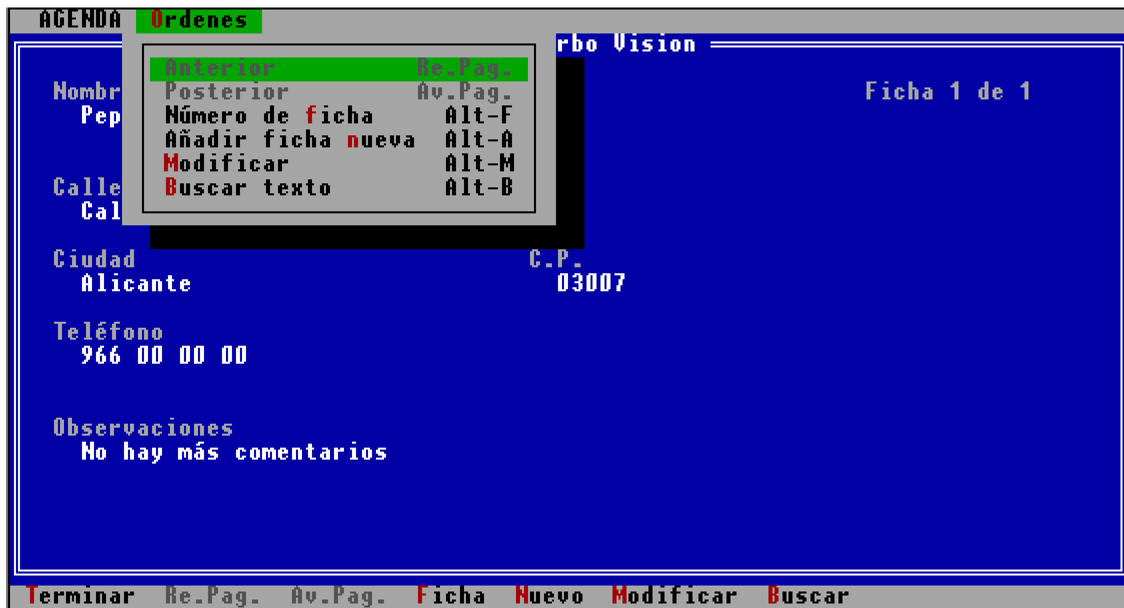
```
TP\EXAMPLES\TVFM
```

(o algo similar, según donde lo hayais instalado). Venga, a jugar... :-)

Curso de Pascal. Tema 17.5: Turbo Vision - Ejemplo.

Esta versión está basada en la segunda de la agenda.

La apariencia que tendrá esta versión será ésta:



Y el fuente podría ser así:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Agenda: versión para  }
{    Turbo Vision          }
{    AGENDATV.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}
{ =======================================================
  Ejemplito de Agenda, adaptado a Turbo Vision.
  Basado en el segundo ejemplo (MiniAgenda2)
  Nacho Cabanes, Junio 96, para CUPAS (Curso de Pascal)
  ======================================================= }
program MiniAgendaTV;
uses App,      { Para objeto TApplication }
  Drivers,   { TEvent, por ejemplo }
  Objects,   { TRect, etc }
  Menus,     { Pues eso ;-) }
  Views,     { Vistas y hcXX }
  MsgBox;    { MessageBox, InputBox, etc }
{$V-}
  { Directiva de compilación: tamaño de strings no estricto (ver
  ampliación 6).  Para poder usar LeeValor con Strings de cualquier
  tamaño }
const
  nombref: string[12]='agenda.dat';                 { Nombre del fichero
}
  longNOM = 20;
  longDIR = 30;
  longCIU = 15;
  longCP  =  5;
```

```
   longTLF = 12;
   longOBS = 40;
   cmSaludo =    1001;            { Las ordenes que podremos dar }
   cmSalir  =    1002;
   cmNumero =    1005;
   cmNuevo =     1006;
   cmModifica =  1007;
   cmBusca =     1008;
   cmAnterior =  101;              { Estas se podrán deshabilitar }
   cmPosterior = 102;
 type                                                  { Nuestro tipo de datos
}
    tipoagenda = record
       nombre:     string[ longNOM ];
       direccion: string[ longDIR ];
       ciudad:     string[ longCIU ];
       cp:         string[ longCP  ];
       telef:      string[ longTLF ];
       observ:     string[ longOBS ]
       end;
    programa = object (TApplication)          { Nuestro programa }
     { Heredados }
     procedure HandleEvent(var Event: TEvent); virtual;  { Manj
eventos }
     procedure InitMenuBar; virtual;                       { Barra de
menú }
     procedure InitStatusLine; virtual;                  { Línea de
estado }
     { Creados }
     procedure Saludo;                                  { Saludo al
entrar }
     procedure CreaInterior;                        { Ventana con el
texto }
     procedure Anterior;                             { Retrocede una
ficha }
     procedure Posterior;                                { Avanza una
ficha }
     procedure Numero;                                  { Salta a una
ficha }
     procedure Modifica;                               { Modifica la
actual }
     procedure Nuevo;                                          { Ficha
nueva }
     procedure Busca;                                       { Busca un
texto }
    end;
   PITexto = ^TITexto;   { Texto con los datos: interior }
   TITexto = object(TView)
     constructor Init(var Limite: TRect);
     procedure Draw; virtual;
   end;
   PVTexto = ^TVTexto;   { Texto con los datos: ventana }
   TVTexto = object(TWindow)
     constructor Init(Limite: TRect; Titulo: String);
     procedure MakeInterior(Limite: TRect);
     { Podría tener su propio HandleEvent y demás, pero dejo que la }
     { maneje el programa principal }
   end;

 var
```

```
      prog: programa;                                 { El programa en sí
}
      FichAgenda: file of tipoagenda;                           { Fichero
}
      ficha: TipoAgenda;                        { Guarda la ficha actual
}
      NumFicha: word;                        { El número de ficha actual
}
      Ultima: word;                          { Número de la última ficha
}
      VTexto: PVTexto;                              { La ventana de texto
}
 constructor TITexto.Init(var Limite: TRect);
 begin
    TView.Init(Limite);
    GrowMode := gfGrowHiX + gfGrowHiY; { Para evitar problemas al
redimensionar }
                                       { aunque dejaré la ventana fija
}
    Options := Options or ofFramed;    { Con borde }
 end;
 procedure TITexto.Draw;
 var
    color1,color2: Byte;
    temp1, temp2:string;   { Para escribir el número de ficha y el
total }
    result: word;
    procedure EscribeTexto(x,y:longint; cadena:string; color:byte);
    var b: TDrawBuffer;
    begin
      MoveStr(b,cadena,color);
      WriteLine(x,y,length(cadena),1,b);
    end;
 begin
    {$I-}
    reset( FichAgenda );
    {$I+}
    if ioresult<>0 then
      begin    { Si no hay fichero, lo creo }
        rewrite(FichAgenda);
        ficha.nombre:='Nacho Cabanes';
        ficha.direccion:='Apartado 5234';
        ficha.ciudad:='Alicante';
        ficha.cp:='03080';
        ficha.observ:='Creador de esto...';
        write(FichAgenda,ficha);
        NumFicha := 1;
      end;
      seek(FichAgenda, NumFicha -1);
      read(FichAgenda,ficha);
      ultima:=filesize(FichAgenda);
    close(FichAgenda);
    { Habilito o deshabilito órdenes según donde me encuentre }
    If NumFicha = 1 then DisableCommands([cmAnterior])
      else EnableCommands([cmAnterior]);
    If NumFicha = ultima then DisableCommands([cmPosterior])
      else EnableCommands([cmPosterior]);
    TView.Draw;
    str(numFicha,temp1);
    str(ultima,temp2);
    with ficha do
```

```
  begin
    color1 := getcolor(1); color2 := getcolor(2);
    EscribeTexto(60,1,'Ficha '+temp1+' de '+temp2,color1);
    EscribeTexto(2,1,'Nombre:',color1);
      EscribeTexto(4,2,Nombre,color2);
    EscribeTexto(2,5,'Calle',color1);
      EscribeTexto(4,6,direccion,color2);
    EscribeTexto(2,8,'Ciudad',color1);
      EscribeTexto(4,9,ciudad,color2);
    EscribeTexto(36,8,'C.P.',color1);
      EscribeTexto(38,9,cp,color2);
    EscribeTexto(2,11,'Teléfono',color1);
     EscribeTexto(4,12,telef,color2);
    EscribeTexto(2,15,'Observaciones',color1);
      EscribeTexto(4,16,observ,color2);
  end;
 end;
 constructor TVTexto.Init(Limite: TRect; Titulo: String);
 begin
   TWindow.Init(Limite, Titulo, wnNoNumber);
   Flags:= flags and not wfClose and not wfGrow
     and not wfMove and not wfZoom;    { Ventana fija, no
redimensionable }
   MakeInterior(Limite);
 end;
 procedure TVTexto.MakeInterior(Limite: TRect);
 var
   Interior: PITexto;
 begin
   GetExtent(Limite);                          { Leemos el tamaño
disponible }
   Limite.Grow(-1,-1);                         { Sera un poco más
pequeño }
   Interior := New(PITexto, Init(Limite));     { Y lo creamos }
   Insert(Interior);
 end;
 procedure Programa.Saludo;                     { ----- Cartelito de
presentación }
 begin
   MessageBox(
     '       MiniAgenda Turbo Vision'#13#13+
     '            ¡Bienvenido!'
     , nil, mfOkButton+mfInformation);
 end;
 function LeeValor(rotulo, texto: string;    { Auxiliar para leer
datos }
   var valor: string; longitud: byte): word;
 var total:word;
   r:trect;
   Maximo: byte;
 begin
   Maximo:=longitud;                        { Hago la entrada más
flexible }
   if longitud>60 then longitud:=60;        { que en InputBox: el tamaño
}
   total:=longitud+length(texto)+10;        { de la línea de entrada
(Input }
   if total<30 then total:=30;              { Line) y el de la ventana
varían }
   r.assign(0,0,total,9);                   { en cada caso. }
   r.move(40-total div 2,7);
```

```
   LeeValor:=InputBoxRect(R,rotulo,texto,
       valor,maximo);
end;
procedure Programa.Modifica;
var
   rotulo: string;
   result: word;
begin
   rotulo:='Modificar';
   with ficha do begin
   result:=LeeValor(rotulo,'Nombre',nombre,longNOM);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Dirección',direccion,longDIR);
   if result=cmCancel then exit;
   result:=Leevalor(rotulo,'Ciudad',ciudad,longCIU);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Código postal',cp,longCP);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Teléfono',telef,longTLF);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Observaciones',observ,longOBS);
   if result=cmCancel then exit;
   end;
   reset(FichAgenda);
   seek( FichAgenda, NumFicha-1 );                 { Como siempre...   :-)
}
   write( FichAgenda, ficha );
   close(FichAgenda);
   VTexto^.redraw;
end;
procedure Programa.Nuevo;
var
   rotulo: string;
   result: word;
   ficha: TipoAgenda;
begin
   rotulo:='Añadir datos';
   fillchar(ficha, sizeof(Ficha), 0);
   with ficha do begin
   result:=LeeValor(rotulo,'Nombre',nombre,longNOM);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Dirección',direccion,longDIR);
   if result=cmCancel then exit;
   result:=Leevalor(rotulo,'Ciudad',ciudad,longCIU);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Código postal',cp,longCP);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Teléfono',telef,longTLF);
   if result=cmCancel then exit;
   result:=LeeValor(rotulo,'Observaciones',observ,longOBS);
   if result=cmCancel then exit;
   end;
   NumFicha := Ultima + 1;                    { Hay que escribir al final
}
   reset(FichAgenda);
   seek( FichAgenda, NumFicha-1 );                              { Se sitúa
}
   write( FichAgenda,ficha );                          { y escribe la ficha
}
   Ultima  := Ultima + 1;                                { Ahora hay una más
}
```

```
  VTexto^.redraw;
end;
procedure Programa.Anterior;
begin
  Dec(NumFicha);
  VTexto^.redraw;
end;
procedure Programa.Posterior;
begin
  Inc(NumFicha);
  VTexto^.redraw;
end;
procedure Programa.Numero;
var
  numeroStr: string[4];   { Numero como string }
  numeroW: word;          { Numero como word }
  result: word;           { Por si se cancela  }
  r: trect;               { Rectángulo para la ventana }
  cod: integer;           { Para "val" }
begin
  r.assign(0,0,68,9);
  r.move(4,12);
  NumeroStr:='';
  result:=InputBoxRect(R,'Agenda: Número de Ficha',
       '¿ Cual es el número de la ficha a la que quiere saltar ?',
       NumeroStr,4);
  if result=cmCancel then exit;
  val (numeroStr, numeroW, cod);
  if not (numeroW in [1..ultima]) then exit else NumFicha:=numeroW;
  VTexto^.redraw;
end;
procedure Programa.CreaInterior;
var
  R: TRect;
begin
  GetExtent(r);
  R.B.Y:=R.B.Y-2;
  VTexto := New(PVTexto, Init(R, 'Agenda Turbo Vision'));
  DeskTop^.Insert(VTexto);
end;
procedure Programa.Busca;
var
  posAnterior: word;    { Por si se cancela, volver a la anterior }
  result: word;         { Para comprobar si se cancela }
  texto: string;        { Texto a buscar }
  i: word;              { Bucles }
begin
  posAnterior := numFicha;
  texto := '';      { Posible mejora: conservar el texto buscado }
  result := InputBox('Agenda: Buscar',
       '¿ Qué texto quiere buscar ?',
       Texto,15);
  if result = cmCancel then exit;
  reset(fichAgenda);
  for i := numFicha to ultima do   { Busca desde la actual }
    begin
    seek(FichAgenda, i-1);
    read(FichAgenda, ficha);
    with ficha do                        { Miro en todos los campos
}
      if (pos(texto, nombre) > 0) or (pos(texto, direccion) > 0) or
```

```
            (pos(texto, ciudad) > 0) or (pos(texto, cp) > 0) or
            (pos(texto, telef) > 0) or (pos(texto, observ) > 0)
        then
          begin
            MessageBox(
            #3'Encontrado'
            , nil, mfOkButton+mfInformation);
            numFicha := i;
            vTexto^.redraw;
            {close(fichAgenda);}
            { No hace falta cerrar el fichero: lo hago al redibujar }
            exit;
          end;
      end;
      MessageBox(
        #3'No encontrado'
        , nil, mfOkButton+mfInformation);
      close(fichAgenda);
      numFicha := posAnterior;
  end;
  procedure Programa.InitMenuBar;
  var R: TRect;
  begin
    GetExtent(R);
    R.B.Y := R.A.Y + 1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
      NewSubMenu('AGENDA', hcNoContext, NewMenu(
        NewItem('~A~cerca de...', '', kbNoKey, cmSaludo, hcNoContext,
        NewLine(
        NewItem('~T~erminar', 'Alt-T', kbAltT, cmQuit, hcNoContext,
        nil)))),
      NewSubMenu('~O~rdenes', hcNoContext, NewMenu(
        NewItem('Anterior', 'Re.Pag.', kbPgUp, cmAnterior, hcNoContext,
        NewItem('Posterior', 'Av.Pag.', kbPgDn, cmPosterior,
hcNoContext,
        NewItem('Número de ~f~icha', 'Alt-F', kbAltF, cmNumero,
hcNoContext,
        NewItem('Añadir ficha ~n~ueva', 'Alt-A', kbAltA, cmNuevo,
hcNoContext,
        NewItem('~M~odificar', 'Alt-M', kbAltM, cmModifica,
hcNoContext,
        NewItem('~B~uscar texto', 'Alt-B', kbAltB, cmBusca,
hcNoContext,
        nil))))))),
      nil)))));
  end;
  procedure Programa.InitStatusLine;
  var R: TRect;
  begin
    GetExtent(R);
    R.A.Y := R.B.Y - 1;
    StatusLine := New(PStatusLine, Init(R,
      NewStatusDef(0, $FFFF,
        NewStatusKey('', kbF10, cmMenu,
        NewStatusKey('~T~erminar', kbAltT, cmQuit,
        NewStatusKey('', kbAltX, cmQuit,     { Salir con Alt+T ó Alt+X }
        NewStatusKey('~Re.Pag.~', kbPgUp, cmAnterior,
        NewStatusKey('~Av.Pag.~', kbPgDn, cmPosterior,
        NewStatusKey('~F~icha', kbAltF, cmNumero,
        NewStatusKey('~N~uevo', kbAltN, cmNuevo,
        NewStatusKey('~M~odificar', kbAltM, cmModifica,
```

```
      NewStatusKey('~B~uscar', kbAltB, cmBusca,
      NewStatusKey('', kbEsc, cmClose,
      nil)))))))))),
    nil)));
end;

procedure Programa.HandleEvent(var Event: TEvent);
begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmSaludo:   Saludo;
      cmAnterior: Anterior;
      cmPosterior:Posterior;
      cmNumero:   Numero;
      cmModifica: Modifica;
      cmNuevo:    Nuevo;
      cmBusca:    Busca;
    else
      Exit;
    end;
  ClearEvent(Event);
  end;
end;
begin                            { ----- Cuerpo del programa ----- }
  assign( FichAgenda, nombref );
  NumFicha := 1;
  Prog.Init;
  Prog.Saludo;
  Prog.CreaInterior;
  Prog.Run;
  Prog.Done;
  writeln('Se acabó...');
end.
```

Curso de Pascal. Ampliación 1: Otras órdenes no vistas.

A lo largo del curso ha habido órdenes que no hemos tratado, bien porque no encajasen claramente en ningún tema, o bien porque eran demasiado avanzadas para explicarlas junto con las más parecidas.

En primer lugar vamos a ir viendo las que podían haber formado parte de temas anteriores, y después las que faltan. Tampoco pretendo que esto sea una recopilación exhaustiva, sino simplemente mencionar algunas órdenes interesantes que parecían haberse quedado en el tintero.

Los temas que se van a comentar son:

- Bucles: break, continue.
- Saltos: goto, label.
- Punteros: getmem, freemem, pointer.
- Fin del programa: exit, halt.
- Números aleatorios: rnd, randomize.
- Inc y Dec.
- Acceso a la impresora.

Bucles: break, continue.

En Turbo Pascal 7.0, tenemos dos órdenes extra para el control de bucles, tanto si se trata de "for", como de "repeat" o "until". Estas órdenes son:

- Break: sale del bucle inmediatamente.
- Continue: pasa a la siguiente iteración.

Un ejemplo "poco útil" que use ambas podría ser escribir los números pares hasta el 10 con un for:

Warning: file_get_contents(sources/ for i := 1 to 1000 do { Nos podemos pasar } begin if i>10 then break; { Si es así, sale } if i mod 2 = 1 then continue; { Si es impar, no lo escribe } writeln(i); { Si no, lo escribe } end;) [function.file-get-contents]: failed to open stream: No such file or directory in **/home/siquees/freepascal.es/tutorials/todojunto.php** on line **14**

Esto se puede imitar en versiones anteriores de TP con el temible "goto".

Goto.

Es una orden que se puede emplear para realizar saltos incondicionales. Su empleo está bastante **desaconsejado**, pero en ciertas ocasiones, y se se lleva cuidado, puede ser útil (para salir de bucles fuertemente anidados, por ejemplo, especialmente si no se dispone de las dos órdenes anteriores.

El formato es

```
goto etiqueta
```

y las etiquetas se deben declarar al principio del programa, igual que las variables. Para ello se usa la palabra "**label**". Vamos a verlo directamente con un ejemplo:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de saltos y   }
{    etiquetas             }
{    LABEL1.PAS            }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program label1;

label
  uno, dos;

var
  donde: byte;

begin
  writeln('¿Quiere saltar a la opción 1 o a la 2?');
  readln (donde);
  case donde of
     1: goto uno;
     2: goto dos;
     else writeln('Número incorrecto');
     end;
  exit;
uno:
  writeln('Esta es la opción 1.  Vamos a seguir...');
dos:
  writeln('Esta es la opcion 2.');
end.
```

Punteros: getmem, freemem, pointer.

Habíamos hablado de "new" y "dispose", en las que el compilador decide la cantidad de memoria que debe reservar. Pero también podemos obligarle a reservar la que nosotros queramos, lo que nos puede interesar, por ejemplo, para leer cadenas de caracteres de longitud variable.

Para esto usamos "getmem" y "freemem", en los que debemos indicar cuánta memoria queremos reservar para el dato en cuestión. Si queremos utilizarlos como new y dispose, reservando toda la memoria que ocupa el dato (será lo habitual), podemos usar "sizeof" para que el compilador lo calcule por nosotros:

```
type
  TipoDato = record
     Nombre: string[40];
     Edad: Byte;
     end;

var
   p: pointer;       { Puntero genérico }

begin
   if MaxAvail < SizeOf(TipoDato) then
      Writeln('No hay memoria suficiente')
   else
      begin
      GetMem(p, SizeOf(TipoDato));
      { Trabajaríamos con el dato, y después... }
      FreeMem(p, SizeOf(TipoDato));
      end;
end.
```

Por cierto, "**pointer**" es un puntero genérico, que no está asociado a ningún tipo concreto de dato. No lo vimos en la lección sobre punteros, porque para nosotros lo habitual es trabajar con punteros que están relacionados con un cierto tipo de datos.

Un ejemplo (avanzado, eso sí) que emplea la palabra pointer se puede encontrar en la ampliación 5, y muestra cómo dibujar figuras que se desplacen por la pantalla sin borrar el fondo (sprites).

Fin del programa: exit, halt.

En el tema 8 (y en los ejemplos del tema 6) vimos que podíamos usar exit para salir de un programa.

Esto no es exacto del todo: **exit** sale del **bloque** en el que nos encontremos, que puede ser un procedimiento o una función. Por tanto, sólo sale del programa si ejecutamos exit desde el cuerpo del programa principal.

Si estamos dentro de un procedimiento, y queremos **abandonar** el programa por completo, deberíamos hacer más de un "exit". Pero también hay otra opción: la orden "**halt**".

Halt sí que abandona el programa por completo, estemos donde estemos. Como parámetro se le puede pasar el "Errorlevel" que se quiere devolver al DOS, y que se puede leer desde un fichero batch.

Por ejemplo: "halt" o "halt(0)" indicaría una salida normal del programa (sin errores), "halt(1)" podría indicar que no se han encontrado los datos necesarios, etc.

Números aleatorios: random, randomize.

Si queremos utilizar números aleatorios (generados al azar) en nuestros programas, podemos emplear la función "**random**". Si la usamos tal cual, nos devuelve un número real entre 0 y 1. Si usamos el formato "random(n)", lo que devuelve es un número entero entre 0 y n-1.

Para que el ordenador comience a generar la secuencia de números aleatorios, podemos usar "**randomize**", que toma como semilla un valor basado en el reloj, lo que supone que sea suficientemente aleatorio:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Ejemplo de números    }
{    aleatorios (al azar)  }
{    RANDOM1.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Free Pascal 2.2.0w  }
{    - Turbo Pascal 7.0    }
{--------------------------}

program random1;

var i:integer;

begin
  Randomize;
  for i := 1 to 50 do
     Write (Random(1000), ' ');
end.
```

Inc y Dec.

Son dos procedimientos que nos permiten aumentar o disminuir el valor de una variable. Así Inc(i) es "casi" lo mismo que i:=i+1 y Dec(i) es "casi" lo mismo que i:=i-1.

Lo de "casi" es porque hay un par de diferencias:

- La ventaja es que con Inc, el compilador genera un código algo más eficiente, con lo que ganamos en velocidad.
- Los dos inconvenientes son:
    - No es estándar, así que es muy probable que no funcione en otro compilador.
    - Sólo se puede usar con tipos de datos ordinales (números enteros, char, etc., pero no con reales, por ejemplo).

Acceso a la impresora.

Acceder a la impresora desde Turbo Pascal es muy fácil:

```
writeln(lst, 'Hola');
```

Nada más y nada menos. Con eso escribiremos "Hola" en la impresora (que está asociada al dispositivo "lst"). Esto está definido en la unidad printer, de modo que deberá nuestra programa deberá comenzar con

```
uses printer;
```

Así, un programa ya más completito quedaría:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Prueba de la impre-   }
{    sora                  }
{    IMPRESOR.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
```

```
{  Comprobado con:          }
{     - Turbo Pascal 7.0    }
{--------------------------}
program impresor;

uses printer;

begin
  writeln(lst,'Hola');
end.
```

Curso de Pascal. Ampliación 2: Gráficos sin BGI.

Habíamos comentado cómo crear gráficos mediante los **drivers BGI** que distribuye Borland. Estos tienen como principal ventaja la gran cantidad de funciones y procedimientos que ponen a nuestra disposición.

Pero también tienen **inconvenientes**, principalmente dos:

- Es incómodo tener que depender siempre de los ficheros BGI, bien por tenerlos junto a nuestros ejecutables, o bien por tener que enlazarlos con ellos.
- No son muy rápidos.

Podemos saltarnos estos dos inconvenientes, accediendo **nosotros** mismos a la pantalla gráfica. Lo haremos de **dos formas**: a través de la BIOS o manejando directamente la memoria de video (a cambio, todo este apartado será exclusivo de Turbo Pascal 7 para MsDOS).

1. A través de la Bios.

Una guía de referencia muy buena para todos aquellos que quieren programar el PC a un nivel algo más bajo del habitual (más cercano a la máquina) es la lista de interrupciones recopilada por Ralf Brown. En ella encontramos lo siguiente:

```
 Interrupt List    Release 41        Last change 6/5/94
 This compilation is Copyright (c) 1989,1990,1991,1992,1993,1994 Ralf
Brown
 [...]
 INT 10 - VIDEO - SET VIDEO MODE
```

```
AH = 00h
AL = mode (see #0009)
```

y la lista de **modos** (muy resumida, dejando sólo las más habituales de las más de 600 líneas que aparecen en esta recopilación):

```
Values for video mode:
     text/ text   pixel   pixel colors    disply scrn  system
     grph resol    box  resolution        pages  addr
 00h = T  40x25    8x8   320x200 16gray     8   B800 CGA
     = T  40x25    8x14  320x350 16gray     8   B800 EGA
     = T  40x25    8x16  320x400  16        8   B800 MCGA
     = T  40x25    9x16  360x400  16        8   B800 VGA
 01h = T  40x25    8x8   320x200  16        8   B800 CGA
     = T  40x25    8x14  320x350  16        8   B800 EGA
     = T  40x25    8x16  320x400  16        8   B800 MCGA
     = T  40x25    9x16  360x400  16        8   B800 VGA
 02h = T  80x25    8x8   640x200 16gray     4   B800 CGA
     = T  80x25    8x14  640x350 16gray     8   B800 EGA
     = T  80x25    8x16  640x400  16        8   B800 MCGA
     = T  80x25    9x16  720x400  16        8   B800 VGA
 03h = T  80x25    8x8   640x200  16        4   B800 CGAy
     = T  80x25    8x14  640x350  16/64     8   B800 EGA
     = T  80x25    8x16  640x400  16        8   B800 MCGA
     = T  80x25    9x16  720x400  16        8   B800 VGA
 04h = G  40x25    8x8   320x200   4      .   B800 CGA,EGA,MCGA,VGA
 05h = G  40x25    8x8   320x200  4gray   .   B800 CGA,EGA
     = G  40x25    8x8   320x200   4      .   B800 MCGA,VGA
 06h = G  80x25    8x8   640x200   2      .   B800 CGA,EGA,MCGA,VGA
 07h = T  80x25    9x14  720x350 mono   var  B000 MDA,Hercules,EGA
     = T  80x25    9x16  720x400 mono     .   B000 VGA
 0Dh = G  40x25    8x8   320x200  16      8   A000 EGA,VGA
 0Eh = G  80x25    8x8   640x200  16      4   A000 EGA,VGA
 0Fh = G  80x25    8x14  640x350 mono     2   A000 EGA,VGA
 10h = G  80x25    8x14  640x350   4      2   A000 64k EGA
     = G    .        .   640x350  16      .   A000 256k EGA,VGA
 11h = G  80x30    8x16  640x480 mono     .   A000 VGA,MCGA
 12h = G  80x30    8x16  640x480  16/256K.   A000 VGA
 13h = G  40x25    8x8   320x200 256/256K.   A000 VGA,MCGA
```

Es decir: para cambiar al modo **640x480 con 16 colores**, tendremos que hacer:

```
AH = 00   (Función de elegir el modo de pantalla)
AL = 12h  (Modo 640x480x16, VGA)
INT 10h   (Interrupción de video)
```

Antes de ver meternos con el Pascal, vamos a ver cómo **dibujar un punto**:

```
INT 10 - VIDEO - WRITE GRAPHICS PIXEL
   AH = 0Ch
   BH = page number
   AL = pixel color (if bit 7 set, value is xor'ed onto screen)
   CX = column
   DX = row
Desc: set a single pixel on the display in graphics modes
Notes:   valid only in graphics modes
   BH is ignored if the current video mode supports only one page
```

y también podemos **leer el color de un pixel** con

```
INT 10 - VIDEO - READ GRAPHICS PIXEL
   AH = 0Dh
   BH = page number
   CX = column
   DX = row
Return: AL = pixel color
Desc: determine the current color of the specified pixel in grahics
modes
Notes:   valid only in graphics modes
   BH is ignored if the current video mode supports only one page
```

Bueno, ya vale de parrafadas en inglés y vamos a empezar a aplicarlo. Hagamos un programa que dibuje puntos en la pantalla en modo **640x480x16**:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Gráficos mediante los }
{    servicios de la BIOS: }
{    640x480, 16 colores   }
{    GRB1.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.00  }
{--------------------------}

program GrB1;  { Gráficos a través de la BIOS - 1 }

uses dos;                     { Usaremos interrupciones }

const NumPuntos = 10000;       { Número de puntos que dibujaremos }
```

```
var
  regs: registers;              { Para acceder a los registros, claro }
  bucle: word;                  { Para bucles, claro }

procedure Modo640x480x16;       { Cambia a modo gráfico }
begin
  regs.ah := 0;                 { Función 0 }
  regs.al := $12;               { El modo es el 12h = 18 }
  intr($10,regs);               { Interrupción de video }
end;

procedure ModoTexto;            { A modo texto, similar }
begin
  regs.ah := 0;
  regs.al := 3;
  intr($10,regs);
end;

procedure PonPixel(x,y: word; color: byte);       { Dibuja Pixel }
begin
  regs.ah := $0c;               { Función 0Ch }
  regs.bh := 0;                 { Página 0 }
  regs.al := color;             { Color }
  regs.cx := x;                 { Columna }
  regs.dx := y;                 { Fila }
  intr($10,regs);
end;

begin
  Modo640x480x16;
  for bucle := 1 to NumPuntos do
    PonPixel( random(639), random(479), random(15) );   { Puntos
                                                        aleatorios}
  readln;
  ModoTexto;
end.
```

Hemos conseguido un programa que dibuja puntos sin necesidad de los BGI. El EXE resultante ocupa escasamente 2.5K (compilado con TP7).

Si lo compilamos con **TMT** Pascal Lite, el tamaño sube hasta los 18K, pero es porque este compilador incluye mayor información adicional, para que el fichero .EXE trabaje en modo protegido del 386 (aprovechando toda la memoria de nuestro ordenador, etc). Por cierto, la línea del "uses", si se compila con TMT, deberá ser:

```
uses dos, use32;
```

Pero esto que hemos hecho también es mejorable: llamar a esa interrupción cada vez que queramos dibujar un punto resulta **lento**, especialmente cuando queramos dibujar líneas o rellenar zonas. La forma más rápida, al menos en ciertos modos gráficos, o si se programa con cuidado, es acceder directamente a la memoria de video.

A través de la memoria de pantalla.

Aquí nos encontramos un primer "pequeño" **problema**: la memoria de pantalla puede empezar en distintas posiciones de la memoria física total del ordenador, según la tarjeta gráfica (o el modo gráfico) que estemos usando. Por ello me centraré en la VGA.

El segundo "pequeño" problema es que incluso para la VGA, el acceso a la memoria **no siempre** es igual: tenemos modos como el 13h (320x200, 256 colores) en lo que cada byte corresponde a un pixel. Otros modos en los que un byte corresponde a 8 pixels (640x480 mono: 11h). En otros, tenemos 4 planos: un byte en cada uno afectará al color de ocho pixels...

Hay montones de libros que os pueden ayudar a programar en esos modos "complicados", en los que hay que fijar sólo algunos de los bits de cada byte para dibujar un pixel, o hacer cambios de plano.

Mi intención no es esa, sino daros la base para que cada uno pueda experimentar por donde quiera, así que me voy a centrar en el modo más fácil de programar, y que además es el más vistoso de la VGA estándar: el modo 13h: **320x200, 256 colores**.

En ese modo tenemos que cada punto de la pantalla corresponde a un byte y viceversa. Así, si el primer byte de la memoria de pantalla es un 3, quiere decir que en las coordenadas (0,0) (esquina superior izquierda) hay un punto de color 3.

Esta memoria de pantalla, en un PC con tarjeta gráfica VGA funcionado con sistema operativo MsDOS, está en el **segmento A000h**, luego la dirección de dicho primer punto será A000:0000. La cantidad de memoria que ocuparemos será 320x200 = 64.000 bytes.

Entonces, cambiaremos al **modo gráfico** como antes:

```
procedure Modo320x200x256;     { Cambia a modo gráfico }
begin
  regs.ah := 0;                { Función 0 }
  regs.al := $13;              { El modo es el 13h = 19 }
  intr($10,regs);              { Interrupción de video }
end;
```

y para **dibujar un punto** podemos acceder directamente a la posición de memoria que nos interesa:

```
procedure PonPixel(x,y: word; color: byte);      { Dibuja Pixel }
begin
```

```
  Mem[$A000 : y * 320 + x] := color;
end;
```

Con lo que la adaptación del ejemplo anterior es inmediata.

También podemos **leer** fácilmente el color de un cierto punto de la pantalla:

```
function LeePixel(x, y : Word) : Byte;        { Color de un pixel }
begin
  LeePixel := Mem[$A000 : y * 320 + x];
end;
```

Otra forma muy sencilla de dibujar puntos en pantalla y leerlos es creando un **array** en la **posición concreta** de la memoria que ya conocemos:

```
  var
    pantalla: array [ 0..319, 0..199 ]  of byte absolute $A000:0000;
```

Dibujaríamos un punto con

```
   pantalla [x,y] := color;
```

y lo leeríamos con

```
   color := pantalla [x,y];
```

Así, si usamos Turbo Pascal 7, el ejemplo anterior podría quedar simplemente

```
 {--------------------------}
 {  Ejemplo en Pascal:      }
 {                          }
 {    Gráficos accediendo a }
 {    memoria de pantalla:  }
 {    320x200, 256 colores  }
 {    Versión para TP7      }
 {    GRB2.PAS              }
 {                          }
 {  Este fuente procede de  }
 {  CUPAS, curso de Pascal  }
 {  por Nacho Cabanes       }
 {                          }
 {  Comprobado con:         }
 {    - Turbo Pascal 7.0    }
 {--------------------------}
```

```
program GrB2;

uses dos;                     { Usaremos interrupciones }

const NumPuntos = 10000;       { Número de puntos que dibujaremos }

var
  regs: registers;            { Para acceder a los registros, claro }
  bucle: word;                { Para bucles, claro }
  pantalla: array [0..319,0..199]  of byte
    absolute $A000:0;         { Pues la pantalla ;-) }

procedure ModoPantalla( modo: byte );
                              { Cambia a un modo dado }
begin
  regs.ah := 0;               { Función 0 }
  regs.al := modo;            { El modo indicado }
  intr($10,regs);             { Interrupción de video }
end;

begin
  ModoPantalla($13);
  for bucle := 1 to NumPuntos do
    Pantalla[ random(319), random(199)] := random(255) ;
  readln;
  ModoPantalla(3);
end.
```

Si compilamos con **TMT**, debemos indicarle la dirección de memoria de la pantalla de una forma ligeramente distinta:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Gráficos accediendo a }
{    memoria de pantalla:  }
{    320x200, 256 colores  }
{    Versión para TMT      }
{    GRB2T.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Tmt Pascal Lt 1.00  }
{--------------------------}

program GrB2T;

uses dos;                     { Usaremos interrupciones }

const NumPuntos = 10000;       { Número de puntos que dibujaremos }

var
  regs: registers;            { Para acceder a los registros, claro }
  bucle: word;                { Para bucles, claro }
  pantalla: array [0..319,0..199]  of byte
```

```
    absolute $A0000;          { Pues la pantalla ;-) }

procedure ModoPantalla( modo: byte );
                              { Cambia a un modo dado }
begin
  regs.ah := 0;               { Función 0 }
  regs.al := modo;            { El modo indicado }
  intr($10,regs);             { Interrupción de video }
end;

begin
  ModoPantalla($13);
  for bucle := 1 to NumPuntos do
    Pantalla[ random(319), random(199)] := random(255) ;
  readln;
  ModoPantalla(3);
end.
```

Si empleamos **Free Pascal**, en general no podremos usar este método, porque existe para distintos tipos de ordenadores, con distintos sistemas operativos y diferentes subsistemas gráficos, y nada nos garantiza que la memoria de video se encuentre en esa posición de memoria.

Líneas y otras figuras simples

Vimos cómo cambiar a un cierto modo gráfico y como dibujar un punto de dos formas: a través de la BIOS como método general, y accediendo a la memoria de video en el caso particular del modo 320x200 en 256 colores de las tarjetas VGA y MCGA.

Hoy vamos a ver cómo **dibujar líneas** y otras figuras sencillas.

Supongo que casi todos los que nos paseamos por aquí tenemos el suficiente nivel en matemáticas como para pensar una forma de dibujar una recta: aplicando la **ecuación** de la recta "tal cual". La recta que pasa por dos puntos (x1,y1) (x2,y2) tiene como ecuación:

```
  x-x1      y-y1
------- = -------
 x2-x1     y2-y1
```

que podríamos recolocar para escribir una variable en función de la otra. Por ejemplo, y en función de x:

```
          y2-y1
y = y1 + ------- (x-x1)
```

```
        x2-x1
```

Esto funciona, claro. Pero se puede optimizar mucho. ¿Por qué? Porque para cada calcular la coordenada y de cada punto estamos haciendo 3 restas, una suma, una multiplicación y una división. La suma y la resta son "menos malas" ;-) pero la multiplicación y aun más la división son operaciones lentas.

La primera mejora "casi evidente" viene si nos damos cuenta de que realmente no hay tal división:

```
    y2-y1          Conocemos y2, y1, x2, x1 => Esta división siempre
   -------         vale lo mismo => es una constante.
    x2-x1
```

Lo hemos convertido en algo parecido a:

```
 y = y1 + c (x-x1)
```

Pero aun así nos queda la multiplicación, y esta no se ve tan claro la forma de eliminarla...

¡ Pero para eso están los inteligentes matemáticos, que se estrujan el coco por nosotros ! :-)

Así es: existen **algoritmos incrementales**, que emplean sumas en vez de multiplicaciones. Posiblemente el más usado sea el de Bresenham, que es el que voy a poner a continuación:

*(Esta rutina no es mía, es de **dominio público** y está tomada de los **SWAG**, unas recopilaciones muy interesantes de fuentes de Pascal; en concreto, esta colaboración es de Sean Palmer, que yo apenas he comentado y poco más).*

```
{Línea, por algoritmo de Bresenham}

procedure linea(x, y, x2, y2 : integer);
var
  d,
  dx, dy,              { Salto total según x e y }
  ai, bi,
  xi, yi               { Incrementos: +1 ó -1, según se recorra }
          : integer;
begin
  if (x < x2) then     { Si las componentes X están ordenadas }
  begin
```

```
    xi := 1;            { Incremento +1 }
    dx := x2 - x;       { Espacio total en x }
  end
  else                  { Si no están ordenadas }
  begin
    xi := - 1;          { Increm. -1 (hacia atrás) }
    dx := x - x2;       { y salto al revés (negativo) }
  end;
  if (y < y2) then      { Análogo para las componentes Y }
  begin
    yi := 1;
    dy := y2 - y;
  end
  else
  begin
    yi := - 1;
    dy := y - y2;
  end;
  plot(x, y);           { Dibujamos el primer punto }
  if dx > dy then       { Si hay más salto según x que según y }
  begin                 { (recta más cerca de la horizontal) }
    ai := (dy - dx) * 2;    { Variables auxiliares del algoritmo }
    bi := dy * 2;           { ai y bi no varían; d comprueba cuando }
    d  := bi - dx;          { debe cambiar la coordenada y }
    repeat
      if (d >= 0) then      { Comprueba si hay que avanzar según y }
      begin
        y := y + yi;        { Incrementamos Y como deba ser (+1 ó -1) }
        d := d + ai;        { y la variable de control }
      end
      else
        d := d + bi;        { Si no varía y, d sí lo hace según bi }
      x := x + xi;          { Incrementamos X como corresponda }
      plot(x, y);           { Dibujamos el punto }
    until (x = x2);     { Se repite hasta alcanzar el final }
  end
  else                  { Si hay más salto según y que según x }
  begin                 { (más vertical), todo similar }
    ai := (dx - dy) * 2;
    bi := dx * 2;
    d  := bi - dy;
    repeat
      if (d >= 0) then
      begin
        x := x + xi;
        d := d + ai;
      end
      else
        d := d + bi;
      y := y + yi;
      plot(x, y);
    until (y = y2);
  end;
end;
```

En este algoritmo, que está expresado de forma **genérica**, basta sustituir el "Plot(x,y)" por nuestra propia rutina de dibujo de puntos, como el PonPixel(x,y,color).

Por si alguien se ha dado cuenta de que en este algoritmo hay una multiplicación de todas formas (por 2, en la definición de ai y bi), y que puede realmente no haber ahorro, esto no es cierto del todo...

¿Por qué? Pues porque las mutiplicaciones por múltiplos de dos se pueden codificar como "desplazamientos" o "rotaciones" de bits, en este caso SHL 1. De hecho, esto lo hace automáticamente el compilador de TP7 en muchos casos.

Y aun así, se trata simplemente de que se vea el algoritmo. Porque a nadie se le escapa que x*2 es lo mismo que x+x, y esta última operación puede ser más rápida, pero también menos legible.

Por cierto, las órdenes como x := x + xi se pueden escribir también mediante la orden "**incrementar**": inc(x,xi), lo que además ayuda al compilador a generar un código más eficiente.

Para curiosos que quieran experimentar un poco, el siguiente algoritmo (también una contribución de Sean Palmer) dibuja una **elipse rellena** (o un círculo, claro):

```
{filled ellipse}
procedure disk(xc, yc, a, b : integer);
var
  x, y       : integer;
  aa, aa2,
  bb, bb2,
  d, dx, dy : longint;
begin
  x   := 0;
  y   := b;
  aa  := longint(a) * a;
  aa2 := 2 * aa;
  bb  := longint(b) * b;
  bb2 := 2 * bb;
  d   := bb - aa * b + aa div 4;
  dx  := 0;
  dy  := aa2 * b;
  vLin(xc, yc - y, yc + y);
  while (dx < dy) do
  begin
    if (d > 0) then
    begin
      dec(y);
      dec(dy, aa2);
      dec(d, dy);
    end;
    inc(x);
```

```
      inc(dx, bb2);
      inc(d, bb + dx);
      vLin(xc - x, yc - y, yc + y);
      vLin(xc + x, yc - y, yc + y);
    end;
    inc(d, (3 * (aa - bb) div 2 - (dx + dy)) div 2);
    while (y >= 0) do
    begin
      if (d < 0) then
      begin
        inc(x);
        inc(dx, bb2);
        inc(d, bb + dx);
        vLin(xc - x, yc - y, yc + y);
        vLin(xc + x, yc - y, yc + y);
      end;
      dec(y);
      dec(dy, aa2);
      inc(d, aa - dy);
    end;
  end;
```

Comentarios: Este procedimiento está transcrito tal y como aparecía en los SWAG. Que cada uno lo estudie por su cuenta si quiere y se atreve. Sólo un par de pistas: INC incrementa una variable y DEC la decrementa. VLIN (x,y1,y2) es un procedimiento que nosotros no hemos definido -deberes, JeJe- y que dibuja una línea vertical entre los puntos (x,y1) y (x,y2).

¿Y por qué se usa VLIN en vez del procedimiento anterior para dibujar líneas? Pues por **rapidez**: normalmente lo más rápido es dibujar una línea horizontal, ya que todos los puntos se encuentran seguidos en la memoria de pantalla. El siguiente caso es el de una línea vertical: cada punto está 320 bytes después del anterior en la memoria de pantalla. En el caso general, estos incrementos varían, y hay que usar algoritmos más genéricos y más difíciles de optimizar.

Algunas **optimizaciones.**

No quiero meterme de lleno en rotaciones y similares. Al fin y al cabo, esto no es un curso de programación gráfica, sino un tema más de un curso de Pascal que tampoco pretende ser tan profundo como pueda serlo un libro. Mi intención es más abrir las puertas, para que quien luego quiera adentrarse más lo tenga medianamente fácil.

Pero considero que hay aspectos importantes en la programación y que a veces no se tienen en cuenta.

Vamos a empezar por hacer un programita que haga **rotar una línea**, como si fueran una aguja de un reloj. Para ello aprovecharemos parte de lo que vimos en el apartado anterior y parte de éste, ya aplicado...

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Líneas que rotan, en  }
{    memoria de pantalla:  }
{    Versión para TP7      }
{    GRB3.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program GrB3;

uses dos, crt;                  { Usaremos interrupciones,
                                  keypressed y delay }

const NumPuntos = 10000;        { Número de puntos que dibujaremos }

var
  regs: registers;             { Para acceder a los registros, claro }
  bucle: real;                 { Para bucles, claro }
  tecla: char;                 { La tecla que se pulse }

procedure ModoPantalla( modo: byte );
                               { Cambia a un modo dado }
begin
  regs.ah := 0;                { Función 0 }
  regs.al := modo;             { El modo indicado }
  intr($10,regs);              { Interrupción de video }
end;

procedure PonPixel(x,y: word; color: byte);      { Dibuja Pixel }
begin
  Mem[$A000 : y * 320 + x] := color;
end;

procedure Linea(x, y, x2, y2 : word; color: byte);
var
  d,
  dx, dy,              { Salto total según x e y }
  ai, bi,
  xi, yi               { Incrementos: +1 ó -1, según se recorra }
          : integer;
begin
  if (x < x2) then     { Si las componentes X están ordenadas }
  begin
    xi := 1;           { Incremento +1 }
    dx := x2 - x;      { Espacio total en x }
  end
  else                 { Si no están ordenadas }
  begin
```

```
    xi := - 1;          { Increm. -1 (hacia atrás) }
    dx := x - x2;       { y salto al revés (negativo) }
  end;
  if (y < y2) then      { Análogo para las componentes Y }
  begin
    yi := 1;
    dy := y2 - y;
  end
  else
  begin
    yi := - 1;
    dy := y - y2;
  end;
  PonPixel(x, y,color);   { Dibujamos el primer punto }
  if dx > dy then       { Si hay más salto según x que según y }
  begin                 { (recta más cerca de la horizontal) }
    ai := (dy - dx) * 2;    { Variables auxiliares del algoritmo }
    bi := dy * 2;           { ai y bi no varían; d comprueba cuando }
    d  := bi - dx;          { debe cambiar la coordenada y }
    repeat
      if (d >= 0) then      { Comprueba si hay que avanzar según y }
      begin
        y := y + yi;        { Incrementamos Y (+1 ó -1) }
        d := d + ai;        { y la variable de control }
      end
      else
        d := d + bi;        { Si no varía y, d sí lo hace según bi }
      x := x + xi;          { Incrementamos X como corresponda }
      PonPixel(x, y, color);          { Dibujamos el punto }
    until (x = x2);     { Se repite hasta alcanzar el final }
  end
  else                  { Si hay más salto según y que según x }
  begin                 { (más vertical), todo similar }
    ai := (dx - dy) * 2;
    bi := dx * 2;
    d  := bi - dy;
    repeat
      if (d >= 0) then
      begin
        x := x + xi;
        d := d + ai;
      end
      else
        d := d + bi;
      y := y + yi;
      PonPixel(x, y, color);
    until (y = y2);
  end;
end;

begin
  ModoPantalla($13);      { Modo 320x200x256 }
  bucle := 0;             { Empezamos en 0 __RADIANES__ }
  repeat
    linea(160,100,        { Línea desde el centro de la pantalla }
      160 + round(60*cos(bucle)),  { Extremo en un círculo }
      100 + round(40*sin(bucle)),
      0);                          { Color negro (borrar) }
    bucle := bucle + 0.1;          { Siguiente posición }

    linea(160,100,         { Otra línea, pero ahora blanca }
```

```
       160 + round(60*cos(bucle)), 100 + round(40*sin(bucle)),
       15);
     delay(25);              { Esperamos 25 milisegundos }
   until keyPressed;         { Seguimos hasta que se pulse una tecla }
   tecla := ReadKey;         { Quitamos esa tecla del buffer del teclado
}
   ModoPantalla(3);          { Y volvemos a modo texto }
 end.
```

Esa combinación de *radio*cos(angulo) y radio*sin(angulo)* es la que nos da las coordenadas de cada punto de una circunferencia de cierto radio, es la que se suele usar para calcular rotaciones en el plano con un cierto radio. No necesitamos gran velocidad, y de hecho hemos puesto un retardo de 25 milisegundos entre línea y línea.

La versión para **TMT** de este programita sería:

```
 {--------------------------}
 {  Ejemplo en Pascal:      }
 {                          }
 {    Líneas que rotan, en  }
 {    memoria de pantalla:  }
 {    Versión para TMT      }
 {    GRB3T.PAS             }
 {                          }
 {  Este fuente procede de  }
 {  CUPAS, curso de Pascal  }
 {  por Nacho Cabanes       }
 {                          }
 {  Comprobado con:         }
 {    - Tmt Pascal Lt 1.00  }
 {--------------------------}

 program GrB3;

 uses dos, crt, use32;          { Usaremos interrupciones,
                                  keypressed y delay }

 const NumPuntos = 10000;       { Número de puntos que dibujaremos }

 var
   regs: registers;            { Para acceder a los registros, claro }
   bucle: real;                { Para bucles, claro }
   pantalla: array [0..199,0..319]  of byte
     absolute $A0000;          { Pues la pantalla ;-) }

   tecla: char;                { La tecla que se pulse }

 procedure ModoPantalla( modo: byte );
                               { Cambia a un modo dado }
 begin
   regs.ah := 0;               { Función 0 }
   regs.al := modo;            { El modo indicado }
   intr($10,regs);             { Interrupción de video }
 end;
```

```
procedure PonPixel(x,y: word; color: byte);       { Dibuja Pixel }
begin
  Pantalla[y, x] := color;
end;

procedure Linea(x, y, x2, y2 : word; color: byte);
var
  d,
  dx, dy,              { Salto total según x e y }
  ai, bi,
  xi, yi               { Incrementos: +1 ó -1, según se recorra }
          : integer;
begin
  if (x < x2) then     { Si las componentes X están ordenadas }
  begin
    xi := 1;           { Incremento +1 }
    dx := x2 - x;      { Espacio total en x }
  end
  else                 { Si no están ordenadas }
  begin
    xi := - 1;         { Increm. -1 (hacia atrás) }
    dx := x - x2;      { y salto al revés (negativo) }
  end;
  if (y < y2) then     { Análogo para las componentes Y }
  begin
    yi := 1;
    dy := y2 - y;
  end
  else
  begin
    yi := - 1;
    dy := y - y2;
  end;
  PonPixel(x, y,color);    { Dibujamos el primer punto }
  if dx > dy then      { Si hay más salto según x que según y }
  begin                { (recta más cerca de la horizontal) }
    ai := (dy - dx) * 2;    { Variables auxiliares del algoritmo }
    bi := dy * 2;           { ai y bi no varían; d comprueba cuando }
    d  := bi - dx;          { debe cambiar la coordenada y }
    repeat
      if (d >= 0) then      { Comprueba si hay que avanzar según y }
      begin
        y := y + yi;        { Incrementamos Y (+1 ó -1) }
        d := d + ai;        { y la variable de control }
      end
      else
        d := d + bi;        { Si no varía y, d sí lo hace según bi }
      x := x + xi;          { Incrementamos X como corresponda }
      PonPixel(x, y, color);          { Dibujamos el punto }
    until (x = x2);    { Se repite hasta alcanzar el final }
  end
  else                 { Si hay más salto según y que según x }
  begin                { (más vertical), todo similar }
    ai := (dx - dy) * 2;
    bi := dx * 2;
    d  := bi - dy;
    repeat
      if (d >= 0) then
      begin
        x := x + xi;
```

```
        d := d + ai;
      end
      else
        d := d + bi;
      y := y + yi;
      PonPixel(x, y, color);
    until (y = y2);
  end;
end;

begin
  ModoPantalla($13);       { Modo 320x200x256 }
  bucle := 0;              { Empezamos en 0 __RADIANES__ }
  repeat
    linea(160,100,         { Línea desde el centro de la pantalla }
      160 + round(60*cos(bucle)),   { Extremo en un círculo }
      100 + round(40*sin(bucle)),
      0);                           { Color negro (borrar) }
    bucle := bucle + 0.1;           { Siguiente posición }
    linea(160,100,          { Otra línea, pero ahora blanca }
      160 + round(60*cos(bucle)), 100 + round(40*sin(bucle)),
      15);
    delay(25);              { Esperamos 25 milisegundos }
  until keyPressed;         { Seguimos hasta que se pulse una tecla }
  tecla := ReadKey;         { Quitamos esa tecla del buffer del teclado
}
  ModoPantalla(3);          { Y volvemos a modo texto }
end.
```

(Sólo cambia la forma de acceder a la pantalla y el procedimiento "PonPixel).

Pero imaginad que estamos rotando una figura complicada, con cientos de puntos, y que además no trabajamos en el plano, sino en el espacio, con lo que tenemos rotaciones en torno a tres ejes (teneis un ejemplo después de la ampliación 5: ensamblador desde Turbo Pascal que, dicho sea de paso, cuenta cómo corregir un fallo del algoritmo que he puesto antes para dibujar líneas).

Si experimentais, incluso complicando este ejemplillo, vereis que a medida que aumenta la complejidad de lo que hay que rotar, se va haciendo más evidente la **lentitud** de este método.

Se diría que las demos que a todos nos asombran no pueden estar hechas así, ¿verdad? Pues el truco se llama **tablas**. Nada más y nada menos. En vez de calcular cada pasada el coseno de 10 grados, se calcula una vez este valor al principio del programa y se guarda en un ARRAY, con lo cual no accederemos como "cos(10)" sino como "coseno[10]".

Esa es la primera mejora, pero aun hay más. Multiplicar por números reales es lento, así que la segunda mejora que se nos puede ocurrir es trabajar con **números enteros**. ¿Pero cómo, si el seno y el coseno van de 0 a 1? Pues multiplicándolos por 100 o 256, por ejemplo, antes de guardarlos en

nuestro array. Al fin y al cabo, en nuestra pantalla todas las coordenadas son enteras. Basta tenerlo en cuenta a la hora de multiplicar por el radio para que no se nos salga de la pantalla... ;-) Además, es mejor usar números como 256 o 128 que 100 o 200. ¿Por qué? Por lo que ya hemos comentado antes: las multiplicaciones y divisiones por múltiplos de dos se pueden expresar como rotaciones de bits (SHL y SHR), mucho más rápidas que una multiplicación en general.

(Hay un ejemplo de todo esto como ampliación al curso: entre los fuentes de ejemplo, hablando de rotaciones en 3D).

Y eso de las tablas se usa en más de una ocasión cuando queremos optimizar rutinas gráficas. Algo tan sencillo como nuestro "PonPixel" contiene una multiplicación. ¿Y si dibujamos 50.000 puntos, hacemos 50.000 multiplicaciones? Se puede evitar con un nuevo array, de modo que en vez de hacer "y*320" se escriba "Por320[y]".

Incluso efectos cómo ese de una lente o una bola de cristal que pasa por encima de un dibujo, y se ve a través suyo el fondo deformado, suelen estar basados en tablas para mayor rapidez...

Pero todo eso y más lo dejo para que jugueis. En el próximo apartado vemos un poco cómo manejar la paleta de colores, y damos por terminada la parte del curso relativa a gráficos.

Ya hemos comentado cómo entrar a un modo gráfico dado, como dibujar un punto, dibujar líneas, círculos o elipses, como hacer rotaciones en el plano, algunas optimizaciones sencillas...

Hoy vamos a tratar cómo borrar la pantalla, como cambiar la paleta de colores, cómo sincronizar con el barrido de la pantalla, cómo escribir en la pantalla gráfica y a dar una idea sobre cómo mover datos en memoria, aplicándolo a un Scroll sencillo.

Borrar la pantalla

Veamos... lo de **borrar la pantalla** es MUY sencillo. Si estamos en nuestro modo 320x200, 256 col., la pantalla ocupaba 64000 bytes, que estaban a partir de la dirección $A000:0000 ¿verdad?

Pues entonces podemos situar una matriz en esa posición de la memoria, como hicimos en el primer apartado de este tema:

```
var pantalla: array[0..319,0..199] of byte absolute $A000:0000;
```

o bien simplemente

```
var pantalla: array[1..64000] of byte absolute $A000:0000;
```

Con cualquiera de estas definiciones, podemos usar la orden "**fillchar**", que llena con un cierto byte unas posiciones de memoria que le indiquemos:

```
fillchar(pantalla, 64000, 0);
```

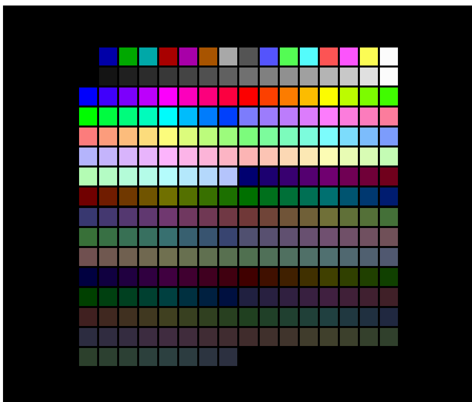
que quiere decir "lléname la variable pantalla con ceros, ocupando 64000 bytes".

Un uso más adecuado para evitar errores, ya que "fillchar" no hace comprobaciones de tamaño, es usar "sizeof" para que el compilador sea quien calcule cuantos ceros hay que poner:

```
fillchar(pantalla, sizeof(pantalla), 0);
```

Lógicamente, si queremos borrar la pantalla empleando el color 2 en vez del 0 (que normalmente es el fondo), basta poner un 2 en vez del cero... ;-) En el caso más general, sería

```
procedure BorraPantalla( color: byte );
begin
  fillchar(pantalla, sizeof(pantalla), color);
end;
```



La paleta de colores

Sigamos... ¿La **paleta de colores**? ¿Qué es eso? Pues supongo que cualquiera que haya llegado hasta aquí recordará que estamos trabajando con 256 colores. Pero estos colores no tienen por qué ser siempre iguales: el color 0 no tiene por qué ser siempre negro. Podemos modificar cualquiera de

los 256 colores físicos indicando la cantidad de Rojo, Verde y Azul (RGB) que queremos que tenga. Esto lo hacemos accediendo a través de puertos:

Tenemos las siguientes **direcciones** importantes:

```
$3c7 =>  Dirección de lectura
$3c8 =>  Dirección de escritura
$3c9 =>  Dirección de datos (R,G y B consecutivos)
```

y la intensidad máxima para R, G o B será 63.

¿A que asusta? No hay motivo, es más fácil de lo que parece: para cambiar las componentes R, G y B de un color dado, hacemos

```
procedure FijaColor( Color, r, g, b: Byte );
begin
  Port[$3c8] := Color;     { Elegimos el color a modificar }
  Port[$3c9] := r;         { Primer dato: cuanto Rojo queremos }
  Port[$3c9] := g;         { Segundo: verde }
  Port[$3c9] := b;         { Tercero: azul }
end;
```

Insisto: R, G y B van de 0 a 63, lo que da un total de 262.144 posibles combinaciones de colores que podemos tomar para rellenar nuestros 256 colores "físicos".

Análogamente, para leer las componentes de un color, sería

```
procedure LeeColor( Color: Byte );
begin
  Port[$3c7] := Color;     { Elegimos el color a leer }
  r := Port[$3c9];         { Primer dato: cuanto Rojo hay }
  g := Port[$3c9];         { Segundo: verde }
  b := Port[$3c9];         { Tercero: azul }
end;
```

donde se ha supuesto que R, G y B son variables globales. También podría ser una función que devolviera los datos en un array, o haber pasado R, G y B como parámetros por referencia (precedidos por la palabra "var").

Esto tiene aplicación inmediata para hacer fundidos de pantalla y similares, o para muchos efectos vistosos. Me voy a limitar a poner uno que funciona incluso en modo texto: vamos a hacer que el fondo pase de negro a blanco y luego vuelva a negro.

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Fundido de la panta-  }
{    lla a blanco y luego  }
{    a negro               }
```

```
{    GRB4.PAS                }
{                            }
{  Este fuente procede de    }
{  CUPAS, curso de Pascal    }
{  por Nacho Cabanes         }
{                            }
{  Comprobado con:           }
{    - Turbo Pascal 7.0      }
{    - Tmt Pascal Lt 1.01    }
{--------------------------}

program GrB4;

uses crt;

var i: byte;

procedure FijaColor( Color, r, g, b: Byte );
begin
  Port[$3c8] := Color;     { Elegimos el color a modificar }
  Port[$3c9] := r;         { Primer dato: cuanto Rojo queremos }
  Port[$3c9] := g;         { Segundo: verde }
  Port[$3c9] := b;         { Tercero: azul }
end;

begin
for i := 0 to 63 do     { Recorremos todas las intensidades }
  begin
  FijaColor(0,i,i,i);   { Igual cantidad de R, G, B => gris }
  delay(10);            { Esperamos antes de seguir }
  end;
for i := 63 downto 0 do     { Igual, descendiendo }
  begin
  FijaColor(0,i,i,i);
  delay(15);
  end;
end.
```

Sin parpadeos..

Cuando enviamos mucha información a la pantalla o cuando cambiamos los colores con rapidez, podremos apreciar a veces "**parpadeos**". Esto también es más fácil de evitar de lo que parece: basta empezar a dibujar cuando llegue el barrido (en inglés "retrace") de la pantalla... :-o ¿Y cómo se hace eso? Seguro que lo ha preguntado más de uno... ;-) Pues esperando a que comience el barrido actual y luego a que empiece el siguiente, así:

```
procedure Retrace;
begin
  repeat until (port[$3da] and 8) = 8;
  repeat until (port[$3da] and 8) <> 8;
end;
```

¡ Magia ! }:-D Pues basta con escribir "Retrace;" antes de empezar a dibujar, y mejorará bastante (si nuestro dibujo no es muy lento), además de que conseguimos una velocidad bastante independiente del ordenador.

Escribir texto

Hay dos formas de **escribir texto**:

Una, que no vamos a tratar, consiste en definir nuestra propias letras, o saber dónde está la definición de las que trae el ordenador, para mirar esa zona de memoria, analizarla y escribir las letras pixel a pixel.

Otra más sencilla, aunque menos versátil, es con la orden "write" igual que hacíamos en modo texto. Para ello, basta usar la unidad CRT pero indicando **DirectVideo := false** con lo que la escritura pasa a través de la BIOS, en vez de hacerse directamente en la memoria de pantalla. Así conseguimos que se pueda enviar a la pantalla gráfica. Veremos un ejemplo al final del tema, en el ejemplo de Scroll.

Finalmente, para copiar zonas de memoria podemos utilizar **move**, que tiene el formato move (org, dest, cuanto) Esto no nos sirve directamente para zonas rectangulares de la pantalla, porque el origen debe ser una sóla zona contigua, lo que no ocurre en un rectángulo, pero podemos evitarlo copiando fila a fila con un bucle.

¿Y cómo se usa? Pues vamos a ver un ejemplillo, aprovechando que en nuestro modo de pantalla cada byte se corresponde con un pixel, para hacer un **Scroll** (desplazamiento) de parte de la pantalla, para que aparezca un texto:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Scroll sencillo en la }
{    parte inferior de la  }
{    pantalla              }
{    GRB5.PAS              }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{    - Tmt Pascal Lt 1.01  }
{--------------------------}

Uses
  Crt, dos;

Const
```

```
  Fila = 23;   { Fila (-1) en la que se escribe el mensaje (base 0) }

Var
  pixel, LetraActual : Byte;  { Para bucles }
  regs: registers;            { Para interrupciones (ModoPantalla) }
  tecla: char;                { Para absorber la tecla que se pulse }

procedure ModoPantalla( modo: byte );        { Cambia a un modo dado }
begin
  regs.ah := 0;               { Función 0 }
  regs.al := modo;            { El modo indicado }
  intr($10,regs);             { Interrupción de video }
end;

procedure Retrace;            { Espera el barrido de la pantalla }
begin
  repeat until (port[$3da] and 8) = 8;
  repeat until (port[$3da] and 8) <> 8;
end;

Procedure Mueve;          { Desplaza el texto a la izqda un pixel }
Var
  LineaAct : Word;        { La línea que se está moviendo }
begin
                      { Desplaza las 8 líneas 1 pixel hacia la izqda }
  For LineaAct := (Fila * 8) to (Fila * 8) + 7 DO
    Move(Mem[$A000 : LineaAct * 320 + 1], Mem[$A000 : LineaAct *
320], 319);
                          { Borra los pixels del final }
  For LineaAct := (Fila * 8) to (Fila * 8) + 7 DO
    Mem[$A000 : LineaAct * 320 + 319] := 0;
end;

procedure ScrollInferior(mensaje:string); { Esto es lo importante }
begin
  ModoPantalla($13);          { Modo Mcga/Vga 320x200x256 }
  DirectVideo := False;       { Para escribir texto en modo gráfico }
  GotoXY(1, Fila + 1);        { Vamos a la posición }
  repeat
  For LetraActual := 1 to Length(Mensaje) do      { Para cada letra }
  begin
    For pixel := 1 to 8 do                 { Los 8 pixels de ancho }
    begin
      Mueve;                               { Mueve el letrero }
      Retrace;                             { Espera al barrido }
      if keypressed then exit;             { Si se pulsa tecla: fuera }
    end;
    GotoXY(40, Fila+1);                    { En la posición adecuada }
    Write(Mensaje[LetraActual]);           { Se escribe la sgte letra }
  end;
  until keypressed;
end;

begin
  ScrollInferior('Esto es un  - S C R O L L -                ');
  tecla := readkey;                 { Vacía el buffer del teclado }
  ModoPantalla(3);                  { Vuelve a modo texto }
end.
```

Una recomendación final: en vuestros programas (si es que los haceis X-D) no se os ocurra copiar el procedimiento "linea", "modopantalla" y similares cada vez. Lo más cómodo es crear una **unidad**, meter nuestros procedimientos y funciones relacionados con los gráficos y acceder a ella cuando nos interese.

Así además, si encontrais otro procedimiento más rápido para dibujar líneas (los hay, si trabajamos en ensamblador, por ejemplo) o cualquier otra mejora, todos los programas la tendrán sólo con recompilar.

Curso de Pascal. Ampliación 3 - Ordenaciones

Este es un tema más propio de un curso de algorítmica que de uno de programación en Pascal, pero como la mayoría de los programas tienen que ordenar los datos para conseguir una acceso más rápido y sencillo, voy a comentar dos de los métodos de ordenación más conocidos, uno sencillo y lento, y otro más rápido y difícil de entender: **burbuja** y **QuickSort**.

En ellos voy a suponer que la lista de datos es un array, y que queremos ordenar esta lista de menor a mayor. Trabajaré con datos de tipo integer por simplicidad, pero adaptarlo a Strings, Records u otros tipos no tiene mayor dificultad.

Al final comentaré también algo sobre la "**búsqueda binaria**", para que se vea a qué me refiero con eso de "acceso más rapido".

Burbuja

Es posible que este sea el algoritmo más sencillo de escribir, y también el menos eficiente. La idea es sencilla: se van desplazando los números más pequeños hacia atrás hasta su posición correcta (de ahí el nombre de "burbuja": es como si los más ligeros fueran subiendo).

La idea es simplemente ésta:

```
for i := maximo downto 2 do                { De final a principio }
  if datos[i] < datos[i-1] then            { Si está colocado al revés }
    swap(datos[i], datos[i-1]);            { Le da la vuelta }
```

es decir, recorre la lista desde el final hacia el principio, comparando cada elemento con el anterior. Si están colocados al revés (primero el mayor y luego el menor), hay que intercambiarlos.

La rutina de intercambio (swap) sería así:

```
procedure swap(var a,b: integer);         { Intercambia dos datos }
var
  tmp: integer;
begin
  tmp := a;
  a := b;
  b := tmp;
end;
```

Esto coloca el último número de la lista en su posición adecuada. Nos puede servir si añadimos siempre un único elemento, que colocamos al final temporalmente y después llevamos a su posición adecuada.

Pero es muy frecuente que haya más de un dato descolocado, y que además no sepamos cuántos son estos datos que están desordenados. Entonces debemos "mejorar" la idea anterior para colocar todos los números que haga falta.

La primera forma intuitiva sería incluir el "for" anterior dentro de otro "for", para comprobar que todos los números quedan colocados. Esto es sencillo de programar, pero es ineficiente: si sólo hay un número descolocado, pero aun así probamos todos uno por uno, perdemos tiempo inútilmente. La forma sería:

```
for j := 1 to maximo-1 do
  for i := maximo downto j+1 do         { De final a principio }
    if datos[i] < datos[i-1] then       { Si está colocado al revés }
      swap(datos[i], datos[i-1]);       { Le da la vuelta }
```

Una forma más rápida para datos que no estén muy desordenados será con un "while" o un "repeat": si damos una pasada a toda la lista sin recolocar ningún número, es que ya están todos colocados, y entonces no tiene sentido seguir mirando. Entonces, nuestra rutina de ordenación quedaría, por ejemplo:

```
procedure Burbuja;                      { Ordena según burbuja }
var
  cambiado: boolean;
begin
  writeln;
  writeln('Ordenando mediante burbuja...');
  repeat
    cambiado := false;                  { No cambia nada aún }
```

```
    for i := maximo downto 2 do         { De final a principio }
      if datos[i] < datos[i-1] then     { Si está colocado al revés }
        begin
        swap(datos[i], datos[i-1]);     { Le da la vuelta }
        cambiado := true;               { Y habrá que seguir mirando }
        end;
  until not cambiado;                   { Hasta q nada haya cambiado }
end;
```

Este algoritmo puede ser útil para datos poco desordenados, en los que añadamos datos al final. Si se trata de datos muy desordenados, resulta lento.

Hay otros bastante parecidos a éste, que van comparando y moviendo elementos de uno en uno. En cambio, hay otros que se basan en "bloques", y que suelen ser recursivos, como MergeSort y QuickSort.

MergeSort

Este algoritmo de ordenación es más eficiente que el anterior para datos bastante desordenados. Se basa en aquello tan popular de "divide y vencerás". La idea en sí también es sencilla: partir un problema complicado en trozos que sean más sencillos de resolver uno por uno.

En este caso, se trata de dividir nuestra lista en "sublistas" cada vez más pequeñas (se hará de forma recursiva). Finalmente, volvemos a juntar todas esas listas en una sola.

¿Y cómo las juntamos? Pues miramos el primer elemento de cada una y los vamos cogiendo por orden, hasta reconstruir la lista inicial, pero ya ordenada.

Este algoritmo no voy a desarrollarlo, porque me centraré en QuickSort, que suele ser más rápido. Aun así, voy a poner el pseudo-código, por si alguien quiere implementarlo:

```
función MERGESORT (a: lista): lista
  opcion
    |a| = 1: devolver a               { Tamaño 1: fin de recursión }
    |a| <> 1:
       (x1,x2) := DESCOMPONER(a)      { Si no, divide en dos trozos }
       s1 := MERGESORT (x1)           { Aplica recursivamente a los }
       s2 := MERGESORT (x2)           {   dos trozos }
       devolver MERGE(s1,s2)          { Y finalmente combina en una }
  fopc                                {   única lista grande }
ff
```

QuickSort

En un caso "general", puede que éste sea el algoritmo más rápido (aunque en muchos casos particulares puede no serlo).

Lo que este hace es mirar el valor del centro de la lista. Mueve a su derecha todos los valores menores y a la izquierda todos los mayores, pero no los ordena aún, sino que luego recursivamente vuelve a hacer lo mismo en ambos trozos. Así finalmente queda ordenado.

Más en detalle: dentro de cada uno de esos dos trozos (valores mayores y menores que el del centro), avanza desde su extremo hacia el centro, comprobando qué valores ya están colocados donde les corresponde, y moviendo al lado opuesto los que no lo estén.

Así, la parte recursiva del algoritmo es:

```
procedure Sort(l, r: Integer);          { Esta es la parte recursiva }
var
  i, j, x, y: integer;
begin
  i := l; j := r;                       { Límites por los lados }
  x := datos[(l+r) DIV 2];              { Centro de la comparaciones }
  repeat
    while datos[i] < x do i := i + 1;   { Salta los ya colocados }
    while x < datos[j] do j := j - 1;   {   en ambos lados }
    if i <= j then                      { Si queda alguno sin colocar }
      begin
      swap(datos[i], datos[j]);         { Los cambia de lado }
      i := i + 1; j := j - 1;           { Y sigue acercándose al centro
}
      end;
  until i > j;                          { Hasta que lo pasemos }
  if l < j then Sort(l, j);             { Llamadas recursivas por cada
}
  if i < r then Sort(i, r);             {   lado }
end;
```

Pues con todo esto, vamos a ver ya un programa de ejemplo que use estos dos métodos para ordenar un array de números enteros:

```
 {--------------------------}
 {  Ejemplo en Pascal:      }
 {                          }
 {    Ejemplo de ordenacio- }
 {    nes: Burbuja y Quick- }
 {    sort                  }
 {    ORDENA.PAS            }
 {                          }
 {  Este fuente procede de  }
 {  CUPAS, curso de Pascal  }
```

```
{  por Nacho Cabanes         }
{                            }
{  Comprobado con:           }
{    - Free Pascal 2.2.0w    }
{    - Turbo Pascal 7.0      }
{    - Tmt Pascal Lt 1.01    }
{--------------------------}

program Ordenar;

const
  maximo = 100;             { Número de datos }
  maxVal = 30000;           { Maximo valor que pueden tomar }

var
  datos: array[1..maximo] of integer;      { Los datos en sí }
  i: integer;                              { Para bucles }

procedure swap(var a,b: integer);          { Intercambia dos datos }
var
  tmp: integer;
begin
  tmp := a;
  a := b;
  b := tmp;
end;

procedure generaNumeros;                 { Genera números aleatorios }
begin
  writeln;
  writeln('Generando números...');
  for i := 1 to maximo do
    datos[i] := random(maxVal);
end;

procedure muestraNumeros;                { Muestra los núms almacenados
}
begin
  writeln;
  writeln('Los números son...');
  for i := 1 to maximo do
    write(datos[i], ' ');
  writeln;
end;

procedure Burbuja;                       { Ordena según burbuja }
var
  cambiado: boolean;
begin
  writeln;
  writeln('Ordenando mediante burbuja...');
  repeat
    cambiado := false;                   { No cambia nada aún }
    for i := maximo downto 2 do          { De final a principio }
      if datos[i] < datos[i-1] then      { Si está colocado al revés }
        begin
        swap(datos[i], datos[i-1]);      { Le da la vuelta }
        cambiado := true;                { Y habrá que seguir mirando }
        end;
  until not cambiado;                    { Hasta q nada se haya
cambiado }
```

```
 end;

 procedure QuickSort;                      { Ordena según Quicksort }

 procedure Sort(l, r: Integer);            { Esta es la parte recursiva }
 var
   i, j, x, y: integer;
 begin
   i := l; j := r;                         { Límites por los lados }
   x := datos[(l+r) DIV 2];                { Centro de la comparaciones }
   repeat
     while datos[i] < x do i := i + 1;     { Salta los ya colocados }
     while x < datos[j] do j := j - 1;     {   en ambos lados }
     if i <= j then                        { Si queda alguno sin colocar
}
       begin
       swap(datos[i], datos[j]);           { Los cambia de lado }
       i := i + 1; j := j - 1;             { Y sigue acercándose al
centro }
       end;
   until i > j;                            { Hasta que lo pasemos }
   if l < j then Sort(l, j);               { Llamadas recursivas por cada
}
   if i < r then Sort(i, r);               {   lado }
 end;

 begin                            { Esto llama a la parte recursiva
}
   writeln;
   writeln('Ordenando mediante QuickSort...');
   Sort(1,Maximo);
 end;


 { ------------ Cuerpo del programa ------------- }
 begin
   randomize;
   generaNumeros;
   muestraNumeros;
   Burbuja;
   muestraNumeros;
   readln;
   generaNumeros;
   muestraNumeros;
   QuickSort;
   muestraNumeros;
   readln;
 end.
```

Búsqueda binaria.

¿Para qué tener los datos ordenados? Pues no sólo para que queden más bonitos en pantalla ;-) Si tenemos 20.000 datos, al hacer una búsqueda nos puede ayudar muchísimo saber que están ordenados.

¿A qué me refiero? Es sencillo: si los datos no están ordenados, para comprobar si un dato ya existe, tendremos que mirar todos. En cambio, si están ordenados sabemos dónde buscar: si queremos saber si "Martínez" está en nuestra agenda (o base de datos, por eso de que ya tiene 20.000 datos y ya es más que una simple agenda ;-)) iremos directamente a la M. Si llegamos a la N (o incluso a "Mas...") y no ha aparecido, sabemos que no está.

Esto es lo que hacemos en un diccionario, por ejemplo, y esta es la idea de la búsqueda binaria.

Lo de "binario" es porque lo que haremos será siempre mirar el valor que tenemos en el medio. Si nos hemos pasado, vamos hacia la izquierda; si nos hemos quedado cortos, vamos a la derecha. En cualquiera de los dos casos, volvemos a mirar en el medio, y así sucesivamente.

Si al final encontramos el dato, claramente es que estaba, y ya sabemos cual es la ficha que debemos modificar; si nos pasamos no estaba, y tendremos que avisar al usuario, o añadir una nueva ficha. Pero hemos hecho muchas menos comparaciones que mirando una por una. ¿Cuantas menos? Pues como cada vez vamos dividiendo a la mitad el tamaño de la zona a mirar, el número de comparaciones estará muy cerca del logaritmo en base 2 del número de datos que teníamos. Así para nuestra base de datos de 20.000 personas, haremos

log2 (20.000) = 11 aprox. => cerca de 12 comparaciones

La cuenta sale rápido: antes teníamos que hacer 20.000 comparaciones en el peor caso, ó 1 en el mejor caso. Consideremos que como media deberíamos hacer 10.000 comparaciones. Ahora hacemos unas 12 (puede que sea 11, o bien 13 ó 14, según lo "cuidadosos" que seamos programando, pero no más) Por tanto, ahora la búsqueda es 1.000 veces más rápida.

Es decir, si el señor "Martínez" nos dice que ha cambiado su dirección, y queremos apuntarlo en nuestra base de datos, la diferencia es tardar 2 segundos en encontrar su ficha (con búsqueda binaria) o cerca de media hora (2.000 segundos, con búsqueda lineal).

Pero vale ya de rollos y vamos a ver un ejemplo, que cree una lista de números, la ordene y luego busque un valor en ella:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Búsqueda binaria      }
{    BUSCABIN.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
```

```
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Free Pascal 2.2.0w   }
{    - Turbo Pascal 7.0     }
{--------------------------}

program BuscaBin;         {  Búsqueda binaria }

const
  maximo = 400;           { Número de datos }
  maxVal = 500;           { Maximo valor que pueden tomar }

var
  datos: array[1..maximo] of integer;     { Los datos en sí }
  i: integer;                             { Para bucles }
  donde: integer;                         { Posicion en la que ha }
                                          {   aparecido }
procedure swap(var a,b: integer);         { Intercambia dos datos }
var
  tmp: integer;
begin
  tmp := a;
  a := b;
  b := tmp;
end;

procedure generaNumeros;                { Genera números aleatorios }
begin
  writeln;
  writeln('Generando números...');
  for i := 1 to maximo do
    datos[i] := random(maxVal);
end;

procedure muestraNumeros;               { Muestra los núms.
almacenados }
begin
  writeln;
  writeln('Los números son...');
  for i := 1 to maximo do
    write(datos[i], ' ');
  writeln;
end;

procedure Burbuja;                      { Ordena según burbuja }
var
  cambiado: boolean;
begin
  writeln;
  writeln('Ordenando mediante burbuja...');
  repeat
    cambiado := false;                  { No cambia nada aún }
    for i := maximo downto 2 do         { De final a principio }
      if datos[i] < datos[i-1] then     { Si está colocado al revés }
        begin
        swap(datos[i], datos[i-1]);     { Le da la vuelta }
        cambiado := true;               { Y habrá que seguir mirando }
        end;
  until not cambiado;                   { Hasta q nada se haya
cambiado }
```

```
end;

function Buscar(minimo, maximo, valor: integer): integer;
                                                  { Búsqueda binaria }
var
  medio: integer;
begin
  writeln('Mirando entre ', minimo, ' y ', maximo,
    ', de valores ', datos[minimo], ' y ', datos[maximo]);

  if minimo >= maximo then          { Si la anchura ya es 1 }
   if datos[minimo]=valor then     {   y el valor es el buscado }
     buscar := minimo              {   devolvemos su posición }
   else buscar := -1               { Si es otro valor -> no está }
  else                             { Si la anchura no es 1 }
    begin
    medio := round((minimo+maximo)/2);       { Hallamos el centro }
    if valor = datos[medio] then             { Comparamos con su valor
}
      buscar := medio                        { Si acertamos, ya esta }
    else if valor > datos[medio] then        { Si no, }
      buscar := buscar(medio+1,maximo,valor)  { Miramos el lado
corresp}
    else
      buscar := buscar(minimo,medio-1,valor)
    end;
end;

{ ------------ Cuerpo del programa ------------- }
begin
  randomize;
  generaNumeros;
  Burbuja;
  muestraNumeros;
  writeln('Buscando ',maxVal div 2,'...');
  donde := Buscar(1,maximo, maxVal div 2);
  if donde = -1 then writeln('No se ha encontrado')
    else writeln('Está en la posición ',donde);
  readln;
end.
```

(Por supuesto, esto es mejorable. Por ejemplo, en un diccionario no tenemos sólo 2 posibilidades, sino cerca de 25 letras por las que empezar a buscar, de modo que es más rápido todavía).

Curso de Pascal. Ampliación 4. Overlays.

Si trabajamos con MsDos y su famoso límite de 640 Kb, nos podemos encontrar con datos que ocupen más espacio que la memoria que tenemos disponible en nuestro ordenador.

Esto puede parecer un problema en un principio, pero una vez que se sabe cómo usar el acceso aleatorio a un fichero, la solución es fácil: leemos sólo los datos que necesitemos en cada momento.

Pero ¿qué ocurre si es nuestro programa el que necesita más memoria de la que tenemos? ¿O simplemente si necesitamos más memoria libre? Es habitual que, igual que ocurre con los datos, muchas partes de un programa se usan con poca frecuencia. En cambio, en un programa no existe nada parecido al acceso aleatorio para tener en memoria sólo las rutinas que nos interesan en un momento concreto.

¿Nada? Eso no es cierto del todo. En Turbo Pascal tenemos una forma de conseguirlo: empleando **overlays** (En Free Pascal todo esto no es necesario: tenemos acceso a toda la memoria del sistema sin necesidad de estos "trucos", así que podemos manejar programas enormes de forma transparente).

La palabra "overlay" sale a querer decir algo parecido a "solapamiento". Y justo esa es la idea: al igual que hacíamos con los datos, podremos tener procedimientos o funciones que se "solapen" en memoria: en cada momento sólo tenemos uno de ellos en la memoria central (RAM), y cuando hace falta otro, éste sale y le deja su hueco en memoria.

Imaginemos que nuestro programa presenta un mensaje al principio de bienvenida (que sólo se emplea una vez en cada ejecución), otro de despedida al final (pasa lo mismo), y hay una rutina de ordenación de los datos, que sólo usamos una vez por semana. Lo que nos interesa es, en vez de tener las 3 cosas en memoria, cargarlas sólo cuando las necesitemos. Y aun hay más: como no tienen por qué ser simultáneas (no se ordena a la vez que se saluda, por ejemplo), podríamos liberar la memoria que ha usado una de ellas antes de cargar la otra.

Eso es lo que nos permite hacer el sistema de "overlays", que Turbo Pascal incorpora como estándar.

Ahora que vamos sabiendo por donde van los tiros, vamos a empezar a aplicarlo.

El programa debe empezar por incluir la **unidad Overlay**:

```
uses overlay, ...
```

También debemos incluir las unidades que queremos solapar:

```
uses overlay, crt, dos, miUnit1, miUnit2 ...
```

Ahora tenemos que emplear 3 directivas de compilación:

- $F+ para indicar al compilador que queremos llamadas lejanas (far, fuera del segmento actual).
- $O+ para decirle que permita usar overlays.
- $O UNIDAD para indicar qué unidades queremos solapar.

Con todo esto, nuestro programa estaría quedando:

```
{$F+,O+}
program PruebaOverlays;

uses overlay, crt, dos, miUnit1, miUnit2;

{$O miUnit1}
{$O miUnit2}

begin
    [...]
```

¿Más cosas? Apenas un par de ellas. Ya en el cuerpo del programa, usamos **OvrInit** para inicializar el sistema de overlays e indicar a nuestra aplicación en qué fichero se encuentran éstos:

```
OvrInit ( 'MIPROG.OVR' );
```

Ahora podemos **comprobar** que todo ha ido bien, mirando el valor de OvrResult. Este será 0 si no ha habido problemas, o un número distinto si no ha podido inicializar todo correctamente.

Hay definidas unas constantes simbólicas para simplificarnos esas comparaciones:

```
     ovrOk          =  0;   { Todo correcto }
     ovrError       = -1;   { Error del manejador de overlays;
                              normalmente será que no hemos creado
bien
                              los overlays }
     ovrNotFound    = -2;   { Fichero de overlays no encontrado. No
                              ocurrirá al compilar, pero puede pasar
                              después, si no distribuimos el .OVR
junto
                              al .EXE }
     ovrNoMemory    = -3;   { No hay memoria suficiente para la zona
de
                              solapamiento }
```

```
    ovrIOError     = -4;    { Error de E/S con el fichero OVR }
    ovrNoEMSDriver = -5;    { No existe memoria EMS (expandida) }
    ovrNoEMSMemory = -6;    { No hay suficiente memoria EMS }
```

¿Por qué habla por ahí de EMS? Pues porque podemos decirle además al compilador que queremos que nuestro programa intente cargar los overlays en memoria expandida (EMS). Si lo consigue, será más rápido leer los procedimientos desde ahí que desde el disco. Si no lo consigue, tampoco hay problema, porque entonces trabajaría con el disco como si nada.

¿Y cómo le pedimos que lo intente cargar en EMS? Pues simplemente escribiendo

```
OvrInitEMS;
```

La última consideración, pero no menos importante, es que para que todo esto funcione debemos **compilar a disco**.

¿Las unidades a solapar? Son unidades normales y corrientes, en las que debemos indicar al principio {$F+,O+}

Vamos ya a ver un ejemplo de cómo se aplica todo esto: El programa principal sería

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Programa que usa      }
{    Overlays              }
{    MIOVR.PAS             }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

{$F+,O+}                    { Necesario }

program MiOvr;

uses
  Overlay, Crt,             { Unidades que empleamos }
  MiUnit1, MiUnit2;

{$O MiUnit1}                { Indicamos cuales queremos solapar }
{$O MiUnit2}

var
  tecla: char;
```

```
begin
  ClrScr;
  OvrInit('MIOVR.OVR');       { Inicializar y reservar memoria }
  {OvrInitEMS;}
  if OvrResult <> 0 then      { Si hay algún error }
  begin
    WriteLn('Error en el sistema de overlays: ', OvrResult);
    Halt(1);
  end;
  repeat
    Proc1;                    { De la unidad 1 }
    Proc2;                    { De la unidad 2 }
    WriteLn;                  { Para que se lea mejor }
    delay(50);                { Y esperamos un poco }
  until KeyPressed;           { Así hasta que se pulse una tecla }
  tecla := ReadKey;           { Absorbemos la tecla pulsada }
end.
```

La primera unidad podría ser:

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Primera unidad sola-  }
{    para para MIOVR       }
{    MIUNIT1.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

{$O+,F+}

unit MiUnit1;

interface

procedure Proc1;


implementation

procedure Proc1;
begin
  Writeln('Estoy en la primera unidad.');
end;

end.
```

Y la segunda unidad:

```
{--------------------------}
```

```
{  Ejemplo en Pascal:       }
{                           }
{    Segunda unidad sola-   }
{    para para MIOVR        }
{    MIUNIT2.PAS            }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Turbo Pascal 7.0     }
{--------------------------}

{$O+,F+}

unit MiUnit2;

interface

procedure Proc2;


implementation

procedure Proc2;
begin
  Writeln('Estoy en la segunda unidad.');
end;

end.
```

Y si queremos emplear EMS, las unidades no cambian, y el programa principal podría ser:

```
{--------------------------}
{  Ejemplo en Pascal:       }
{                           }
{    Programa que usa       }
{    Overlays y los guarda  }
{    en memoria EMS         }
{    MIOVR2.PAS             }
{                           }
{  Este fuente procede de   }
{  CUPAS, curso de Pascal   }
{  por Nacho Cabanes        }
{                           }
{  Comprobado con:          }
{    - Turbo Pascal 7.0     }
{--------------------------}

{$F+,O+}                      { Necesario }

program MiOvr2;

uses
  Overlay, Crt,               { Unidades que empleamos }
  MiUnit1, MiUnit2;
```

```
{$O MiUnit1}                    { Indicamos cuales queremos solapar }
{$O MiUnit2}

var
  tecla: char;

begin
  ClrScr;
  OvrInit('MIOVR.OVR');         { Inicializar y reservar memoria }
  OvrInitEMS;
  if (OvrResult = -5) or
     (OvrResult = -6) then      { Si hay problemas con la EMS }
  begin
    WriteLn('Error con la EMS.  Usando el disco...');
    Delay(1000);                { Avisamos y seguimos }
  end
  else
    if OvrResult <> 0 then        { Si hay algún error "serio" }
    begin
      WriteLn('Error en el sistema de overlays: ', OvrResult);
      Halt(1);
    end;
  repeat
    Proc1;                      { De la unidad 1 }
    Proc2;                      { De la unidad 2 }
    WriteLn;                    { Para que se lea mejor }
    delay(50);                  { Y esperamos un poco }
  until KeyPressed;             { Así hasta que se pulse una tecla }
  tecla := ReadKey;             { Absorbemos la tecla pulsada }
end.
```

Como último comentario: al buffer de solapamiento se le asigna el tamaño más pequeño que haga falta, y que coincide con el Overlay más grande. Si queremos un buffer mayor, para intentar que el programa haga menos lecturas de disco y sea más rápido, podemos leer el tamaño actual del buffer con **OvrGetBuf**, y cambiarlo con **OvrSetBuf**.

Curso de Pascal. Ampliación 5. Ensamblador desde Turbo Pascal.

El **ensamblador** es un lenguaje de muy bajo nivel, que nos permite el máximo control del procesador y la máxima velocidad de ejecución. Como inconveniente, es mucho más difícil de programar y depurar que los lenguajes de alto nivel como Pascal.

Para poder conseguir la máxima velocidad en los puntos críticos, sin necesidad de realizar todo el programa en ensamblador, la mayoría de los lenguajes actuales nos permiten incluir "trozos" de ensamblador en nuestros

programas.

Ya desde las primeras versiones de **Turbo Pascal** podíamos incluir código máquina "en linea", entre líneas en Pascal, con la orden **inline**: por ejemplo para imprimir la pantalla, la secuencia de instrucciones en ensamblador sería:

```
 PUSH BP    ( Salva en la pila el registro BP, para que no se modifique
)
 INT 5      ( Llama a la interrupcion 5, que imprime la pantalla )
 POP BP     ( Restaura el valor del registro BP )
```

Estas líneas, en código máquina (el ensamblador tiene una traducción casi directa) serían:

```
 PUSH BP   ->  $55
 INT 5     ->  $CD  $05
 POP BP    ->  $5D
```

Así que si introducimos esta secuencia de 4 bytes en un punto de nuestro programa, se imprimirá la pantalla. Entonces, nos basta con hacer:

```
procedure PrintScreen;
begin
  inline($55/$CD/$05/$5D);
end;
```

Desde el cuerpo de nuestro programa escribimos "PrintScreen" y ya está.

Un comentario sobre este sistema: imprime la pantalla a través del DOS, por lo que no habrá problema si es una pantalla de texto, pero puede que nos haga falta tener cargado GRAPHICS o algún dispositivo similar si es una pantalla en modo gráfico.

Desde la versión **6.0 de Turbo Pascal**, la cosa es aún más sencilla. Con "inline" conseguíamos poder introducir órdenes en código máquina, pero es un sistema engorroso: tenemos que ensamblar "a mano" o con la ayuda de algún programa como DEBUG, después debíamos copiar los bytes en nuestro programa en Pascal, y el resultado era muy poco legible. A partir de esta versión de TP, podemos emplear la orden "**asm**" para incluir ensamblador directamente:

```
procedure PrintScreen;
begin
  asm
    push bp
```

```
    int 5
    pop bp
  end
end;
```

Es decir, nos basta con encerrar entre **asm** y **end** la secuencia de órdenes en ensamblador que queramos dar. Si queremos escribir más de una orden en una línea, deberemos separarlas por punto y coma (;), siguiendo la sintaxis normal de Pascal, pero si escribimos cada orden en una línea, no es necesario, como se ve en el ejemplo. Los **comentarios** se deben escribir en el formato de Pascal: encerrados entre { y } ó (* y *).

Las **etiquetas** (para hacer saltos a un determinado punto de la rutina en ensamblador) pueden usar el formato de Pascal (tener cualquier nombre de identificador válido), y entonces tendremos que declararlas con **label**, o bien podemos emplear las llamadas "**etiquetas locales**", que no se pueden llamar desde fuera de la rutina en ensamblador, y que no hace falta declarar, pero su nombre debe empezar por **@.** Un ejemplo puede ser la versión en ensamblador de la rutina para sincronizar con el barrido de la pantalla que vimos en la ampliación 2 ("Gráficos sin BGI"):

```
procedure Retrace;
begin
asm
  mov  dx, $03da
 @ntrace:                  { Espera fin del barrido actual }
  in   al, dx
  test al, 8
  jnz  @ntrace
 @vtrace:                  { Espera a que comience el nuevo barrido }
  in   al, dx
  test al, 8
  jz   @vtrace
end;
end;
```

Como son etiquetas locales, a las que sólo vamos a saltar desde dentro de este mismo procedimiento, comenzamos su nombre con @ y no necesitamos declararlas.

Tenemos a nuestra disposición todas las ordenes de ensamblador del 8086. También podemos acceder a las del **80286** si usamos la directiva {$G+}, y/o

las del 8087 si empleamos {$N+}. Por ejemplo, para dibujar puntos en la pantalla en modo 320x200 de 256 colores podemos usar:

```
{$G+}
Procedure Putpixel (X,Y : Integer; Col : Byte);
Begin
  Asm
    mov     ax,$A000
    mov     es,ax
    mov     bx,[X]
    mov     dx,[Y]
    mov     di,bx
    mov     bx, dx                  { bx = dx }
    shl     dx, 8                   { dx = dx * 256 }
    shl     bx, 6                   { bx = bx * 64 }
    add     dx, bx                  { dx = dx + bx (= y*320) }
    add     di, dx                  { Posición final }
    mov     al, [Col]
    stosb
  End;
End;
```

Es decir: para multiplicar por 320, no usamos las instrucciones de multiplicacion, que son lentas, sino las de desplazamiento, de modo que al desplazar 8 posiciones estamos multiplicando por 256, al desplazar 6 multiplicamos por 64, y como 256+64=320, ya hemos hallado la fila en la que debemos escribir el punto. El {$G+} lo hemos usado porque instrucciones como "shl dx, 8" sólo están disponibles en los 286 y superiores; en un 8086 deberíamos haber escrito 8 instrucciones "shl dx,1" o haber usado el registro CL.

Podemos hacer una optimización: en todos los procedimientos que hemos visto, todo era ensamblador. Esto no tiene por qué ocurrir así: podemos tener Pascal y ensamblador mezclados en un mismo procedimiento o función. Pero cuando sea sólo ensamblador podemos emplear la directiva **assembler**, que permite al compilador de Turbo Pascal hacer una serie de optimizaciones cuando genera el código. El formato de un procedimiento que emplee esta directiva es:

```
procedure Modo320; assembler;
asm
  mov ax,$13
  int $10
end;
```

(debemos indicar "assembler;" después de la cabecera, y no hacen falta el "begin" y el "end" del procedimiento).

Como ejemplo de todo esto, un programita que emplea estos procedimientos para dibujar unas líneas en pantalla, esperando al barrido antes de dibujar cada punto (al final de este tema hay otro ejemplo más):

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Dibujo de puntos en   }
{    pantalla con ensam-   }
{    blador                }
{    GRAFASM.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program GrafAsm;
{$G+}

uses crt;

procedure Modo320; assembler;
asm
  mov ax,$13
  int $10
end;

procedure ModoTxt; assembler;
asm
  mov ax,3
  int $10
end;

Procedure Putpixel (X,Y : Integer; Col : Byte); assembler;
  Asm
    mov     ax,$A000
    mov     es,ax
    mov     bx,[X]
    mov     dx,[Y]
    mov     di,bx
    mov     bx, dx                  { bx = dx }
    shl     dx, 8                   { dx = dx * 256 }
    shl     bx, 6                   { bx = bx * 64 }
    add     dx, bx                  { dx = dx + bx (= y*320)  }
    add     di, dx                  { Posición final }
    mov     al, [Col]
    stosb
end;

procedure Retrace;
begin
```

```
asm
  mov  dx, $03da
 @ntrace:                    { Espera fin del barrido actual }
  in   al, dx
  test al, 8
  jnz  @ntrace
 @vtrace:                    { Espera a que comience el nuevo barrido }
  in   al, dx
  test al, 8
  jz   @vtrace
end;
end;

var i, j: integer;

begin
  Modo320;
  for i := 0 to 40 do
    for j := 1 to 200 do
      begin
      PutPixel(j+i*3,j,j);
      retrace;
      end;
  readkey;
  ModoTxt;
end.
```

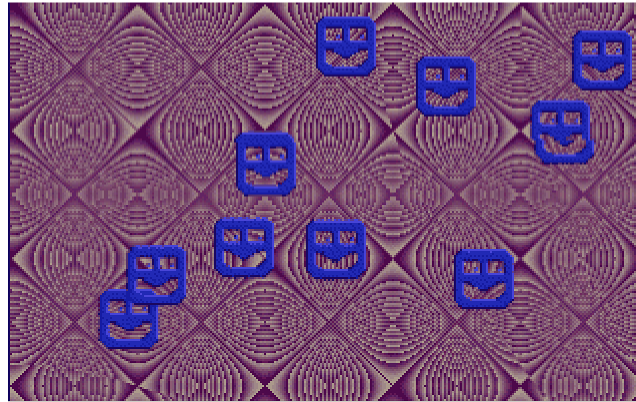
Finalmente, también tenemos la posibilidad de usar un ensamblador **externo**, como Turbo Assembler (TASM, de Borland), o MASM, de Microsoft. Estos programas crean primero un fichero objeto (con extensión OBJ), antes de enlazar con el resto de módulos (si los hubiera) y dar lugar al programa ejecutable.

Pues nosotros podemos integrar ese OBJ en nuestro programa en Pascal usando la directiva {$L}, y declarando como "**external**" el procedimiento o procedimientos que hallamos realizado en ensamblador, así:

```
procedure SetMode(Mode: Word); external;
{$L MODE.OBJ}
```

Finalmente, otro ejemplo más elaborado. Se trata de "**sprites**", figuras transparentes que se mueven por la pantalla. Con "transparentes" me refiero a que si hay algún hueco, debe verse el fondo a través suyo. Esto es totalmente imprescindible en los videojuegos: por ejemplo, mientras que anda nuestro personaje, tiene que verse el fondo entre sus piernas o junto a su cabeza en vez de un fondo negro.

Está tomado de una práctica que hice para una asignatura de la Universidad, en la que teníamos que manejar la pantalla VGA en modo gráfico 320x200x256. Como el lenguaje era libre, empleé Pascal, que es mi favorito, y como había que conseguir el menor tamaño posible (en el ejecutable) y una cierta rapidez, incluí bastantes cosas en ensamblador. Este es el resultado...



```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Dibujo de "sprites"   }
{    (imágenes transpa-    }
{    rentes) en pantalla   }
{    NSPRITE.PAS           }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}
program nSprite;
{$G+ Dibuja Sprites en Pantalla }
const
  segVideo: word = $a000;
  NumSprites = 10;            { Número de sprites }
  xSize = 30; ySize = 30;     { Tamaño de cada uno }
type
  tipoSprite = array[1..xSize, 1..ySize] of byte;   { El sprite en sí
}
const
  sprite : tipoSprite =
   ((0,0,0,2,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,0,0,0),
    (0,0,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,0,0),
    (0,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,0),
    (2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3),
    (2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3),
    (2,1,1,1,1,2,2,2,2,2,2,2,2,1,1,1,1,2,2,2,2,2,2,2,2,1,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,2,1,1,3,0,0,0,0,0,0,0,0,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,2,1,1,3,0,0,0,0,0,0,0,0,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,2,1,1,3,0,0,0,0,0,0,0,0,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,2,3,0,2,1,1,3,0,0,0,0,0,2,3,0,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,3,3,0,2,1,1,3,0,0,0,0,0,3,3,0,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,2,1,1,3,0,0,0,0,0,0,0,0,2,1,1,1,3),
    (2,1,1,1,1,3,3,3,3,3,3,3,3,1,1,1,1,3,3,3,3,3,3,3,3,1,1,1,1,3),
```

```
    (2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3),
    (2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3),
    (2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3),
    (2,1,1,1,1,2,2,2,2,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,2,1,1,1,1,1,1,1,1,1,1,1,1,2,0,0,0,0,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,2,1,1,1,1,1,1,1,1,2,0,0,0,0,0,0,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,2,1,1,1,1,2,0,0,0,0,0,0,0,0,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,2,2,2,2,0,0,0,0,0,0,0,0,0,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1,1,1,3),
    (2,1,1,1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1,1,1,1,3),
    (2,1,1,1,1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1,1,1,1,1,3),
    (2,1,1,1,1,1,1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,1,1,1,1,1,1,3),
    (2,1,1,1,1,1,1,1,1,3,3,3,3,3,3,3,3,3,3,3,3,1,1,1,1,1,1,1,1,2),
    (0,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,0),
    (0,0,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,0,0),
    (0,0,0,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,0,0,0),
    (0,0,0,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0)
  );
type SprDat= record
  x, y: word;            { Coordenadas de cada sprite }
  vx, vy: shortint;      { Velocidad (incremento) según cada eje }
end;

{ --- Componentes R,G,B para un color --- }
procedure setpal(col,r,g,b:byte); assembler;
asm
  mov dx,03c8h;
  mov al,col;
  out dx,al;
  inc dx;
  mov al,r; out dx,al;
  mov al,g; out dx,al;
  mov al,b; out dx,al;
end;

{ --- Copia 64000 bytes de una pantalla virutal a otra (física) --- }
procedure copia(org,dst:word); assembler;
asm
  push ds;
  mov ds,[org];
  xor si,si;
  mov es,[dst]
  xor di,di;
  mov cx,320*200/2;
  rep movsw;
  pop ds;
end;

{ --- Borra 64000 bytes en una zona de memoria --- }
procedure cls(dst:word); assembler;
asm
  mov es,[dst];
  xor di,di;
  xor ax,ax;
  mov cx,320*200/2;
  rep stosw;
end;

{ --- Sincroniza con el barrido de la pantalla --- }
procedure retrace; assembler; asm
  mov dx,03dah;
```

```
@vert1: in al,dx;
  test al,8;
  jnz @vert1
@vert2: in al,dx;
  test al,8;
  jz @vert2;
  end;

{ --- Escribe un sprite en la pantalla --- }
procedure putsprite(x,y,sprseg, sprofs,virseg:word); assembler;
asm
  push ds
  mov ds,sprseg;
  mov si,sprofs            { Segmento y desplaz. del sprite }
  mov es,virseg; xor di,di { Segmento y desp. de la pantalla virtual
}
  mov ax,[y];              { Numero de fila }
  shl ax,6;
  mov di,ax;
  shl ax,2;
  add di,ax                { Fila * 320 }
  add di,[x]               { Fila * 320 + columna }
  mov dx,320-xsize         { Pixels restantes en la línea }
  mov bx,ysize             { Altura del dibujo }
 @l1:
  mov cx,xsize             { Anchura de cada fila }
 @l0:
  lodsb;                   { Leo un byte del sibujo }
  or al,al;
  jz @noDib                { Si es 0 (transp.), lo salto }
  mov [es:di],al           { Si no, lo dibujo }
 @noDib:
  inc di;                  { Siguiente pixel }
  dec cx;                  { Queda uno menos por dibujar en la fila }
  jnz @l0;                 { Si aun quedan, repito }
  add di,dx;               { Si no quedan, voy al principio de la
sgte fila }
  dec bx;                  { Queda una fila menos }
  jnz @l1;                 { Repito hasta que se acaben las filas }
  pop ds
end;

{ --- Variables que usaré en el cuerpo --- }
var
  PantVirt: pointer;   { La pantalla virtual }
  SegVirt: word;       { Segmento donde se encuentra }
  Fondo: pointer;      { La pantalla de fondo }
  SegFon: word;        { y del fondo }
  D:                   { Datos de cada sprite }
    array[1..numSprites] of SprDat;
  i,j: integer;        { Bucles }
  label bucle;

{---------------------------- }
{ --- Cuerpo del programa --- }
{---------------------------- }
begin
  asm mov ax,13h; int 10h; end;  { Cambio a modo 320x200x256 }
  randomize;                     { Números aleatorios }
  getmem(PantVirt,320*200);      { Reservo y vacío pantalla virtual }
    SegVirt := seg(PantVirt^);
```

```
    cls(SegVirt);
  getmem(Fondo,320*200);          { Y la de fondo }
    SegFon:=seg(Fondo^);
    cls(SegFon);
  for i := 1 to NumSprites do     { Datos aleatorios de los Sprites }
    with d[i] do
      begin
      x := random (219-xSize)+50;
      y := random (99-ySize)+50;
      repeat
        vx := random(6) - 3;
      until vx<>0;
      repeat
        vy := random(6) - 3;
      until vy<>0;
      end;
  for i:=1 to 128 do              { Paleta de colores del fondo }
    setpal(127+i,20+i div 5,i div 3,20+i div 7);
  SetPal(1,10,10,45);
  SetPal(2,0,0,25);
  SetPal(3,20,20,60);

  for i:=0 to 319 do              { Dibujo el patrón de fondo }
    for j:=0 to 199 do
      mem[SegFon:j*320+i]:=128+abs(i*i-j*j) and 127;
 bucle:
    copia(SegFon,SegVirt);        { Copio el fondo en la pantalla
virtual }
    for i := 1 to numSprites do   { Dibujo los sprites }
      begin
      PutSprite( d[i].x, d[i].y, seg(Sprite), ofs(Sprite), SegVirt);
      inc(d[i].x,d[i].vx);            { Actualizo las coordenadas }
      if (d[i].x < 5) or (d[i].x > (315-xSize)) then
        d[i].vx := -d[i].vx;
      inc(d[i].y,d[i].vy);
      if (d[i].y < 5) or (d[i].y > (195-ySize)) then
        d[i].vy := -d[i].vy;
      end;
    retrace;                      { Sincronizo con el barrido }
    copia(SegVirt,SegVideo);      { Y copio la pantalla virtual en la
visible }
  { Repito hasta que se pulse una tecla }
  asm  mov ah, 1;  int 16h;  jz bucle; end;
  { Absorbo esa pulsación de tecla }
  asm  mov ah, 0;  int 16h;  end;
  freemem(PantVirt,320*200);      { Libero la memoria reservada }
  freemem(Fondo,320*200);
  asm mov ax,3; int 10h; end;     { Vuelvo a modo texto }
 end.
```

Este programa está comprobado con Turbo Pascal 7.0. Eso sí, como reserva dos pantallas virtuales puede que no quede memoria suficiente para ejecutarlo desde el IDE normal (TURBO.EXE). Entonces habría que usar el compilador de línea de comandos (TPC.EXE) o el de modo protegido (TPX.EXE).

Curso de Pascal. Ampliación 6. Directivas del compilador (Turbo Pascal).

Lo que vamos a ver a continuación no son órdenes del lenguaje Pascal, sino órdenes directas al compilador. Son exclusivas de Turbo Pascal (aunque cada compilador suele tener las suyas propias, y es posible que coincidan). Estas órdenes nos permitirán indicarle al compilador cosas como:

- Que incluya otro fichero justo en esa posición.
- Que genere código para 286 o superior.
- La cantidad de memoria (heap y pila) que queremos reservar.
- Que no compile ciertas partes del programa.
- [...]

Dividiremos las directivas en **tres tipos**:

- Switches, que activan o desactivan una posibilidad.
- Parametrizadas, que detallan características.
- Condicionales.

Algunas de estas directivas ya las hemos mencionado en lecciones del curso. Ahora iremos viendo cada uno de estos tipos por orden alfabético. Las que voy a tratar son las de Turbo Pascal 7.0 para DOS; en versiones anteriores pueden no existir todas, y en versiones posteriores (p . Borland Pascal 7.0, que compila también para DPMI y Windows) puede haber más.

Algunas directivas pueden activar como "**switches**" (interruptores), activando o desactivando ciertas características. Estas son:

$A: Alineación de datos.
$B: Modo de evaluación Booleana.
$D: Información para depuración.
$F: Forzar llamadas lejanas (Far).
$G: Generar código para 80286.
$I: Comprobación de Entrada/salida.
$L: Información de símbolos locales.
$N: Coprocesador numérico 80x87 (y $E emulación).
$P: Parámetros String abiertos.
$Q: Comprobación de overflow.
$R: Comprobación de rango.
$S: Comprobación de desbordamiento de pila.
$T: Comprobación de punteros con tipo.

$V: Comprobación de Strings.
$X: Sintaxis extendida.

$A: Alineación de datos.

Esta opción intercambia entre el alineamiento de byte y de doble byte para las variables.

Cuando se indica {$A+}, que es el valor por defecto, las variables de más de un byte de tamaño se colocan de modo que empiecen en una dirección de memoria par.

¿Para qué es esto? Para conseguir mayor velocidad: en las CPUs 8086 y superiores (no en el 8088), se accede con más rapidez a las direcciones pares de memoria (un ciclo de reloj frente a dos).

Con {$A-} las variables no se alinean de ninguna forma, se colocan en la próxima posición libre de memoria, sea par o impar.

$B: Modo de evaluación Booleana.

Si se indica {$B+}, cuando haya dos comparaciones booleanas enlazadas por AND o por OR, se evalúa la expresión completa.

Si se usa {$B-}, que es el valor por defecto, la evaluación puede terminar antes si se conoce cual va a ser el resultado. Por ejemplo, si una orden se debe ejecutar si ocurre una cosa Y otra (se deben dar dos condiciones simultaneas), y el compilador no se molesta en mirar la segunda si ya ve que la primera es falsa.

$D: Información para depuración.

La opción {$D+} guarda junto con el programa ejecutable una información adicional que nos permite hacerla funcionar paso a paso, a la vez que en pantalla se nos muestra la línea que se está ejecutando. Esto nos ayuda a detectar errores con más facilidad.

Por defecto, esta información se guarda automáticamente. Si queremos evitarlo, deberemos usar {$D-}.

Esta opción se suele usar junto con la de información sobre símbolos locales $L.

$F: Forzar llamadas lejanas (Far).

Afecta a la generación de código máquina:

Si se indica {$F+}, las llamadas a los procedimientos y funciones restantes se haran considerándolas como "far", es decir, suponiendo que se encuentran en otro segmento de 64K. Esto es necesario cuando se emplean Overlays, por ejemplo.

El valor por defecto es {$F-}, en el que se considera que todas las procedimientos y funciones se encuentran en el mismo segmento, y se usan llamadas "cercanas" (near).

Cuando los procedimientos o funciones se encuentren en otra "unit", irán a un segmento distinto, pero no hace falta indicárselo al compilador, que lo tiene en cuenta automáticamente.

$G: Generar código para 80286.

El valor por defecto es {$G-}, que hace que sólo se empleen órdenes genéricas del 8086.

Con {$G+} permitimos al compilador que utilice algunas órdenes del 80286 para optimizar la generación de código, a cambio de que el programa no funcione en equipos XT.

$I: Comprobación de Entrada/salida.

Si está activada con {$I+}, que es el valor por defecto, y durante la ejecución del programa aparece algún error de entrada/salida (I/O), se interrumpe el programa y se muestra un mensaje de error.

Si la desactivamos con {$I-}, el programa no se aborta, sino que se devuelve un código de error en la variable IOResult, que nosotros debemos comprobar.

El uso más habitual es para comprobar si un fichero existe antes de intentar acceder a él, aunque también se puede usar para cosas más "sofisticadas", como comprobar si en un "readln" que esperaba un valor numérico hemos introducido otra cosa.

$L: Información de símbolos locales.

Hace que el compilador guarde información sobre las variables de un programa o unidad, para que podamos modificarlas en media sesión de depuración. Por defecto, está activado ({$L+}).

Se suele usar en conjunción con $D.

$N: Coprocesador numérico 80x87.

Si se habilita con $N+, las operaciones con números reales se realizarán mediante el coprocesador matemático. Estos además nos permite usar cuatro nuevos tipos numéricos: Single, Double, Extended, y Comp.

Si está deshabilitado (valor por defecto o con {$N-}), las operaciones con números reales se realizan por software, y no están disponibles estos 4 tipos.

Si queremos usarlos pero no tenemos coprocesador, existe la posibilidad de "emularlo" mediante software con la directiva {$E+}.

El uso habitual es {$N-,E-} en condiciones normales, o {$N+,E+} para usar estos tipos ampliados. En este último caso, se emplea el coprocesador si existe, o se emula automáticamente si no es así.

$P: Parámetros String abiertos.

Con $P+, todos los parámetros de tipo String se consideran string abiertos (OpenString, ver $V).

Con {$P-} se tratan como en versiones anteriores de TP.

$Q: Comprobación de overflow.

Si está habilitado, se comprueba que no haya overflow (desbordamiento de la capacidad de un cierto tipo de datos) en el resultado de alguna de estas operaciones:

  +, -, * Abs, Sqr, Succ, Pred

Por ejemplo, si en una variable de tipo byte intentamos almacenar 200+200, que supera el valor máximo de 255, el programa se abortaría con un mensaje de error.

Usar {$Q+} hace el programa más grande y lento, así que conviene emplearlo sólo mientras se esté depurando, y no conservarlo en la versión definitiva.

$R: Comprobación de rango.

Comprueba que el valor asignado a una variable (o al índice de un array o un string) esté dentro del rango correcto.

Al igual que ocurre con $Q, conviene usarlo sólo mientras se esté depurando el programa.

$S: Comprobación de desbordamiento de pila.

Comprueba si la pila se desborda, y si es así, interrumpe el programa.

Como en la pila se almacenan las direcciones de retorno cuando se llama a una función o procedimiento, el desbordar la capacidad de la pila puede suponer que se lea un dato erróneo y se salte a una dirección indeterminada de la memoria, lo que puede ser muy peligroso.

El valor por defecto es {$S+} (sí se comprueba).

$T: Comprobación de punteros con tipo.

El operador @ devuelve la dirección de una variable (por ejemplo), por lo que se puede usar para crear variable que sean referencias a otras.

Si empleamos {$T-} (por defecto), @ siempre devolverá un puntero sin tipo, pero con {$T+} devolverá un puntero al tipo de la variable que se trate, por lo que sólo será compatible con punteros del mismo tipo.

$V: Comprobación de Strings.

Si se indica {$V-}, se podrá pasar como a una función o un procedimiento como parámetro un string de tamaño distinto al esperado, sin que el compilador proteste.

Con {$V+} (valor por defecto en TP7) no se puede. Por ejemplo: si una función que hemos definido espera como parámetro un string[4] y le intentamos pasar una variable que hemos definido como string[2], el compilador nos dirá que son de distinto tipo.

En TP7 se recomienda usar la opción de "strings abiertos" (open strings), $P en vez de ésta.

$X: Sintaxis extendida.

Con sintaxis extendida ($X+, valor por defecto), se puede llamar a funciones como si fuesen procedimientos, despreciando los valores que devuelven.

Se puede ver un ejemplo en la lección sobre Turbo Vision, en el segundo apartado: MessageBox es realmente una función, que devuelve un código que nos informa sobre el botón que se ha pulsado; si no nos interesa este valor (como en el ejemplo, en el que sólo hay un botón), podemos llamar a la función como si fuese un procedimiento.

Las directivas **parametrizadas** son aquellas que incluyen una serie de parámetros, y que por ello son más flexibles que las anteriores, que sólo podian habilitar o desabilitar una función.

$I xxx: Incluir fichero.
$L xxx: Enlazar fichero objeto.
$M xxx: Tamaños de memoria.

$I xxx: Incluir fichero.

Incluye otro fichero fuente en esa posición. Si no se indica la extensión, se considerará que es .PAS.

Hoy en día, con la posibilidad de utilizar unidades, o incluso fuentes de gran tamaño (hasta 1 Mb en el editor TPX de TP7), no resulta necesaria.

$L xxx: Enlazar fichero objeto.

Incluye un fichero objeto (.OBJ, creado con ensamblador, por ejemplo) en esa posición.

$M xxx: Tamaños de memoria.

El formato es {$M Pila, HeapMin, HeapMax}

Por ejemplo, si queremos dejar 2K para la pila, y que el Heap (espacio de memoria para variables dinámicas) pueda estar entre 8 y 640K sería:

{$M 8192,8192,655360}

Las directivas **condicionales** son las que hacen que una parte del programa se compile sólo en ciertas circunstancias: si se ha especificado una cierta opción (directiva de tipo "switch") o si se ha definido una constante simbólica.

```
{$IFOPT}          Compila lo que sigue si se ha definido un cierto
                  switch.  Por ejemplo:

                  {$IFOPT N+}
                     var x: double;
                  {$ELSE}
                     var x: real;
                  {$ENDIF}

{$DEFINE Name}    Define una constante simbólica
{$UNDEF Name}     Cancela un definición
{$IFDEF Name}     Compila el código que sigue si el nombre se definió
{$IFNDEF}         Compila si no se definió
{$ELSE}           En caso contrario (ver ejemplo anterior)
{$ENDIF}          Fin de compilación condicional (ver ejemplo anterior)
```

Un ejemplo de su uso sería

```
{$DEFINE depurando}
[...]
{$IFDEF depurando}
    writeln('Ahora x vale', x );
{$ENDIF}
```

Una vez que el programa funcione correctamente, eliminamos la línea del $DEFINE (borrándola o con $UNDEF), y todas las partes que habíamos añadido para depurar entre $IFDEF y $ENDIF quedarán automáticamente sin compilar.

Curso de Pascal. Fuentes de ejemplo - Rotaciones 3D.

En el tema de "Gráficos sin BGI" hemos visto por encima cómo hacer rotaciones en el plano.

Como eso de los gráficos es una de las cosas más vistosas que se pueden hacer con un ordenador, especialmente cuando se trata de 3 dimensiones, vamos a profundizar un poco más, y a poner un par de ejemplos.

Las rotaciones son sencillas cuando se tiene una cierta base de álgebra de matrices, y no tanto si no es el caso. De cualquier modo, podemos usar las formulitas "tal cual", sin saber cómo trabajan.

Para girar en torno al **eje X**, la matriz de rotación es:

```
 ¦ 1    0    0  ¦      donde sx es el seno del ángulo que se rota
 ¦ 0    cx   sx ¦      y cx es su coseno.
 ¦ 0   -sx   cx ¦
```

Si esto lo convertimos a formulitas

```
 x = x
 y = (y * cx) - (z * sx)
 z = (y * sx) + (z * cx)
```

De forma similar, en torno al **eje Y** tenemos:

```
 ¦ cy   0   -sy ¦      igualmente, sy y cy son seno y coseno del
 ¦ 0    1    0  ¦      ángulo girado
 ¦ sy   0    cy ¦
```

que queda como

```
 x = (x * cy) + (z * sx)
 y = y
 z = (z * cy) - (x * sy)
```

Y alrededor del **eje Z**:

```
 ¦  cz   sz   0 ¦      cz y sz son... ¡ lo de siempre !
 ¦ -sz   cz   0 ¦
 ¦  0    0    1 ¦
```

que queda como

```
 x = (x * cz) - (y * sz)
 y = (x * sz) + (y * cz)
 z = z
```

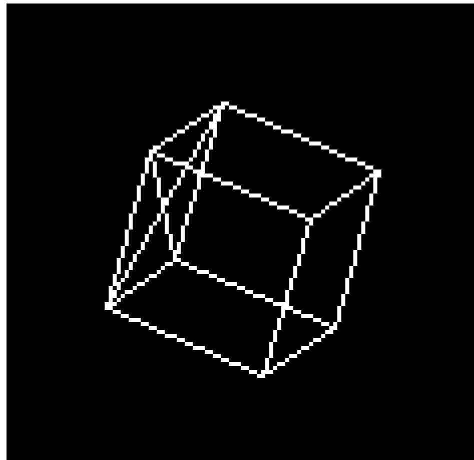
(esta última es la rotación en el plano que habíamos visto).

Hay autores que usan estos 3 grupos de fórmulas de forma independiente, y hay quien prefiere multiplicar las tres matrices para obtener la que sería la "matriz de giro" de un punto cualquiera, que queda algo así

```
¦  (cz*cy)+(sz*sx*sy)   (cy*-sz)+(cz*sx*sy)  (cx*sy) ¦
¦                                                    ¦
¦  (sz*cx)              (cz*cx)              (-sx)   ¦
¦                                                    ¦
¦  (-sy*cz)+(sz*sx*cy)  (sz*sy)+(cz*sx*cy)   (cx*cy) ¦
```

En cualquier caso, vamos a dejarnos de rollos y a ver un par de **ejemplos** de aplicación de esto.

El primero está basado en un fuente de Peter M. Gruhn, que es muy fácil de seguir, porque la parte encargada de las rotaciones sigue claramente los 3 grupos de fórmulas anteriores. El resto es la definición de la figura, las rutinas para dibujar un punto o una línea (que ya hemos visto) y poco más. Allá va:



```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Rotar un cubo en 3D   }
{    ROTACUBO.PAS          }
{                          }
{  Este fuente procede de  }
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

program RotaCubo;
{$G+}
{
  Basado en un fuente de Dominio Público, por
  Peter M. Gruhn 1993

  El original se puede encontrar en los SWAG
```

```
  Modificaciones por Nacho Cabanes, 1996:
  - Modo 320x200x256, sin BGI (usa algoritmo de Bresenham para dibujar
    líneas y sincroniza con el barrido de la VGA).
  - Emplea algo de ensamblador (ver Ampliación 5)
  - El cubo se mueve sólo.


Posibles mejoras (muchas, sólo pongo algunas como ejemplo):
  - Emplear aritmética entera, y tablas de senos y cosenos para
    mayor velocidad, aunque en este caso no es necesario, porque
    se rotan muy pocos puntos.
  - Que las rotaciones no sean aditivas (se vuelva a rotar a partir
    del original, no de la figura ya rotada, para que los errores no
    se vayan sumando).
  - Más flexibilidad: definir las líneas a partir de sus dos vértices
    en vez de crear las figuras "a pelo".
  - Definir caras 3D para dibujar figuras sólidas.
}

uses
  crt;

const
  gradRad = 1 {grados} * 3.1415926535 {radianes} / 180 {por grado};
  { Convierte un grado a radianes (sin y cos usan radianes) }

type
  punto = record   { Punto en 3d }
    x, y, z : real;
  end;

var
  img : array [0..7] of punto;   { Nuestra imagen tendrá 8 puntos }
  tecla: char;
  color: byte;


procedure retrace; assembler;   { Espera el barrido de la pantalla }
asm
  mov dx,3dah
 @vert1:
  in al,dx
  test al,8
  jz @vert1
 @vert2:
  in al,dx
  test al,8
  jnz @vert2
end;


procedure init;     { Inicializa }
begin
  asm
    mov ax, $13     { Modo 320x200x256 }
    int $10
  end;
  { Datos de la imagen }
  img[0].x := -35;  img[0].y := -35;  img[0].z := -35;
  img[1].x := 35;   img[1].y := -35;  img[1].z := -35;
  img[2].x := 35;   img[2].y := 35;   img[2].z := -35;
```

```
  img[3].x := -35;  img[3].y := 35;   img[3].z := -35;
  img[4].x := -35;  img[4].y := -35;  img[4].z := 35;
  img[5].x := 35;   img[5].y := -35;  img[5].z := 35;
  img[6].x := 35;   img[6].y := 35;   img[6].z := 35;
  img[7].x := -35;  img[7].y := 35;   img[7].z := 35;
end;


Procedure Ponpixel (X,Y : Integer; Col : Byte); assembler;
  { Dibuja un punto en la pantalla gráfica, en 320x200x256 }
  Asm
    mov     ax,$A000
    mov     es,ax
    mov     bx,[X]
    mov     dx,[Y]
    mov     di,bx
    mov     bx, dx                  { bx = dx }
    shl     dx, 8                   { dx = dx * 256 }
    shl     bx, 6                   { bx = bx * 64 }
    add     dx, bx                  { dx = dx + bx (= y*320) }
    add     di, dx                  { Posición final }
    mov     al, [Col]
    stosb
end;


procedure LineaB(x, y, x2, y2 : word; color: byte);
{ Dibuja una línea, basado en el algoritmo de Bresenham }
{ Original de Sean Palmer; una pequeña corrección por Nacho Cabanes }
var
 d,
 dx, dy,             { Salto total según x e y }
 ai, bi,
 xi, yi              { Incrementos: +1 ó -1, según se recorra }
        : integer;
begin
 if (x=x2) and (y=y2) then  { Corrige un fallo: si es un sólo punto }
   begin                    { el algoritmo (tal y como era) falla }
   PonPixel(x,y,color);
   exit;
   end;
 if (x < x2) then    { Si las componentes X están ordenadas }
 begin
   xi := 1;          { Incremento +1 }
   dx := x2 - x;     { Espacio total en x }
 end
 else                { Si no están ordenadas }
 begin
   xi := - 1;        { Increm. -1 (hacia atrás) }
   dx := x - x2;     { y salto al revés (negativo) }
 end;
 if (y < y2) then    { Análogo para las componentes Y }
 begin
   yi := 1;
   dy := y2 - y;
 end
 else
 begin
   yi := - 1;
   dy := y - y2;
 end;
```

```
 PonPixel(x, y,color);   { Dibujamos el primer punto }
 if dx > dy then         { Si hay más salto según x que según y }
 begin                   { (recta más cerca de la horizontal) }
   ai := (dy - dx) * 2;   { Variables auxiliares del algoritmo }
   bi := dy * 2;          { ai y bi no varían; d comprueba cuando }
   d  := bi - dx;         { debe cambiar la coordenada y }
   repeat
     if (d >= 0) then     { Comprueba si hay que avanzar según y }
     begin
       y := y + yi;       { Incrementamos Y (+1 ó -1) }
       d := d + ai;       { y la variable de control }
     end
     else
       d := d + bi;       { Si no varía y, d sí lo hace según bi }
     x := x + xi;         { Incrementamos X como corresponda }
     PonPixel(x, y, color);          { Dibujamos el punto }
   until (x = x2);           { Se repite hasta alcanzar el final }
 end
 else                        { Si hay más salto según y que según x }
 begin                       { (más vertical), todo similar }
   ai := (dx - dy) * 2;
   bi := dx * 2;
   d  := bi - dy;
   repeat
     if (d >= 0) then
     begin
       x := x + xi;
       d := d + ai;
     end
     else
       d := d + bi;
     y := y + yi;
     PonPixel(x, y, color);
   until (y = y2);
 end;
end;



procedure linea(x1, y1, z1, x2, y2, z2 : real);
{ Convierte las coordenadas de real a entero y muestra centrado en
  pantalla.  La coordenada Z se desprecia en este ejemplo, pero se
podría
  usar para dar una mayor sensación de perspectiva (cónica en vez de
  cilíndrica. }
begin
  lineaB(round(x1) + 160, round(y1) + 100, round(x2) + 160, round(y2)
+ 100,
    color);
end;


procedure dibujaImg;
{ Dibuja la imagen (ésta en concreto -> poco versátil ) }
begin
  linea(img[0].x, img[0].y, img[0].z, img[1].x, img[1].y, img[1].z);
  linea(img[1].x, img[1].y, img[1].z, img[2].x, img[2].y, img[2].z);
  linea(img[2].x, img[2].y, img[2].z, img[3].x, img[3].y, img[3].z);
  linea(img[3].x, img[3].y, img[3].z, img[0].x, img[0].y, img[0].z);

  linea(img[4].x, img[4].y, img[4].z, img[5].x, img[5].y, img[5].z);
```

```
  linea(img[5].x, img[5].y, img[5].z, img[6].x, img[6].y, img[6].z);
  linea(img[6].x, img[6].y, img[6].z, img[7].x, img[7].y, img[7].z);
  linea(img[7].x, img[7].y, img[7].z, img[4].x, img[4].y, img[4].z);

  linea(img[0].x, img[0].y, img[0].z, img[4].x, img[4].y, img[4].z);
  linea(img[1].x, img[1].y, img[1].z, img[5].x, img[5].y, img[5].z);
  linea(img[2].x, img[2].y, img[2].z, img[6].x, img[6].y, img[6].z);
  linea(img[3].x, img[3].y, img[3].z, img[7].x, img[7].y, img[7].z);

  linea(img[0].x, img[0].y, img[0].z, img[5].x, img[5].y, img[5].z);
  linea(img[1].x, img[1].y, img[1].z, img[4].x, img[4].y, img[4].z);
end;


procedure rotx;
{ Rotación en torno al eje X.  Un poco de álgebra lineal... }
var
  i : integer;
begin
  color := 0;
  dibujaImg;
  for i := 0 to 7 do
  begin
    img[i].x :=  img[i].x;
    img[i].y :=  img[i].y * cos(gradRad) + img[i].z * sin(gradRad);
    img[i].z := -img[i].y * sin(gradRad) + img[i].z * cos(gradRad);
  end;
  color := 15;
  dibujaImg;
end;


procedure roty;
{ Rotación en torno al eje Y }
var
  i : integer;
begin
  color := 0;
  dibujaImg;
  for i := 0 to 7 do
  begin
    img[i].x := img[i].x * cos(gradRad) - img[i].z * sin(gradRad);
    img[i].y := img[i].y;
    img[i].z := img[i].x * sin(gradRad) + img[i].z * cos(gradRad);
  end;
  color := 15;
  dibujaImg;
end;


procedure rotz;
{ Rotación en torno al eje Z }
var
  i : integer;
begin
  color := 0;
  dibujaImg;
  for i := 0 to 7 do
  begin
    img[i].x :=  img[i].x * cos(gradRad) + img[i].y * sin(gradRad);
    img[i].y := -img[i].x * sin(gradRad) + img[i].y * cos(gradRad);
```

```
    img[i].z :=  img[i].z;
  end;
  color := 15;
  dibujaImg;
end;


begin
  init;              { Inicializar }
  repeat
    retrace; rotx;   { Rotar y dibujar }
    retrace; roty;
    retrace; rotz;
  until (keypressed) { Hasta pulsar ESC }
    and (readkey = #27);
  asm
    mov ax, 3        { Modo texto }
    int $10
  end;
end.
```

Ahora vamos a ver otro ejemplo bastante más elaborado. Este está basado en un fuente de Bas van Gaalen.

- Rota una figura más complicada, que además es sólida (no deben verse las caras ocultas).
- Tiene un fondo que no se modifica.
- Al proyectar a 2 dimensiones da una cierta impresión de profundidad.
- Las caras están sombreadas. Aun así, este sombreado no es el más correcto, porque está basado en la distancia de cada plano al observador, de modo que dos planos contiguos se verán distintos (como pasa en la E que he cogido como ejemplo). El resultado sería más vistoso si el sombreado se basase en la inclinación de las caras (vector normal a cada plano).

```
{--------------------------}
{  Ejemplo en Pascal:      }
{                          }
{    Rota una E sólida 3D  }
{    ROTAE.PAS             }
{                          }
{  Este fuente procede de  }
```

```
{  CUPAS, curso de Pascal  }
{  por Nacho Cabanes       }
{                          }
{  Comprobado con:         }
{    - Turbo Pascal 7.0    }
{--------------------------}

{--------------------------------------------------}
{  E rotada en 3D                                  }
{  Por Nacho Cabanes, 96                           }
{                                                  }
{  Basado (mucho) en 3DHEXASH, de Bas Van Gaalen }
{  (dominio público, recopilado en GFXFX)          }
{                                                  }
{  Modificaciones sobre el original:               }
{  - Comentado, para que sea más fácil de seguir }
{  - Traducido a español  :-)                      }
{  - Cambiadas sentencias Inline por Asm           }
{  - Añadido un fondo al dibujo                    }
{  - La figura ahora es una E  :-)                 }
{                                                  }
{  Otras posibles mejoras:                         }
{  - Sombreado en función de la dirección de       }
{    cada cara, no de su distancia.                }
{--------------------------------------------------}

program RotaE;
{$G+}

uses
  crt;
const
  divd=128;    { Para convertir de reales a enteros los senos/cosenos
}
  dist=200;    { Distancia del observador }
  segVideo:word=$a000;  { Segmento de video: VGA modo gráfico }

  NumPuntos = 23;       { Numero de puntos }
  NumPlanos = 19;       { Número de caras }

  { Ahora van los puntos en sí }
  punto:array[0..NumPuntos,0..2] of integer=(
    (-40, 40, 20),( 40, 40, 20),( 40, 27, 20),(-27, 27, 20),  { E
superior }
    (-27,  7, 20),( 27,  7, 20),( 27, -7, 20),(-27, -7, 20),
    (-27,-27, 20),( 40,-27, 20),( 40,-40, 20),(-40,-40, 20),
    (-40, 40,  0),( 40, 40,  0),( 40, 27,  0),(-27, 27,  0),  { E
inferior }
    (-27,  7,  0),( 27,  7,  0),( 27, -7,  0),(-27, -7,  0),
    (-27,-27,  0),( 40,-27,  0),( 40,-40,  0),(-40,-40,  0));
  { Y ahora los 4 puntos que forman cada plano }
  plano:array[0..NumPlanos,0..3] of byte=(
    (0,3,8,11),(0,1,2,3),(4,5,6,7),(8,9,10,11),             {
Superior }
    (12,15,20,23),(12,13,14,15),(16,17,18,19),(20,21,22,23),  {
Inferior }
    (1,2,14,13),(2,3,15,14),(3,4,16,15),(4,5,17,16),          {
Uniones }
    (6,7,19,18),(7,8,20,19),(8,9,21,20),(9,10,22,21),
    (10,11,23,22),(11,0,12,23),(0,1,13,12),(5,6,18,17)
    );
```

```
var
  { Coordenada "z" de cada plano, usada para sombrear: los más lejanos
    serán más oscuros }
  polyz:array[0..NumPlanos] of integer;
  pind:array[0..NumPlanos] of byte;
  { Tablas de senos y cosenos }
  ctab:array[0..255] of integer;
  stab:array[0..255] of integer;
  { La pantalla temporal en la que realmente se dibujará y el fondo }
  pantTemp, fondo:pointer;
  { Las direcciones en que empiezan ambos }
  segTemp, segFondo:word;
  { Límites de la pantalla, para no dibujar fuera }
  minx,miny,maxx,maxy:integer;

{ -------------------------------------------------------------------
------ }

procedure retrace; assembler; asm
  { Sincroniza con el barrido de la VGA }
  mov dx,3dah; @vert1: in al,dx; test al,8; jz @vert1
  @vert2: in al,dx; test al,8; jnz @vert2; end;

procedure copia(src,dst:word); assembler; asm
  { Copia 64K de una dirección de memoria a otra }
  push ds; mov ax,[dst]; mov es,ax; mov ax,[src]; mov ds,ax
  xor si,si; xor di,di; mov cx,320*200/2; rep movsw; pop ds; end;

procedure setpal(c,r,g,b:byte); assembler; asm
  { Cambia un color de la paleta: fija la cantidad de
  rojo, verde y azul }
  mov dx,3c8h; mov al,[c]; out dx,al; inc dx; mov al,[r]
  out dx,al; mov al,[g]; out dx,al; mov al,[b]; out dx,al; end;

function coseno(i:byte):integer; begin coseno:=ctab[i]; end;
function seno(i:byte):integer; begin seno:=stab[i]; end;
  { Seno y coseno, a partir de tablas para mayor velocidad }

{ -------------------------------------------------------------------
------ }

procedure horline(xb,xe,y:integer; c:byte); assembler;
{ Dibuja una línea horizontal a una cierta altura y con un color dado
}
asm
  mov bx,xb
  mov cx,xe
  cmp bx,cx
  jb @skip
  xchg bx,cx
 @skip:
  inc cx
  sub cx,bx
  mov es,segTemp
  mov ax,y
  shl ax,6
  mov di,ax
  shl ax,2
  add di,ax
  add di,bx
```

```
  mov al,c
  shr cx,1
  jnc @skip2
  stosb
 @skip2:
  mov ah,al
  rep stosw
 @out:
end;

function MaxI(A,B:Integer):Integer; assembler;
{ Valor máximo de 2 dados }
asm
  mov ax, a
  mov bx, b
  cmp ax,bx
  jg @maxax
  xchg ax, bx
@maxax:
end;


function MinI(A,B:Integer):Integer; assembler;
{ Valor mínimo de 2 dados }
asm
  mov ax, a
  mov bx, b
  cmp ax,bx
  jl @minax
  xchg ax, bx
@minax:
end;


function EnRango(valor,min,max:integer):integer; assembler;
{ Comprueba si un valor está entre dos dados }
asm
  mov ax, valor
  mov bx, min
  mov cx, max
  cmp ax,bx
  jg @maxAx
  xchg ax, bx
@maxAx:
  cmp ax,cx
  jl @minAx
  xchg ax, cx
@minAx:
end;


procedure polygon( x1,y1, x2,y2, x3,y3, x4,y4 :integer; c:byte);
{ Dibuja un polígono, dados sus 4 vértices y el color }
{ Este sí es el original de Bas van Gaalen intacto...  O:-)  }
var pos:array[0..199,0..1] of integer;
  xdiv1,xdiv2,xdiv3,xdiv4:integer;
  ydiv1,ydiv2,ydiv3,ydiv4:integer;
  dir1,dir2,dir3,dir4:byte;
  ly,gy,y,tmp,paso:integer;
begin
  { Determinar punto más alto y más bajo y ventana vertical }
```

```
ly:=MaxI(MinI(MinI(MinI(y1,y2),y3),y4),miny);
gy:=MinI(MaxI(MaxI(MaxI(y1,y2),y3),y4),maxy);

if ly>maxy then exit;
if gy<miny then exit;

{ Ver dirección (-1=arriba, 1=abajo) y calcular constantes }
dir1:=byte(y1<y2); xdiv1:=x2-x1; ydiv1:=y2-y1;
dir2:=byte(y2<y3); xdiv2:=x3-x2; ydiv2:=y3-y2;
dir3:=byte(y3<y4); xdiv3:=x4-x3; ydiv3:=y4-y3;
dir4:=byte(y4<y1); xdiv4:=x1-x4; ydiv4:=y1-y4;

y:=y1;
paso:=dir1*2-1;
if y1<>y2 then begin
  repeat
    if EnRango(y,ly,gy)=y then begin
      tmp:=xdiv1*(y-y1) div ydiv1+x1;
      pos[y,dir1]:=EnRango(tmp,minx,maxx);
    end;
    inc(y,paso);
  until y=y2+paso;
end
else begin
  if (y>=ly) and (y<=gy) then begin
    pos[y,dir1]:=EnRango(x1,minx,maxx);
  end;
end;

y:=y2;
paso:=dir2*2-1;
if y2<>y3 then begin
  repeat
    if EnRango(y,ly,gy)=y then begin
      tmp:=xdiv2*(y-y2) div ydiv2+x2;
      pos[y,dir2]:=EnRango(tmp,minx,maxx);
    end;
    inc(y,paso);
  until y=y3+paso;
end
else begin
  if (y>=ly) and (y<=gy) then begin
    pos[y,dir2]:=EnRango(x2,minx,maxx);
  end;
end;

y:=y3;
paso:=dir3*2-1;
if y3<>y4 then begin
  repeat
    if EnRango(y,ly,gy)=y then begin
      tmp:=xdiv3*(y-y3) div ydiv3+x3;
      pos[y,dir3]:=EnRango(tmp,minx,maxx);
    end;
    inc(y,paso);
  until y=y4+paso;
end
else begin
  if (y>=ly) and (y<=gy) then begin
    pos[y,dir3]:=EnRango(x3,minx,maxx);
  end;
```

```
  end;

  y:=y4;
  paso:=dir4*2-1;
  if y4<>y1 then begin
    repeat
      if EnRango(y,ly,gy)=y then begin
        tmp:=xdiv4*(y-y4) div ydiv4+x4;
        pos[y,dir4]:=EnRango(tmp,minx,maxx);
      end;
      inc(y,paso);
    until y=y1+paso;
  end
  else begin
    if (y>=ly) and (y<=gy) then begin
      pos[y,dir4]:=EnRango(x4,minx,maxx);
    end;
  end;

  for y:=ly to gy do horline(pos[y,0],pos[y,1],y,c);
end;

{ ----------------------------------------------------------------------
------ }

procedure quicksort(lo,hi:integer);
{ Una de las rutinas de ordenación más habituales.  Mucho mejor que
  burbuja (por ejemplo) cuando hay bastantes puntos }
procedure sort(l,r:integer);
var i,j,x,y:integer;
begin
  i:=l; j:=r; x:=polyz[(l+r) div 2];
  repeat
    while polyz[i]<x do inc(i);
    while x<polyz[j] do dec(j);
    if i<=j then begin
      y:=polyz[i]; polyz[i]:=polyz[j]; polyz[j]:=y;
      y:=pind[i]; pind[i]:=pind[j]; pind[j]:=y;
      inc(i); dec(j);
    end;
  until i>j;
  if l<j then sort(l,j);
  if i<r then sort(i,r);
end;

begin
  sort(lo,hi);
end;


{ ----------------------------------------------------------------------
------ }

procedure rotarImg;
{ Pues eso ;-) }
const
  xst=1; yst=2; zst=-3;
var
  xp,yp,z:array[0..NumPuntos] of integer;
  x,y,i,j,k: integer;
  n,Key,angx,angy,angz: byte;
```

```
begin
  angx:=0; angy:=0; angz:=0;
  fillchar(xp,sizeof(xp),0);
  fillchar(yp,sizeof(yp),0);
  repeat
    copia(segFondo,segTemp);
    for n:=0 to NumPuntos do begin
      { Proyectamos las coordenadas en 3D y luego a 2D }
      i:=(coseno(angy)*punto[n,0]-seno(angy)*punto[n,2]) div divd;
      j:=(coseno(angz)*punto[n,1]-seno(angz)*i) div divd;
      k:=(coseno(angy)*punto[n,2]+seno(angy)*punto[n,0]) div divd;
      x:=(coseno(angz)*i+seno(angz)*punto[n,1]) div divd;
      y:=(coseno(angx)*j+seno(angx)*k) div divd;
      z[n]:=(coseno(angx)*k-seno(angx)*j) div divd+coseno(angx) div 3;
      xp[n]:=160+seno(angx)+(-x*dist) div (z[n]-dist);
      yp[n]:=100+coseno(angx) div 2+(-y*dist) div (z[n]-dist);
    end;
    for n:=0 to NumPlanos do begin
      { Coordenada Z asignada al plano para sombrearlo: en función de
la
        distancia al observador (media de las Z de las esquinas).
Está
        dividido entre 5 y no entre 4 para limitar un poco más el
rango
        de valores que puede tomar }

polyz[n]:=(z[plano[n,0]]+z[plano[n,1]]+z[plano[n,2]]+z[plano[n,3]])
        div 5;
      pind[n]:=n;
    end;
    quicksort(0,NumPlanos);    { Ordenamos los planos }
    for n:=0 to NumPlanos do
      { Dibujamos los planos por orden }
      polygon(xp[plano[pind[n],0]],yp[plano[pind[n],0]],
              xp[plano[pind[n],1]],yp[plano[pind[n],1]],
              xp[plano[pind[n],2]],yp[plano[pind[n],2]],
              xp[plano[pind[n],3]],yp[plano[pind[n],3]],polyz[n]+55);
    inc(angx,xst); inc(angy,yst); inc(angz,zst);
    copia(segTemp,segVideo);  { Ponemos en la pantalla visible }
  until keypressed;
end;

{ ----------------------------------------------------------------------
------ }

var i,j:word;
begin
  asm mov ax,13h; int 10h; end;            { Modo 320x200, 256 colores
}
  for i:=0 to 255 do
    ctab[i]:=round(-cos(i*pi/128)*divd);   { Creo las tablas }
  for i:=0 to 255 do
    stab[i]:=round(sin(i*pi/128)*divd);
  minx:=0; miny:=0; maxx:=319; maxy:=199;  { Límites de la pantalla }

  getmem(pantTemp,64000);                  { Reservo la pantalla
temporal }
  segTemp := seg(pantTemp^);
  getmem(fondo,64000);                     { Y el fondo }
  segFondo := seg(fondo^);
```

```
  { Dibujo el fondo }
  for i:=0 to 319 do
    for j:=0 to 199 do
      mem[segFondo:j*320+i]:=(i+j) mod 102 +152;
  for i:=0 to 255 do stab[i]:=round(sin(i*pi/128)*divd);
  { Colores del rótulo }
  for i:=1 to 150 do setpal(i,30+i div 6,20+i div 7,10+i div 7);
  { Colores del fondo }
  for i:=151 to 255 do setpal(i,i div 7, i div 7,i div 5);

  rotarImg;

  { Se acabó -> liberamos la memoria reservada }
  freemem(pantTemp,64000);
  freemem(fondo,64000);

  { Y volvemos a modo texto }
  textmode(lastmode);
end.
```

Venga, a experimentar... }:-)