ACADEMIA CARTAGENA99
C/ Cartagena 99, Bº C , 28002 Madrid
91 51 51 321
academia@cartagena99.com.es

Curso de Pascal

Tema 0. Introducción..... 3
Tema 1. Generalidades del Pascal. 4
Tema 2. Introducción al manejo de variables. 5
Tema 3. Entrada/Salida básica. 10
Tema 5: Condiciones..... 13
Tema 6: Bucles. 16
Tema 7: Constantes y tipos. 22
Tema 8: Procedimientos y funciones..... 27
Tema 9: Otros tipos de datos. 36
Tema 10: Pantalla en modo texto. 40
Tema 11: Manejo de ficheros..... 44
Tema 12: Creación de unidades..... 55
Tema 13: Variables dinámicas..... 59
Tema 13b: ARBOLES. 70
Apéndice 1: Otras órdenes no vistas. 73
Apéndice 2: Palabras reservadas de Turbo Pascal (ordenadas alfabéticamente) 77

Tema 0. Introducción.

Hay distintos lenguajes que nos permiten dar instrucciones a un ordenador. El 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.
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.

La mayoría de los lenguajes actuales son compiladores, y suelen 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.

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 (es el caso de Visual Basic).
- **COBOL**, que fue muy utilizado para negocios, aunque últimamente está bastante en desuso.
- **FORTRAN**, concebido para ingeniería, operaciones matemáticas, etc. También va quedando desplazado.
- **Ensamblador**, muy cercano al código máquina, pero sustituye las secuencias de ceros y unos (bits) por palabras más fáciles de recordar, como MOV, ADD, CALL o JMP.
- **C**, el mejor considerado actualmente, porque no es difícil 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.
- **PASCAL**, el lenguaje estructurado (ya se irá viendo esto más adelante) por excelencia, 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 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, pero resulta más fácil de aprender, es muy rápido, crea ficheros EXE más pequeños, etc.

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. De momento, anticipemos que "**Object 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.

Tema 1. Generalidades del Pascal.

Como lenguaje **estructurado** que es, muchas veces habrá que dividir en bloques las distintas partes que componen un programa. Estos bloques se denotan marcando su principio y su final con las palabras **begin** y **end**.

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. 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, o 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.)

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...

Con poco más que lo visto hasta ahora ya se podría escribir un pequeño programa que hiciera aparecer el mensaje "Hola" en la pantalla:

```
program Saludo;


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

La palabra **program** no es necesaria en muchos compiladores actuales, pero sí lo era inicialmente en Pascal estándar, y el formato era

```
program NombrePrograma (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 se usa el Pascal de GNU, deberá poner:

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

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

Saludo es un **identificador** que nos va a servir para indicar el nombre del programa. Los "identificadores" son palabras que usaremos para referirnos a una variable, una constante, el nombre de una función o de un procedimiento, etc.

Una **variable** equivale a la clásica incógnita "x" que todos hemos usado en matemáticas, que puede ser cualquier número. Ahora nuestras "incógnitas" podrán tener cualquier valor (no sólo un número: también podremos guardar textos, fichas sobre personas o libros, etc.) y nombres más largos (y que expliquen mejor su contenido).

Estos nombres de "identificadores" serán combinaciones de letras 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; así que podemos dejarlo, por si después añadimos otra orden entre "write" y "end".

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. 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.
```

Una última **observación**: si se compila este programa desde Turbo Pascal 5.0 o una versión superior, aparentemente "no pasa nada". No es así, sino que se ejecuta y se vuelve al editor tan rápido que no nos da tiempo a verlo. La solución es pulsar *Alt+F5* para que nos muestre la pantalla del DOS.

Tema 2. Introducción al manejo de 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;


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.

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.

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**".

Finalmente, aparece en pantalla la palabra "Hola" seguida por el nombre que se ha introducido. Como se ve en el ejemplo, "writeln" puede escribir más de un dato, pero eso lo estudiaremos con detalle un poco más adelante...

Tipos básicos de datos

En Pascal debemos **declarar** las variables que vamos a usar. Esto puede parecer incómodo para quien ya haya trabajado en 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:

- **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.

- **Integer**. Es un número entero con signo, que puede valer desde -32768 hasta 32767. Ocupa 2 bytes de memoria.

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

- **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]**, 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 podemos usar el formato "string[n]" o simplemente "string", que equivale a "string[255]", como aparecía en el ejemplo anterior.

- **Real**. Es un numero real con signo. Puede almacenar números con valores entre 2.9e-39 y 1.7e38 (en notación científica, e5 equivale a multiplicar por 10^5), con 11 o 12 dígitos significativos, y que ocupan 6 bytes en memoria.

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

- **Array**. Se emplea para definir **vectores o matrices**. Se deberá indicar el índice inferior y superior, separados por dos puntos (**..**), así como el tipo de datos.

Ejemplo: un vector formado por 10 números enteros sería

```
vector: array[1..10] of integer
```

y una matriz de dimensiones 3x2 que debiera contener números reales:

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

Para mostrar en pantalla el segundo elemento del vector se usaría

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

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

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

- **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 Ejemplo_de_registro;
```

```
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 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).

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 Ejemplo_de_registro;


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.

Ejemplos.

Ejemplo 1: 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.
```

Ejemplo 2: 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.
```

Ejemplo 3: 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.

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.
```

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.

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.

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:

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

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.

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:

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

La salida por pantalla de este programa sería:

```
 1.2345678900E+06
         1234567.890

.ej1234567.89

.ej1234567.9
```

```
.ej1234568

 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.

En este programa ha aparecido también otra cosa nueva: los **comentarios**. Un comentario es algo que no se va a ejecutar, 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.

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.

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.
- ...

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.

En operaciones como +, - y * no debería haber ninguna duda. 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:

```
program operaciones;


var
  e1, e2: integer;      (* Números 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.
```

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

```
var
  texto1, texto2, texto3: string;


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

Cuando tratemos tipos de datos más avanzados, veremos que +, - y * también se pueden utilizar para **conjuntos**, e indicarán la unión, diferencia e intersección.

Operadores lógicos

Vimos de pasada que en el tema 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.
```

Los operadores "and", "or" y "not", junto con otros, se pueden utilizar también para **operaciones entre bits** de números enteros. Lo comento de pasada para no liar a los que empiezan. De momento, ahí va resumido y sin más comentarios. Quien quiera saber más, lo podrá ver en las ampliaciones al curso básico.

```
  Operador Operación

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


Queda como ejercicio hallar (y tratar de entender) el resultado de este
programita:
    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 );
      writeln( 125 and 6 );           (* Este para los más osados *)
    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 condición then sentencia**". Veamos un ejemplo breve antes de seguir:

```
    program if1;


    var numero: integer;


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

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 la anterior, como una propia variable booleana. Así, una forma más "rebuscada" (pero que a veces resultará más cómoda y más legible) de hacer lo anterior sería:

```
    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.

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

```
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.

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**:

```
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.

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

```
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.
```

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 no usa TP/BP, sino un compilador que protesta con el "else", sólo tiene que probar con "otherwise".

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

```
program case1;


var
```

```
    letra: char;


begin
  WriteLn('Escriba un letra');
  ReadLn(letra);
  case letra of
    ' ':                WriteLn('Un espacio');
    'A'..'Z', 'a'..'z':  WriteLn('Una letra');
    '0'..'9':           WriteLn('Un dígito');
    '+', '-', '*', '/':  WriteLn('Un operador');
  else
    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**).

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 empezar a ver ya por encima:

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

La diferencia entre estos dos últimos es que "while" comprueba la condición antes de ejecutar las otras 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...

For.

El formato de "**for**" es

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

Vamos a ver algunos ejemplos.

Primero, un miniprograma que escriba los números del uno al diez:

```
var
  contador: integer;
```

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

Los bucles "for" se pueden **enlazar** uno dentro de otro, de modo que un segundo ejemplo que escribiera las tablas de multiplicar del 1 al 5 podría ser

```
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.
```

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.

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

```
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.
```

Conviene recordar que es muy conveniente usar la **escritura indentada**, que en este caso ayuda a ver dónde empieza y termina lo que hace cada "for".

Una observación: para "contar" no necesariamente hay que usar números:

```
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 usar "trucos": 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).

Como **ejercicios** propuestos:

1. Un programa 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 multiplique dos matrices.
6. Para los que conozcan el problema, uno de resolución de sistemas de ecuaciones por Gauss.

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 verlas con más detalle:

La sintaxis de "**while**"

```
while condición do
  sentencia;
```

Se podría traducir como "MIENTRAS condición HAZ sentencia", o sea, que la sentencia se va a repetir mientras la condición sea cierta.

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

```
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) );
    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');
```

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. Cuidado 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 *.

```
program ClaveDeAcceso;


var
  ClaveCorrecta, Intento: String;


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

Como ejercicios propuestos:

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 ha programado en Basic puede que se pregunte por la orden **goto.** Existe en Pascal, pero su uso va a en contra de todos los principios de la Programación Estructurada, solo está "medio-permitido" en casos muy concretos, así que lo veremos más adelante.

Soluciones a los ejercicios.

Vamos a empezar a resolver los ejercicios que habíamos ido proponiendo.

Antes que nada, hay que puntualizar una cosa: el que así se resuelva de una forma no quiere decir que no se pueda hacer de otras. Ni siquiera que la mía sea la mejor, porque trataré de adaptarlos al nivel que se supone que tenemos.

1.- Un programa que escriba los números 2, 4, 6, 8 ... 16.

```
program del2al16;

var i: integer;

begin
  for i := 1 to 8 do
    writeln( i*2 );
end.
```

2.- Otro que escriba 6, 5, 4,..., 1.

```
program del6al1;

var i: integer;

begin
  for i := 6 downto 1 do
    writeln( i );
end.
```

3.- Otro que escriba 3, 5, 7,..., 21.

```
program del3al21;

var i: integer;

begin
  for i := 1 to 10 do
    writeln( i*2 +1 );
end.
```

4.- Otro que escriba 12, 10, 8,..., 0.

```
program del12al0;
```

```
var i: integer;

begin
  for i := 6 downto 0 do
    writeln( i*2 );
end.
```

5.- Otro que multiplique dos matrices.

Saltamos la parte que declara las variables y que pide los datos. La parte de la multiplicación sería, para matrices cuadradas de 10 por 10, por ejemplo:

```
for i := 1 to 10 do
  for j := 1 to 10 do
    c[i,j] := 0;                    (* limpia la matriz destino *)

 for i :=1 to 10 do
   for j := 1 to 10 do
     for k := 1 to 10 do
       c[i,j] := c[i,j] + a[k,j] * b[i,k];
```

6.- Resolución de sistemas de ecuaciones por Gauss.

La única dificultad es conocer el problema: basta hacer 0 por debajo de la diagonal principal (y encima también, si se quiere, pero no es necesario) e ir despejando cada variable.

7.- Mejorar el programa de la clave de acceso para que avise de que la clave no es correcta.

```
program ClaveDeAcceso2;

var
  ClaveCorrecta, Intento: String;

begin
  ClaveCorrecta := 'PascalForever';
  repeat
    WriteLn( 'Introduce la clave de acceso...' );
    ReadLn( Intento )
    if Intento <> ClaveCorrecta then
      writeln( ' Esa no es la clave correcta! ');
  until Intento = ClaveCorrecta
 (* Aquí iría el resto del programa *)
end.
```

2.- Mejorar más todavía para que sólo haya tres intentos.

```
program ClaveDeAcceso3;

var
  ClaveCorrecta, Intento: String;
  NumIntento: integer;                  (* número de intento *)

begin
  ClaveCorrecta := 'PascalForever';
  NumIntento := 0;                     (* aún no hemos probado *)
  repeat
    NumIntento := NumIntento + 1;        (* siguiente intento *)
    WriteLn( 'Introduce la clave de acceso...' );
    ReadLn( Intento )
    if Intento <> ClaveCorrecta then
      begin
      writeln( ' Esa no es la clave correcta! ');
      if NumIntentos = 3 then exit        (* sale si es el 3º *)
      end
  until Intento = ClaveCorrecta
 (* Aquí iría el resto del programa *)
end.
```

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

Sólo ponemos una de ellas, porque las demás son muy similares.

```
program ClaveDeAcceso4;              (* equivale a ClaveDeAcceso2*)

var
  ClaveCorrecta, Intento: String;

begin
  ClaveCorrecta := 'PascalForever';
  Intento := '';                      (* cadena vacía *)
  while Intento <> ClaveCorrecta do   (* mientras no acertemos *)
    begin
    WriteLn( 'Introduce la clave de acceso...' );
    ReadLn( Intento )
    if Intento <> ClaveCorrecta then
      writeln( ' Esa no es la clave correcta! ');
    end;                               (* fin del "while" *)
 (* Aquí iría el resto del programa *)
end.
```

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 exista en los últimas versiones del lenguaje). Me explico con un ejemplo :

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...

La solución será 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
...
```

El identificador "const" tiene también en Turbo Pascal otro uso menos habitual: definir lo que se llaman **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. 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. Podemos hacer

```
const intentos: integer = 3;


begin
...
```

Recordemos que una "constante con tipo" se manejará **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.

Definición de tipos.

El tipo de una variable es lo que indicamos cuando la declaramos:

```
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...

```
program PruebaTipos;


var


  ficha1: record
    nombre: string;
    direccion: string;
    edad: integer;
    observaciones: string
  end;


  ficha2: record
    nombre: string;
    direccion: string;
    edad: integer;
    observaciones: string
  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.
```

Veamos qué haría este programa: 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...

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 que se entenderá mejor más adelante, cuando empecemos 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

```
type TipoFicha = record
    nombre: string;
    direccion: string;
    edad: integer;
    observaciones: string
  end;


var ficha1: TipoFicha;
...
var ficha2: TipoFicha;
```

```
...
```

Ahora sí que podremos asignar valores entre variables que hayamos definido en distintas partes del programa, podremos usar esos tipos para crear ficheros (que también veremos más adelante), etc.

Tema 8: Procedimientos y funciones.

Conceptos básicos.

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, si queremos crear nuestra, 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 algo parecido a

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

Así resulta bastante más fácil de seguir.

En nuestro caso, 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. Veámoslo 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.
- 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 no podemos acceder a las locales desde fuera del procedimiento en el que las hemos definido.
- La función **exit**, que no habíamos visto aún, permite interrumpir la ejecución del programa (o de un procedimiento) en un determinado momento.

Veamos el segundo ejemplo: una **función** que eleve un número a otro (no existe 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;                    (* inicialización *)
  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.

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

```
program PruebaDePotencia;


var
  numero1, numero2: integer;                (* Variable 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.
```

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

```
program ProcConParametros;


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


begin                   (* Comienzo del programa *)
  writeln;              (* Línea en blanco *)
  saludo( 'Eva' );      (* Saludamos a Eva *)
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.

Pero antes, 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 programa que halle la letra (NIF) que corresponde a un cierto DNI.

Parámetros.

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 tema, cuando empezamos a usar "WriteLn":

```
writeln( 'Hola' );
```

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

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

```
program PruebaDeParametros;


var dato: integer;


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


begin
  dato := 2;
  writeln( dato );
  modifica( dato );
  writeln( dato );
end.
```

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". Todos de acuerdo en que 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**. Si realmente queremos modificar el parámetro, lo que hacemos es simplemente añadir la palabra "var" delante de cada parámetro que queremos permitir que se pueda modificar. El programa quedaría:

```
program PruebaDeParametros2;
```

```
var dato: integer;


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


begin
  dato := 2;
  writeln( dato );
  modifica( dato );
  writeln( dato );
end.
```

Esta vez la última línea del programa sí que escribe un 3 y no un 2, porque hemos permitido que los cambio hechos a la variable salgan del procedimiento. 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".

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

```
program PruebaDeParametros3;


var dato: integer;


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


begin
  dato := 2;
  writeln( dato );
  modifica( dato );
  writeln( dato );
end.
```

Recursividad.

La idea en sí es muy sencilla: un procedimiento o función es recursivo si se llama a sí mismo. Para buscar un utilidad, vamos a verlo con un ejemplo clásico: el factorial de un número.

Partimos de la definición de factorial:

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

Por otra parte,

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

Luego podemos escribir cada factorial en función del factorial del siguiente número:

n! = n · (n-1)!

Acabamos de dar la definición recursiva del factorial. Ahora sólo queda ver cómo se haría eso programando:

```
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.
```

Dos **comentarios** sobre este programa:

- 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 deje el ordenador colgado.
- No conviene poner números demasiado grandes. Los enteros van desde -32768 hasta 32767, luego si el resultado es mayor que este número, tendremos un desbordamiento y el resultado será erróneo. En cuanto a qué es "demasiado grande": 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 parece preocupante, no hay por qué darle muchas vueltas, porque en el próximo tema veremos que hay otros tipos de datos que almacenan números más grandes o que nos permiten realizar ciertas cosas con más comodidad.

Un par de **ejercicios**:

- Una función recursiva que halle el producto de dos números enteros.

- Otra que halle la potencia (a elevado a b), también recursiva.

Soluciones.

Pues aquí van unas soluciones (insisto en que no tienen por qué ser las únicas ni las mejores) a los ejercicios propuestos en el tema 8:

1.- Adaptar la función "potencia" que hemos visto para que trabaje con números reales, y permita cosas como 3.2 ^ 1.7

Partimos del programa propuesto como ejemplo, que era:

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

No basta con cambiar los tipos de las variables, poniendo "real" en vez de "integer". Si hacemos esto, ni siquiera podremos compilar el programa, porque una variable de tipo "real" no se puede usar para controlar un bucle.

Una forma de hacerlo es empleando números exponenciales y logaritmos: como son operaciones inversas, tenemos que exp(log(x)) = x, donde "log" y "exp" sí que son funciones predefinidas en Pascal. Pero, por otra parte, sabemos que log (a^b) = b · log a. Así, una forma (no la mejor) de hacerlo sería simplemente

```
function PotReal(a,b: real): real;   (* a elevado a b, reales *)
begin

  PotReal := exp ( b * log ( a )) ;
end;
```

2.- Hacer una función que halle la raíz cúbica del número que se le indique.

Mirando la función anterior, ya es fácil: una raíz cúbica es lo mismo que elevar a 1/3, así que se puede hacer;

```
function RaizCubica(n: real): real;
begin
  RaizCubica := exp ( 0.33333333 * log ( n )) ;
end;
```

3.- 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.).

Trabajando con números reales, sería:

```
program Suma_Y_Producto;
```

```
var
  num1, num2: real;                    (* Definición de variables *)
  opcion: char;

function suma( a,b : real): real;    (* Esta es la función "suma" *)
begin
  suma := a + b ;
end;

function producto( a,b : real): real;           (* y el producto *)
begin
  producto := a * b ;
end;

begin
  write( 'Introduzca el primer número: ');
  readln( num1 );
  write( 'Introduzca el segundo número: ');
  readln( num2 );
  writeln( '¿Qué operación desea realizar?' );
  writeln( 's = Suma       p = Producto' );
  readln( opcion );
  case opcion of
    's': writeln( 'Su suma es ', suma(num1,num2) );
    'p': writeln( 'Su producto es ', producto(num1,num2) );
  else
    writeln( 'Operación desconocida!' );
  end;                                    (* Fin del case *)
end.                                      (* y del programa *)
```

4.- Un programa que halle la letra que corresponde a un cierto DNI.

La forma de calcular la letra que corresponde a un cierto DNI es muy sencilla: basta con dividir el número del DNI entre 23 y coger el resto de esa división. Según el valor de ese resto, se asigna una letra u otra:

0=T 1=R 2=W 3=A 4=G 5=M 6=Y 7=F 8=P 9=D 10=X 11=B 12=N 13=J

14=Z 15=S 16=Q 17=V 18=H 19=L 20=C 21=K 22=E

Así, un programa que halle la letra (implementando esta parte como función, para que quede "más completo") puede ser simplemente:

```
program Nif;              (* Letra del NIF.  Nacho Cabanes, Jun. 92 *)

var numero:longint;

function LetraNif ( dni: longint ): char;
const valores: string[24]='TRWAGMYFPDXBNJZSQVHLCKE';
begin
  LetraNif := valores [( dni mod 23 ) + 1];
end;

begin
  writeln('¿Cual es el DNI cuyo NIF quiere hallar?');
  readln(numero);
  writeln('La letra es ', LetraNif( Numero ) ,'.');
end.
```

5.- Qué ocurre si el primer programa del tema 8.2 lo modificamos para que sea así:

```
program PruebaDeParametros3;

var dato: integer;

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

begin
  dato := 2;
  writeln( dato );
  modifica( dato );
  writeln( dato );
end.
```

Sólo ha cambiado el nombre de la variable global, que ahora se llama "dato", igual que la local. Pero de cualquier modo, ambas variables SON DISTINTAS, de modo que el programa sigue funcionando igual que antes, cuando los nombres de las variables eran distintos.

Es aconsejable, para evitar problemas, no utilizar variables globales y locales con el mismo nombre.

6.- Hallar una función recursiva que halle el producto de dos números enteros positivos.

Este es un caso con poco sentido en la práctica, pero que se puede resolver simplemente recordando que un producto no son más que varias sumas:

```
function producto( a,b : integer) : integer;
var temporal: integer;
begin
  if b = 1 then
    producto := a
  else
    producto := a + producto (a, b-1 );
end;
```

Y un programa que la utilizase sería:

```
var num1, num2: integer;

begin
  write( 'Introduzca el primer número: ');
  readln( num1 );
  write( 'Introduzca el segundo número: ');
  readln( num2 );
  writeln( 'Su producto es ', producto(num1,num2) );
end.
```

7.- Otra que halle la potencia (a elevado a b), también recursiva.

La idea es la misma que antes: simplificar el problema, esta vez escribiendo la potencia como varios productos:

```
function potencia( a,b : integer) : integer;
var temporal: integer;
begin
  if b = 1 then
    potencia := a
  else
    potencia := a * potencia (a, b-1 );
end;
```

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.

Esta vez vamos a ampliar con otros tipos de datos que podemos encontrar en Turbo Pascal (aunque puede que no en otras versiones del lenguaje Pascal).

Como resumen previo, vamos a ver lo siguiente:

- 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**. Sin 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, Ó=224). La forma de hallar el código ASCII de una letra es simplemente ord(letra), como ord('Ó').

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):

```
var bucle: byte;


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

Si tenemos **coprocesador** matemático, (vamos a suponer que no es el caso y no vamos a dar más detalles de su uso), 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 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, el en ejemplo anterior

```
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).

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]"). Ocupa n+1 bytes porque también se guarda la longitud de la cadena (el espacio que ocupa realmente).

Vamos a ver cómo acceder a caracteres individuales de una cadena. 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. Esto se hace en la posición 0. Veamos un programa de ejemplo:

```
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.

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.

```
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);             (* 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 *)


  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 *)


  UnDato.Nombre := 'Nacho3';     (* Campo normal *)
  UnDato.Tipo   := Str;          (* Ahora un string *)
  UnDato.S      := 'Nada';       (* el texto "Nada" *)
end.
```

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...' );
```

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.

Tema 10: Pantalla en modo texto.

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, o en otros compiladores (como TMT Pascal Lite) y de cualquier modo, nada de esto es Pascal estándar.

Nos centraremos primero en cómo se haría con las versiones **5.0 y superiores** de Turbo Pascal. Luego comentaremos cómo se haría con Turbo Pascal 3.01.

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 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
  [...]
```

Vamos a mencionar algunos de los procedimientos y funciones más importantes. Para ver los demás, basta pasearse por la ayuda de Turbo Pascal: escribir **crt** y pulsar Ctrl+F1 encima de esa palabra, que nos muestra información sobre esa palabra clave (o la que nos interesase).

- **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:

- 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 vídeo, pero eso es bastante más complicado.

Aquí va un programa de ejemplo que maneja casi todo esto:

```
program PruebaDeCRT;

{ Acceso a pantalla en modo texto }
{ Exclusivo de Turbo Pascal       }

{ -- Comprobado con: TP 7.0 -- }

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 dispositivo especial, el teclado, denotado con kbd) en vez de con "tecla := readkey". Con estos dos cambios, el programa anterior funciona perfectamente.

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 hemos 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:

```
program PruebaDeCRT;        { Versión para SURPAS }


{ ----- Aquí 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.
```

ACADEMIA CARTAGENA99
C/ Cartagena 99, Bº C , 28002 Madrid
91 51 51 321
academia@cartagena99.com.es

Tema 11: Manejo de ficheros.

Ahora vamos a ver cómo podemos leer ficheros y crear los nuestros propios. Voy a dividir este tema en tres 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.

Ficheros de texto.

Vayamos a lo que nos interesa hoy: un **fichero de texto**. Es un fichero formado por caracteres ASCII 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 ya es más complicado).

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.

Nos vamos a centrar en el manejo de ficheros con Turbo Pascal.

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**.

```
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. Debería ser 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.

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 );
```

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:

```
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).

Como último ejemplo sobre este tema, un programa que lea el AUTOEXEC.BAT y cree una copia en el directorio actual llamada AUTOEXEC.BAN. La orden del DOS equivalente sería **COPY C:\AUTOEXEC.BAT AUTOEXEC.BAN**

```
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.
```

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).

Ficheros con tipo.

Ya hemos visto cómo acceder a los ficheros de texto, tanto para leerlos como para escribir en ellos. Hoy 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.

Pues 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 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:

```
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 *)
```

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**".

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

```
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.
```

El listado debe ser 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, queda propuesto elaborar una **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.

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..."

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 "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 (el resultado de multiplicar "cuantos" por el tamaño de cada dato).

El significado de "Resultado" es el siguiente: nos indica cuantos datos ha leído realmente. De este modo, si vemos que le hemos dicho que leyera 30 fichas y sólo ha leído 15, 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 programa que copia un fichero leyendo bloques de 2K:

```
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.
```

Una mejora: 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.

Y un segundo ejemplo, que muestra parte de la información contenida en la cabecera de un fichero **GIF**, leyendo un "record":

```
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('Versión: ', Firma, NumVer);
        Writeln('Resolución: ', Tam_X, 'x',
          Tam_Y, 'x', 2 SHL (_Packed and 7));
      end;
    end.
```

Ejemplo: agenda.

```
{ ========================================================
  Ejemplo de Agenda, con lo que hemos visto hasta ahora.

  Características:
  - Número de fichas ilimitado, pero mayor lentitud, porque
    los datos están permanentemente en disco, y en memoria se
    almacena sólo la ficha actual.
  - Muestra un ficha.
  - Se puede ver la siguiente, la anterior o la número "x"
  - Se pueden añadir fichas nuevas.

  Posibles mejoras:
  - Muuuuuchas.  Por ejemplo...
  - Mejorar la presentación.
  - Mejorar la entrada de datos y su modificación.
  - Buscar un determinado texto en las fichas.
  - Imprimir una o varias fichas.
  - Ordenar el fichero alfabéticamente.
  - Mantener todas o parte de las fichas en memoria para mayor
    velocidad.
  - Borrar fichas.
  - Clave de acceso.
  - Datos encriptados.
  - Etc, etc, etc...

  Nacho Cabanes, Marzo 93.
  ======================================================== }

program MiniAg;

uses crt;                        { Manejo de la pantalla y el teclado }

const
  nombref: string[12]='agenda.dat';              { Nombre del fichero }

type                                          { Nuestro tipo de datos }
   tipoagenda = record
      nombre:    string[20];
      direccion: string[30];
      ciudad:    string[15];
      cp:        string[5];
      telef:     string[12];
      observ:    string[40]
      end;

var
    FichAgenda: file of tipoagenda;                          { Fichero }
    ficha: TipoAgenda;                        { Guarda la ficha actual }
    NumFicha: word;                         { El número de ficha actual }
    Ultima: word;                           { Número de la última ficha }
```

```
    opcion: char;                    { La opción del menú que se elige }


procedure Pausa;                { ----- Espera a que se pulse una tecla }
var tecla: char;
begin
  tecla := readkey;
end;


procedure Saludo;                    { ----- Cartelito de presentación }
begin
  TextBackground(Black); TextColor(LightGray);
  ClrScr;
  window(23,3,80,25);
  writeln;writeln;writeln;
  TextColor(LightCyan);
  writeln('+------------------------------+');
  writeln('¦                              ¦__');
  writeln('¦                              ¦__');
  writeln('¦         A G E N D A          ¦__');
  writeln('¦                              ¦__');
  writeln('¦                              ¦__');
  writeln('+------------------------------+__');
  writeln('  ____________________________________');
  TextColor(LightGray);
  window(1,1,80,25);
  gotoxy(1,25);                        { Para que el cursor no moleste }
  pausa;
end;


procedure Escribe;               { ----- Escribe los datos en pantalla }
var
  i: byte;                                              { i: para bucles }
begin
  ClrScr;
  TextColor(White);
  TextBackground(Blue);
  gotoxy(1,2);write('    Agenda    (ficha actual: ',
    NumFicha,'/',ultima,')');
  clrEol;                  { borra hasta el final de la línea en azul }
  gotoxy(1,3);
      for i:= 1 to 80 do write('-');                      { 80 guiones }
  TextBackground(Black);
  TextColor(LightGray);
  seek( FichAgenda, NumFicha-1 );    { se coloca en la ficha que toca }
  read( FichAgenda, ficha );                                 { y la lee }
  with ficha do
    begin                                           { Escribe cada dato }
    gotoxy(1,6);
    writeln('Nombre:  ', nombre); writeln;
    writeln; writeln('Dirección: ', direccion);
    writeln; writeln('Ciudad: ', ciudad);
    writeln; writeln('Código Postal: ', cp);
    writeln; writeln('Teléfono: ', telef);
    writeln; writeln;
    writeln('Observaciones:  ',observ);
    end;
  TextColor(White);
  TextBackground(Blue);
    gotoxy(1,23);                        { Abajo: escribe las opciones }
      for i:= 1 to 80 do write('-');
    gotoxy(1,24);write('      1-Anterior  2-Posterior  3-Número'
      + '  4-Añadir  5-Corregir  0-Terminar');
  clrEol;
```

```
  TextBackground(Black);
  TextColor(LightGray);
end;


procedure FichaNueva;                    { ----- Añade una ficha nueva }
begin
  ClrScr;
  TextColor(Yellow);
  NumFicha := Ultima + 1;                  { Hay que escribir al final }
  writeln('Añadiendo la ficha ',NumFicha,'.');
  TextColor(LightGray);
  with ficha do                                        { Pide cada dato }
    begin
    writeln; writeln('¿ Nombre ?');
    readln( nombre );
    writeln; writeln('¿ Dirección ?');
    readln( direccion );
    writeln; writeln('¿ Ciudad ?');
    readln( ciudad );
    writeln; writeln('¿ Código Postal ?');
    readln( cp );
    writeln; writeln('¿ Teléfono ?');
    readln( telef );
    writeln; writeln('¿ Observaciones ?');
    readln( observ );
  end;
  seek( FichAgenda, NumFicha-1 );                          { Se sitúa }
  write( FichAgenda,ficha );                      { y escribe la ficha }
  Ultima  := Ultima + 1;                           { Ahora hay una más }
end;


procedure Modifica;                  { ----- Modifica la ficha actual }
var
  temporal:string[100];           { Almacena cada valor temporalmente }
begin
  ClrScr;
  TextColor(Yellow);
  writeln('Corrigiendo la ficha ',NumFicha,'.');
  TextColor(LightGray);
  with ficha do
    begin
    writeln; writeln('¿ Nombre (',
      nombre,') ?');                       { Muestra el valor anterior }
      readln(temporal);                                { y pide el nuevo }
      if temporal<>''                                { Si tecleamos algo, }
        then nombre:=temporal;       { lo modifica (si no, no cambia) }
    writeln;
    writeln('¿ Dirección (',direccion,') ?');
      readln(temporal);
      if temporal<>'' then direccion:=temporal;
    writeln; writeln('¿ Ciudad (',ciudad,') ?');
      readln(temporal);
      if temporal<>'' then ciudad:=temporal;
    writeln; writeln('¿ Código Postal (',cp,') ?');
      readln(temporal);
      if temporal<>'' then cp:=temporal;
    writeln; writeln('¿ Teléfono (',telef,') ?');
      readln(temporal);
      if temporal<>'' then telef:=temporal;
    writeln; writeln('Observaciones (',observ,') ?');
      readln(temporal);
      if temporal<>'' then observ:=temporal;
  end;
  seek( FichAgenda, NumFicha-1 );              { Como siempre... }
```

```
  write( FichAgenda, ficha )
end;


procedure NumeroFicha;     { ----- Va a la ficha con un cierto numero }
var
  numero: word;
begin
  ClrScr;
  TextColor(Yellow);
  writeln('Saltar a la ficha con un determinado número ');
  TextColor(LightGray);
  writeln;
  writeln('¿ Qué número de ficha ?');
  readln( numero );
  if numero>0 then                          { comprueba que sea válido }
    if numero<=ultima then
      NumFicha:=numero                { si es <= que la última, salta }
    else
      NumFicha:=ultima;          { si es mayor, se queda en la última }
end;


procedure Prepara;            { ----- Inicialización de las variables }
begin                                 { y apertura/creación del fichero }
  NumFicha := 1;
  Ultima := 1;
  assign( FichAgenda, nombref );
  {$I-}                                   { Desactiva errores de E/S }
  reset( FichAgenda );                              { e intenta leer }
  {$I+}
  if ioresult <>0  then                       {Si no existen los datos}
    begin
    ClrScr;
    writeln(' No existen datos.',
      '  Pulse una tecla para introducirlos.');
    pausa;
    rewrite( FichAgenda );                                  { los crea }
    FichaNueva;                          { y obliga a añadir una ficha }
    end;
  Ultima := FileSize( FichAgenda );                { Número de fichas }
end;


begin                                 { ----- Cuerpo del programa ----- }
  Saludo;
  prepara;
  repeat
    Escribe;
    opcion:='a';
    while not (opcion in ['0'..'8']) do
      opcion := readkey;
    case opcion of
      '1': { Ficha anterior }
           if NumFicha>1 then NumFicha := NumFicha - 1;
      '2': { Ficha posterior }
           if NumFicha<ultima then NumFicha := NumFicha + 1;
      '3': { Número de ficha }
           NumeroFicha;
      '4': { Añadir una ficha }
           FichaNueva;
      '5': { Corregir la ficha actual }
           Modifica;
      end;
  until opcion='0';  { Terminar }
  Close( FichAgenda );
```

```
  ClrScr;
end.
```

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.

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 DOS, etc. Pero por ahora 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. Las unidades nos ayudan a conseguir dos cosas:

- La primera ventaja 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.

Por ejemplo: 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 procedimiento RotaPunto, p.ej., que ahora tomará de nuestra unidad "Graf3D" sino que si de repente 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.

Un ejemplo para que se entienda mejor...

```
unit miCrt1;              { Unidad que "mejora" la CRT }
```

```
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:

```
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. Vamos a mejorar ligeramente nuestra unidad, añadiéndole un procedimiento "pausa":

```
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 }
```

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

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

```
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:

```
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 resumiremos:

- Al compilar una unidad se crea un fichero **.TPU**, al que se puede acceder desde nuestros programas con dos condiciones: que empleemos la misma versión de Turbo Pascal (el formato de las TPU varía en cada versión), y que sepamos cómo es la parte pública (interface).

- Cada unidad tiene su propio **segmento de código** (esto va para quien conozca la estructura de la memoria en los PC), así que una unidad pueda 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 en el próximo tema...

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 estructuras 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 conocer 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.

Introducción.

La idea de variable dinámica está muy relacionada con el concepto de **puntero (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 ejecuta 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).

Veámoslo con un ejemplo (bastante inútil, "puramente académico") que después comentaré un poco.

```
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 }
```

Ya 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.

Arrays de punteros.

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 de ficheros de texto. 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...

```
program Lector;              { Lee ficheros de texto }

uses                                    { Unidades externas: }
```

```
   crt;                                 { Pantalla de texto y teclado }

const
   MaxLineas = 2000;                    { Para modificarlo fácilmente }

   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 leído 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.
```

Listas enlazadas.

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". 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 := 45;                {   la edad }
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 quisiéramos 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 }
```

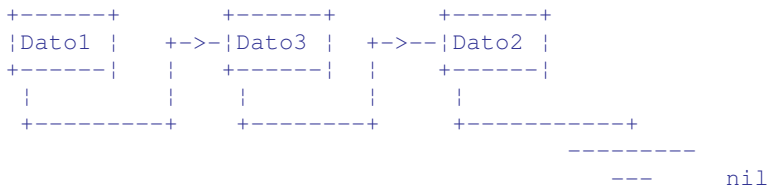
```
new (dato3);
dato3^.nombre := 'Carlos';
dato3^.direccion := 'Por ahí';
dato3^.edad := 14;
dato3^.siguiente := dato2;        { enlazamos con la siguiente }
```

```
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 "pascalero" 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:

```
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?

Árboles binarios.

Hemos visto cómo crear listas dinámicas enlazadas, y cómo podíamos ir insertando los elementos en ellas de forma que siempre estuviesen ordenadas.

Hay varios casos particulares. Sólo comentaré un poco algunos de ellos:

- 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 (salvo los malabaristas). 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.

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 ATRÁS, 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 profundizaremos más en ellas.

ACADEMIA CARTAGENA99
C/ Cartagena 99, Bº C , 28002 Madrid
91 51 51 321
academia@cartagena99.com.es

Tema 13b: ARBOLES.

En primer lugar, veamos de donde viene el nombre. 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 a un 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.

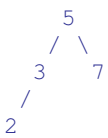
Como ejemplo, vamos a introducir en un árbol binario de búsqueda los datos 5,3,7,2,4,8,9

Primer número: 5 (directo)

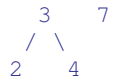
Segundo número: 3 (menor que 5)

Tercer número: 7 (mayor que 5)

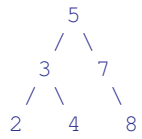
Cuarto: 2 (menor que 5, menor que 3)



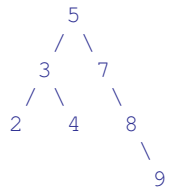
Quinto: 4 (menor que 5, mayor que 3)



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



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



¿Y qué **ventajas** tiene esto? Pues la rapidez: tenemos 7 elementos, lo que en una lista supone que si buscamos un dato que casualmente está al final, haremos 7 comparaciones; en este árbol, tenemos 4 alturas => 4 comparaciones como máximo.

Y si además hubiéramos "**equilibrado**" el árbol (irlo recolocando, de modo que siempre tenga la menor altura posible), serían 3 alturas.

Esto es lo que se hace en la práctica cuando en el árbol se va a hacer muchas más lecturas que escrituras: se reordena internamente después de añadir cada nuevo dato, de modo que la altura sea mínima en cada caso.

De este modo, el número máximo de comparaciones que tendríamos que hacer sería log2(n), lo que supone que si tenemos 1000 datos, en una lista podríamos llegar a tener que hacer 1000 comparaciones, y en un árbol binario, log2(1000) => 10 comparaciones como máximo. La ganancia es evidente.

No vamos a ver cómo se hace eso de los "equilibrados", 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;
```

Un **comentario**: esta última rutina es peligrosa, porque indicamos que "punt" está libre y después miramos cual es su hijo izquierdo (después de haber borrado la variable). Esto funciona en Turbo Pascal , porque marca esa zona de memoria como disponible pero no la borra físicamente, pero puede dar problemas con otros compiladores o si se adapta esta rutina a otros lenguajes (como C). Una forma más segura de hacer lo anterior sería:

```
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;
```

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?

Apéndice 1: Otras órdenes no vistas.

A lo largo del curso ha habido órdenes que no hemos tratado, bien porque no encajasen claramente en ningún tema, o bien porque eran demasiado avanzadas para explicarlas junto con las más parecidas.

En primer lugar vamos a ir viendo las que podían haber formado parte de temas anteriores, y después las que faltan. Tampoco pretendo que esto sea una recopilación exhaustiva, sino simplemente mencionar algunas órdenes interesantes que parecían haberse quedado en el tintero.

Los **temas** que se van a comentar son:

- Bucles: break, continue.
- Saltos: goto, label.
- Punteros: getmem, freemem, pointer.
- Fin del programa: exit, halt.
- Números aleatorios: rnd, randomize.
- Funciones: abs, sin, cos, arctan, round, trunc, sqr, sqrt, exp, ln, odd, potencias.

Bucles: break, continue.

En Turbo Pascal 7.0, tenemos dos órdenes extra para el control de bucles, tanto si se trata de "for", como de "repeat" o "until". Estas órdenes son:

- **Break**: sale del bucle inmediatamente.
- **Continue**: pasa a la siguiente iteración.

Un ejemplo "poco útil" que use ambas podría ser escribir los números pares hasta el 10 con un for:

```
for i := 1 to 1000 do            { Nos podemos pasar }
   begin
   if i>10 then break;           { Si es así, sale }
   if i mod 2 = 1 then continue; { Si es impar, no lo escribe }
   writeln( i );                 { Si no, lo escribe }
   end;
```

Esto se puede imitar en versiones anteriores de TP con el temible "goto", que vamos a comentar a continuación:

Goto.

Es una orden que se puede emplear para realizar saltos incondicionales. Su empleo está bastante desaconsejado, pero en ciertas ocasiones, y se lleva cuidado, puede ser útil (para salir de bucles fuertemente anidados, por ejemplo, especialmente si no se dispone de las dos órdenes anteriores.

El formato es

goto etiqueta

y las etiquetas se deben declarar al principio del programa, igual que las variables. Para ello se usa la palabra "**label**". Vamos a verlo directamente con un ejemplo:

```
label uno, dos;


var
   donde: byte;


begin
```

```
      writeln('¿Quiere saltar a la opción 1 o a la 2?');
      readln (donde);
      case donde of
         1: goto uno;
         2: goto dos;
         else writeln('Número incorrecto');
         end;
      exit;


   uno:
      writeln('Esta es la opción 1.  Vamos a seguir...');
   dos:
      writeln('Esta es la opción 2.');
   end.
```

Punteros: getmem, freemem, pointer.

Habíamos hablado de "new" y "delete", en las que el compilador decide la cantidad de memoria que debe reservar. Pero también podemos obligarle a reservar la que nosotros queramos, lo que nos puede interesar, por ejemplo, para leer cadenas de caracteres de longitud variable.

Para esto usamos "**getmem**" y "**freemem**", en los que debemos indicar cuánta memoria queremos reservar para el dato en cuestión. Si queremos utilizarlos como new y dispose, reservando toda la memoria que ocupa el dato (será lo habitual), podemos usar "sizeof" para que el compilador lo calcule por nosotros:

```
    type
      TipoDato = record
         Nombre: string[40];
         Edad: Byte;
         end;


   var
      p: pointer;


   begin
      if MaxAvail < SizeOf(TipoDato) then
         Writeln('No hay memoria suficiente')
      else
         begin
         GetMem(p, SizeOf(TipoDato));
         { Trabajaríamos con el dato, y después... }
         FreeMem(p, SizeOf(TipoDato));
         end;
   end.
```

Por cierto, "**pointer**" es un puntero genérico, que no está asociado a ningún tipo concreto de dato. No lo vimos en la lección sobre punteros, porque para nosotros lo habitual es trabajar con punteros que están relacionados con un cierto tipo de datos.

Exit, halt.

En el tema 8 (y en los ejemplos del tema 6) vimos que podíamos usar exit para salir de un programa.

Esto no es exacto del todo: **exit** sale del bloque en el que nos encontremos, que puede ser un procedimiento o una función. Por tanto, sólo sale del programa si ejecutamos exit desde el cuerpo del programa principal.

Si estamos dentro de un procedimiento, y queremos abandonar el programa por completo, deberíamos hacer más de un "exit". Pero también hay otra opción: la orden "**halt**".

Halt sí que abandona el programa por completo, estemos donde estemos. Como parámetro se le puede pasar el "Errorlevel" que se quiere devolver al DOS, y que se puede leer desde un fichero batch.

Por ejemplo: "halt" o "halt(0)" indicaría una salida normal del programa (sin errores), "halt(1)" podría indicar que no se han encontrado los datos necesarios, etc.

Números aleatorios.

Si queremos utilizar números aleatorios en nuestros programas, podemos emplear la función "**random**". Si la usamos tal cual, nos devuelve un número real entre 0 y 1. Si usamos el formato "random(n)", lo que devuelve es un número entero entre 0 y n-1.

Para que el ordenador comience a generar la secuencia de números aleatorios, podemos usar "**randomize**", que toma como semilla un valor basado en el reloj, lo que supone que sea suficientemente aleatorio:

```
var i:integer;


begin
   Randomize;
   for i := 1 to 50 do
      Write (Random(1000), ' ');
end.
```

Algunas Funciones.

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

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

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

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

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

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

Apéndice 2: Palabras reservadas de Turbo Pascal (ordenadas alfabéticamente):

Son las palabras clave que no se pueden redefinir:

and: Para encadenar comparaciones ("y"), u operador de bits (producto).

array: Matriz o vector a partir de un tipo base.

asm: Accede al ensamblador integrado.

begin: Comienzo de una sentencia compuesta.

case: Elección múltiple según los valores de una variable.

const: Comienzo de la declaración de constantes.

constructor: Inicializa un objeto que contiene métodos virtuales.

destructor: Elimina un objeto que contiene métodos virtuales.

div: División entera.

do: Ordenes a realizar en un bucle "for" o en un "while".

downto: Indica el final de un bucle "for" descendente.

else: Acción a realizar si no se cumple la condición dada en un "if".

end: Fin de una sentencia compuesta.

file: Tipo de datos: fichero.

for: Bucle repetitivo. Se usa con "to" o "downto", y con "do".

function: Definición de una función.

goto: Salta a una cierta etiqueta dentro del programa.

if: Comprobación de una condición. Se usa con "then" y "else".

implementation: Parte privada de una unidad: detalle que los procedimientos y funciones que la forman, junto con los datos no accesibles.

in: Comprueba la pertenencia de un elemento en un conjunto.

inherited: Accede al método de igual nombre del objeto padre.

inline: Inserta instrucciones en código máquina dentro del programa.

interface: Parte pública de una unidad: datos accesibles y cabeceras de los procedimientos y funciones.

label: Etiqueta a la que se puede saltar con "goto".

mod: Resto de la división entera.

nil: Puntero nulo.

not: Operador lógico ("no") y de bits.

object: Declaración de un objeto.

of: Indica el tipo base de un "array", "string", "set" o "file".

or: Operador lógico ("ó") y de bits (suma).

packed: Tipo de array. No necesario en Turbo Pascal.

procedure: Definición de un procedimiento.

program: Nombre de un programa.

record: Tipo de datos: registro.

repeat: Repite una serie de órdenes hasta ("until") que se cumple una condición.

set: Tipo conjunto.

shl: Desplazamiento de bits hacia la izquierda.

shr: Desplazamiento de bits hacia la derecha.

string: Tipo de datos: cadena de caracteres.

then: Acción a realizar si se cumple una condición dada por "if".

to: Indica el final de un bucle "for" ascendente.

type: Comienzo de la declaración de tipos.

unit: Comienzo (y nombre) de la definición de una unidad.

until: Condición de fin de un "repeat".

uses: Lista las unidades que va a utilizar un programa u otra unidad.

var: Comienzo de la declaración de variables.

while: Repite una orden mientras se dé una condición.

with: Para acceder a los campos de un "record" sin tener que nombrarlo siempre.

xor: Operación de bits (suma exclusiva).