Curso de programación en Pascal

Este texto pretende ser una introducción a la programación de ordenadores en lenguaje Pascal.

Este texto ha sido escrito por Nacho Cabanes. Si quiere conseguir la última versión, estará en mi página web:

www.nachocabanes.com

Para cualquier sugerencia, no dude en contactar conmigo a través de mi web.

Revisión actual: 3.79a

Contenido

(Revisión actual: 3.79a)

Revisiones de este texto hasta la fecha:

- 3.79a, de 15-Ago-06. Revisión inicial de la versión 3.73, probando casi todos los ejemplos con Free Pascal. Incluye los temas 1 a 13, sin terminar de revisar.
- (No se incluyen aquí los cambios entre versiones anteriores)

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:

1. 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).
2. 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.

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 va 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.

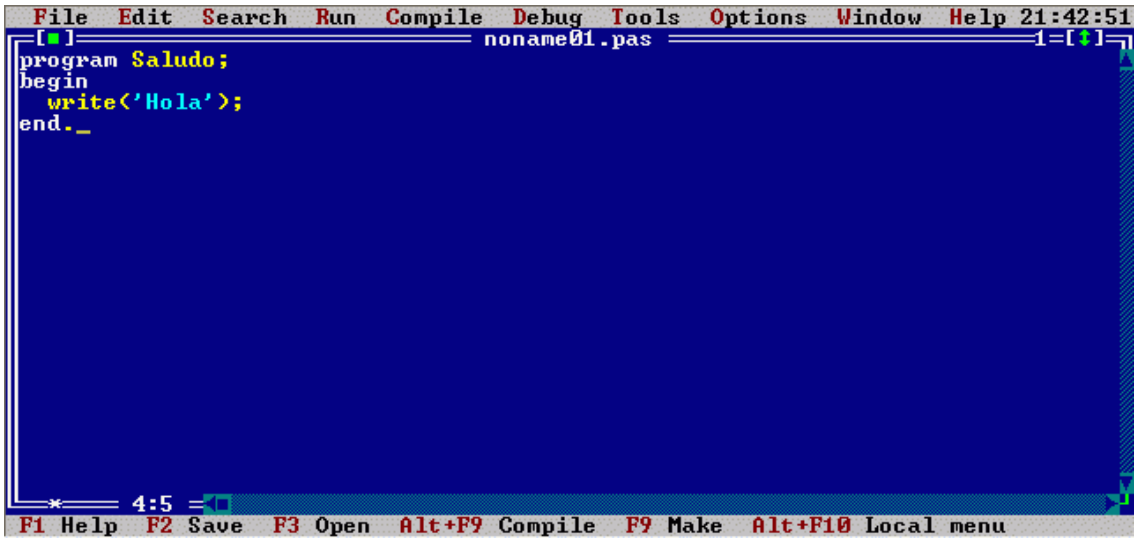
Los detalles concretos sobre **cómo probar este programa** dependerán del compilador que se esté utilizando. Unos tendrán un Entorno Integrado, desde el que escribir los programas y probarlos (como Free Pascal, Turbo Pascal y Surpas), mientras que en otros hará falta un editor para teclear los programas 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.

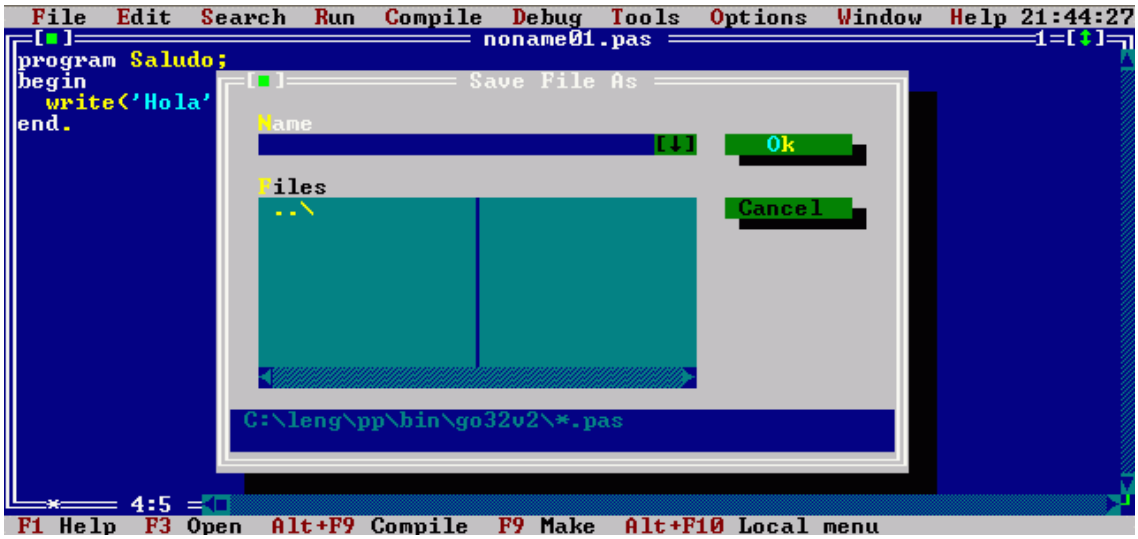
Tienes instrucciones sobre cómo instalar Free Pascal, pero en este momento vamos a suponer que ya está instalado y a ver cómo se teclearía y se probaría este programa bajo **Windows:**

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 ejemplo que hemos visto, pero que, en vez de escribir Hola, escriba tu nombre.*

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 defina una variable que sea un registro con dos campos: X e Y, ambos números enteros. El campo X debe valer 20, e Y debe valer 30. Después deberá mostrar en pantalla la suma de X e Y..*

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.

Tema 2: Ejercicios resueltos 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 l¡nea 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);      (* šQu, hara 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 compilado 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, 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:

```
{-------------------------}
{  Ejemplo en Pascal:     }
{                         }
{    Como escribir las    }
{      comillas           }
{    COMILLA.PAS          }
{                         }
{  Este fuente procede de }
{  CUPAS, curso de Pascal }
{  por Nacho Cabanes      }
{                         }
{  Comprobado con:        }
{    - 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:        }
{    - 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...

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:        }
{    - 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?

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:         }
{    - 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).

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

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:        }
 {    - Turbo Pascal 7.0   }
 {    - Tmt Pascal Lt 1.20 }
 {-------------------------}

 program BitOps;   { Operaciones entre bits }

var
   a, b: integer;

 begin

   a := 67;
   b := 33;
   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...

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:        }
```

```
{    - 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).

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:         }
{    - 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:        }
{    - 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.

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:        }
{    - 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.

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:        }
{    - 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.
```

Tema 5.6: Varias condiciones simultáneas.

Si se deben cumplir **varias condiciones** a la vez, podemos **enlazarlas** con "and" (y). 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 [...]
```

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:         }
{    - 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").

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 2.0.2   }
{    - 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.
```

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 2.0.2  }
{    - 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.
```

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 2.0.2  }
{    - 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 2.0.2   }
{    - 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".

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".

Como ejercicios propuestos:

1.- Un programita que escriba los números 2, 4, 6, 8 ... 16.
2.- Otro que escriba 6, 5, 4,..., 1.
3.- Otro que escriba 3, 5, 7,..., 21.
4.- Otro que escriba 12, 10, 8,..., 0.
5.- Otro que escriba los números pares hasta el 20, excluyendo el 12.
6.- Otro que multiplique dos matrices.
7.- Para los más osados (y que conozcan el problema), uno 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:          }
{    - Free Pascal 2.0.2    }
{    - 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');
```

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 2.0.2   }
{    - 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? Es bastante por hoy. Ahora os toca experimentar a vosotros.

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

Los **ejercicios** propuestos sobre While y Repeat son:

1. Mejorar el programa de la clave de acceso para que avise de que la clave no es correcta.
2. Mejorar más todavía para que sólo haya tres intentos.

3. Adaptar la primera versión (el ejemplo), la segunda (la mejorada) y la tercera (re-mejorada) para que empleen "while" y no "until".

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:         }
{    - Free Pascal 2.0.2   }
{    - 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
...
```

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: 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:        }
{    - 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:         }
{    - 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...

Tema 8: Procedimientos y funciones.

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.
```

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.

Tema 8.2: Procedimientos y 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.

Tema 8.3: Procedimientos y funciones (3).

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:        }
{    - 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:         }
{    - 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( 'Juan' );     (* Saludamos a Juan *)
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.

Tema 8.4b: Ejercicios propuestos.

Veamos unos ejercicios propuestos de lo que hemos visto:

- Adaptar la función "potencia" que hemos visto para que trabaje con números reales, y permita cosas como 3.2 ^ 1.7.
- Hacer una función que halle la raíz cúbica del número que se le indique.
- Definir las funciones suma y producto de tres números y hacer un programa que haga una operación u otra según le indiquemos (con "case", etc).
- Un programita que halle la letra (NIF) que corresponde a un cierto DNI.
- Crear un programita 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: Procedimientos con parámetros (2).

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:         }
{    - 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: Procedimientos con parámetros (3).

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:        }
{    - 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".

Tema 8.7: Procedimientos con parámetros (4).

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:         }
{    - 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:         }
{    - 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.

Tema 8.8b: 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").

Tema 8.9: Ejercicios propuestos.

Un par de ejercicios muy facilitos, y otro algo más difícil:

- Una función recursiva que halle el producto de dos números enteros.
- Otra que halle la potencia (a elevado a b), también recursiva.
- 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).

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:         }
{    - Turbo Pascal 7.0    }
{    - Turbo Pascal 5.0    }
{    - Surpas 1.00         }
{    - FreePascal 2.0.2    }
{--------------------------}

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:        }
{    - Turbo Pascal 7.0   }
{    - Turbo Pascal 5.0   }
{    - Surpas 1.00        }
{    - Tmt Pascal Lt 1.01 }
{    - FreePascal 2.0.2   }
{-------------------------}

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... X-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 altavozx 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:         }
{    - 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:        }
 {    - 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:

(## Imagen todavía no disponible ##)

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:         }
{    - 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 11: 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).

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:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{--------------------------}

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:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{--------------------------}

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:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{--------------------------}

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.                                         (* Y se acabó *)
```

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:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{    - 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:         }
{    - Turbo Pascal 7.0    }
{    - Free Pascal 2.0.2   }
{--------------------------}

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:        }
{    - 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:         }
{    - 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:         }
{    - 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:        }
{    - 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:         }
{    - 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:         }
{    - 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.

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:         }
{    - 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 }
```

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;
```

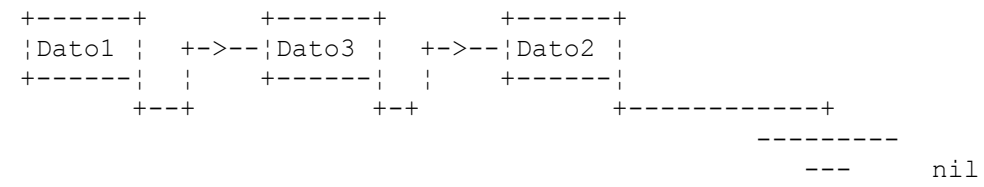
Si quisieramos introducir los datos **ordenados alfabéticamente**, basta con ir comparando cada nuevo dato con los de la lista, e insertarlo donde corresponda. Por ejemplo, para insertar un nuevo dato entre los dos anteriores, haríamos:

```
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:         }
{    - 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.
```

Ejercicios propuestos:

1. ¿Se podría quitar de alguna forma el segundo "else" de InsertaLista?
2. ¿Cómo sería un procedimiento que borrase toda la lista?
3. ¿Y uno de búsqueda, que devolviera la posición en la que está un dato, o NIL si el dato no existe?
4. ¿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)

```
  5
 / \
3   7
```

```
   /
  2
```

Quinto: 4 (menor que 5, mayor que 3)

```
     5
    / \
   3   7
  / \
 2   4
```

Sexto: 8 (mayor que 5, mayor que 7)

```
     5
    / \
   3   7
  / \   \
 2   4   8
```

Séptimo: 9 (mayor que 5, mayor que 7, mayor que 8)

```
     5
    / \
   3   7
  / \   \
 2   4   8
          \
           9
```

¿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). Creo recordar que esto funciona en Turbo Pascal O porque marca esa zona de memoria como disponible pero no la borra físicamente.

Esto 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:

```
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:         }
{    - 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.

Aquí van unos ejercicios propuestos:

- Implementar una pila de strings[20].
- Implementar una cola de enteros.
- Implementar una lista doblemente enlazada que almacene los datos leídos de un fichero de texto (mejorando el lector de ficheros que vimos).
- Hacer lo mismo con una lista simple, pero cuyos elementos sean otras listas de caracteres, en vez de strings de tamaño fijo.

- 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).
- ¿Cómo se borraría un único elemento del árbol?

N.