Notas
Introducción
Principal
Capítulo 2º
Capítulo 3º
Capítulo 4º
Capítulo 5º
Capítulo 6º
Capítulo 7º
Capítulo 8º

Capítulo 1º

ESTRUCTURA DE UN PROGRAMA EN PASCAL

Hecha la introdución ahora empezaremos a meternos en materia. Esta vez ya atacamos fuerte y nos metemos con cómo se "monta" un programa en Pascal.

Al Pascal no se le llama programación estructurada porque sí. A la hora de ponerse a hacer un programa hay que seguir una estructura bastante estricta.

El esquema general de un programa sería este:

```
* Nombre del programa
* USES
* Definición de tipos globales
* Definición de constantes globales
* Definición de variables globales
* Procedimientos
      * Declaración
      * Tipos locales
      * Constantes locales
      * Variables locales
      * Codigo del procedimiento
* Código principal
```

Así al principio puede parecer mucha cosa, pero si os digo que por muy complicado que sea el programa que hagais NUNCA vais a usar más elementos de los que he puesto aquí... ¿entonces qué? ;) Lo que sí que podremos usar, sobretodo al principio, son menos elementos.

Por ahora veamos por partes en qué consiste cada "trozo" de programa.

NOMBRE DEL PROGRAMA

Pues algo tan tonto como lo que podeis estar pensando. Consiste en darle un nombre al programa. Se hace poniendo al principio del programa una linea como esta:

```
Program Programa_Que_Dice_Hola;
```

"Program" es una palabra reservada del Pascal que sirve para indicar que a continuación ponemos el nombre del programa.

El nombre del programa NO puede contener espacios ni algunos otros caracteres "raros" como, por ejemplo, la "Ñ".

El punto y coma final es IMPRESCINDIBLE. En Pascal ese punto y como se pone como "separador" de instruciones. Así, después de cada instrución pondremos un punto y coma. Ya lo iremos viendo.

USES

Uses es la palabra reservada de Pascal para incluir librerías. Como ya he dicho antes, una librería no es más que un conjunto de funciones, que nosotros mismo podemos hacernos. Cualquier compilador de Pascal viene con unas librerías standar que traen funciones para trabajar con archivos, memoria, directorios, servicios del Dos, servicios de video, etc...

Sin esas librerías no podríamos hacer prácticamente nada. Siempre podríamos hacernoslas nosotros, pero sería un trabajo inmenso.

De entre todas las librerías del Pascal hay una que es especial, la librería SYSTEM.TPU. Es especial porque contiene las funciones más básicas, como pueden ser las que nos sirven para escribir en la pantalla, y esta librería se incluye SIEMPRE en nuestros programas.

Otras librerías muy utilizadas con CRT.TPU, DOS.TPU y GRAPH.TPU, por ejemplo, pero de esto ya nos encargaremos cuando hablemos específicamente de las librerías.

Para incluir una o varias librerías hay que añadir una linea de este estilo al programa:

```
USES Crt, Dos, Graph;
```

DEFINICION DE TIPOS GLOBALES

En esta seción de declaran las estructuras globales que vamos a utilizar. Se les llama globales porque las estructuras que definamos aquí podrán ser utilizadas en cualquier punto del programa. Los tipos de datos son una de las cosas completas de Pascal, hablaremos de ellos en la próxima entrega. ;)

DEFINICION DE CONSTANTES GLOBALES y DEFINICION DE VARIABLES GLOBALES

De las constantes y las variables hablaremos en el próximo capítulo. ;)

PROCEDIMIENTOS

Un procedimiento podríamos definirlo como un trozo de programa "autónomo". Es lo que antes he llamado una función. Las librerías están llenas de pedazos de programa autónomos de estos. ;)

Aunque en un principio puede parecer una chorrada tener que ir partiendo los programa en procedimientos, esto resulta muy útil para hacer un programa. Para dejarlo claro pondré un ejemplo sencillo:

Pongamos que tenemos que hacer un programa que dibuje muchos cuadrados. Cada cuadrado se dibuja a base de 4 lineas. Y cada linea se dibuja a base de muchos puntos colocados en linea.

Podríamos hacer un programa como este:

```
Dibujar_Punto en 0,0
```

```
Dibujar_Punto en 1,0
Dibujar_Punto en 2,0
.......
Dibujar_Punto en 20,20
```

Este programa lo que haría sería dibujar todos los puntos de todos los cuadrados uno a uno. Si nos equivocamos en un sólo punto más nos vale volver a empezar porque seguro que no encontraremos en qué linea nos hemos equivocado. Nos quedaría un código muy "enborronado".

Ahora pongamos que para hacer el mismo programa declaramos un procedimiento que dibuje una linea entera. Así, para dibujar un cuadrado ahora ya sólo tenemos que poner:

```
Dibuja_Linea de 0,0 a 20,0
Dibuja_Linea de 0,0 a 0,20
Dibuja_Linea de 20,0 a 20,20
Dibuja_Linea de 0,20 a 20,20
```

Y ya tendríamos dibujado un cuadrado. Y si ahora al trozo de programa que acabamos de escribir lo llamamos Dibuja_Cuadrado, podríamos poner:

```
Dibuja_Cuadrado de 0,0 a 20,20
```

De esta manera el procedimiento Dibuja_Cuadrado "llamaría" 4 veces al procedimiento Dibuja_Linea para dibujar las 4 lineas correspondientes. Y cada procedimiento Dibuja_Linea llamaría a Dibuja_Punto las veces necesarias para dibujar la linea en cuestión. Vendría a ser como si se repartiesen la faena.

Así, el mismo programa del principio lo podríamos escribir así:

```
Dibuja_Cuadrado de 0,0 a 20,20
Dibuja_Cuadrado de 10,10 a 30,30
.......
```

Espero que con este ejemplo quede demostrada la utilidad de dividir un programa en procedimientos. Ya iremos viendo más adelante cómo se hace concretamente esto.

CODIGO PRINCIPAL

Se trata del cuerpo del programa. Normalmente consistirá sólo en llamadas a los procedimientos que hayamos definido.

PROGRAMA DE EJEMPLO

Para empezar a concretar las cosas y para que os empeceis a acostumbrar a los listados en Pascal os pongo aquí un pequeño programa de ejemplo.

```
Program Programa_Que_Dice_Hola;
Uses Crt;                    { incluir la librería CRT }

Const                        { declaración de constantes globales }
  Texto = 'Hola!!! ¿Qué tal? ;)';
```

```
Procedure Escribe_Texto;      { Declaración del procedimiento }
Begin
  WriteLn(Texto);             { escribe la constante "Texto" }
End;

Begin                         { inicio del código principal }
Escribe_Texto;                { llamada al procedimiento }
End.
```

Y esto es todo por hoy. En la próxima entrega nos empezaremos a meter con las variables, las contastes, los tipos y las estructuras de datos, que ya empieza a ser un plato fuerte. Por hoy ya sólo queda el mini-glosario que incluiré en todos los faSículos que crea que tienen palabras que necesitan explicación. ;)

GLOSARIO:

* Palabra reservada: como su nombre indica, es una palabra que en un determinado lenguaje ha sido reservada para realizar una función concreta. Una palabra reservada sólo puede ser usada para la finalidad que se le da en el lenguaje en cuestión y usarla de otro modo (intentando declararla como variable o como procedimiento, por ejemplo) dará un error de compilación.

En el próximo capítulo hablaremos de las variables que se utilizan para trabajar con ficheros y de las variables "compuestas".

por ComaC

Notas
Principal
Capítulo 1º
Capítulo 2º
Capítulo 3º
Capítulo 4º
Capítulo 5º
Capítulo 6º
Capítulo 7º
Capítulo 8º

Capítulo 0º

Introdución

Venga! Tal y como prometí aquí está la primera entrega del curso de Pascal. Ya que es el primer faSículo y dado que estamos en exámenes, empezaré flojito. Por hoy sólo definiremos algunos conceptos y daremos una introdución a lo que es el Pascal.

Algunas cosas antes de empezar:

1º Si alguien tiene pensado guardar los mensajes del cursillo para luego subirlo a la BBS o para tenerlo guardado, que no lo haga, porque ya lo haré yo mismo. (y según como incluiré también imágenes para aclarar algunas cosas o alguna otra cosa interesante)

2º El cursillo consiste en: yo pongo un mensaje, tú te lo lees y lo aprendes. Pero no me gustaría que se limitara sólo a eso. Yo NO voy a proponer ningún problema para que lo resolvais ni voy a proponer ningún exámen. Si quereis algo de este estilo lo teneis que pedir. Estoy abierto a cualquier tipo de actividad que ayude a entender y asimilar el cursillo, pero teneis que ser vosotros los que lo pidais. (Si quereis hacer algo pero no sabeis qué, pues me lo deciis y ya os propondré algo)

Creo que con esto ya he dicho todo lo que quería decir antes de comenzar. Así que EMPEZAMOS!!!

CAPITULO 0º: Introdución

Este capítulo no contiene nada especialmente importante. Mi recomendación es que lo leais 2 o 3 veces, que lo entendais bien y que os quedeis con una visión global de lo que es el Pascal, lo que es un Compilador, etc... No os fijeis en los detalles sino sólo en lo más general.

TÉRMINOS QUE VAMOS A USAR

Por ahora os pondré sólo unos cuantos términos muy básicos que nos van a salir sin parar.

• Código Fuente: es lo que nosotros escribimos. Lo que llamaríamos normalmente "el programa". Por ejemplo, el código fuente de un programa que escribiera "Hola!" en la pantalla y volviera al DOS sería este:

```
Begin
Write('Hola!');
End;
```

¿Y qué se puede hacer con un código fuente? Pues básicamente se pueden hacer dos cosas: interpretarlo o compilarlo.

• Interpretación: consiste en que un programa hace de intérprete entre en código fuente y el microprocesador. Esto és: para el ejemplo de arriba, un intérprete leería la instrucción "Write('Hola!);" y sabría que a esa instrucción le corresponde la acción de escribir "Hola!", así que el intérprete escribiría "Hola!". Pero atención! En este caso no es el programa el que escribe sino el intérprete.
La interpretación es muy común en BASIC, por ejemplo. Un programa interpretado siempre será bastante más lento que un programa compilado, ya que el intérprete ha de leer la instrución, procesarla, transformarla en una cadena de código que entienda el microprocesador y ejecutarla.

• Compilación: consiste en convertir el código fuente en un ejecutable que pueda funcionar por sí mismo. Lo que hace el compilador es sustituir las instruciones que nosotros hemos escrito por trozos de código que realizan la función. ¿Y de dónde salen estos trozos de código? Algunos salen de las librerías, pero los principales los escribió el autor del compilador en ASM. (en algunos compiladores estos códigos están en C) Así, a "groso modo" podemos decir que programar no es más que hacer un collage con el código que escribió el autor del compilador.

• Código objeto: es el archivo resultante de la compilación. No tiene porqué ser necesariamente un EXE o un COM. También puede ser un .OBJ (en C o ASM) o un .TPU (como librería de Pascal). Por ahora nosotros sólo haremos EXEs.

• Librería: se trata de un archivo que contiene funciones ya hechas y cada una asignada a un nombre, de manera que podemos utilizarlas desde nuestro programa sin tener que escribirlas. El Turbo Pascal trae una cuantas librerías de serie (como cualquier compilador) y también nosotros podemos hacer las nuestras. Pero esto lo dejamos para el capítulo de cómo hacer librerías.

• IDE: (Integrated Developement Envoirement = Entorno integrado de desarrollo). Long time ago para hacer un programa necesitabas un editor de texto para escribir el código fuente y un compilador.

Escribías el programa, salias del editor, compilabas, lo probabas y volvías al editor para continuar. Ya que eso era un tanto engorroso aparecieron los IDE. (aunque yo para programa en ASM lo sigo haciendo de esta manera :m) El IDE es un programa que sirve de editor, compilador y depurador a la vez. Además de traer un sistema de ayuda muy útil y otras tantas cosas. De esta manera es mucho más cómodo programar. Podemos hacerto todo desde el IDE. En nuestro caso, el IDE es lo que nos aparece al ejecutar el TURBO.EXE. ;) El Turbo Pascal también tiene un compilador "a la antigua" o, como debería decirse, un compilador desde la linea de comandos. El TPC.EXE. Para compilar el programa HOLA.PAS escribiríamos: "TPC HOLA.PAS"

INTRODUCION AL PASCAL

El Pascal es un lenguaje de propósito general. Esto quiere decir que sirve para cualquier cosa. En contraposición a lenguajes como CLIPPER, que está orientado principalmente al manejo de bases de datos.

También es un lenguaje de programación estructurada. Esto quiere decir que en Pascal no podemos poner las cosas como nos dé la gana y dónde nos dé la gana, sino que tenemos que seguir algunas reglas. Al principio puede parecer imposible programar siguiendo estas reglas, pero una vez acostumbrados vereis que es mejor programar así.

Los elementos más importantes de la programación estructurada son las variables y los procedimientos. Las variables tienen que definirse antes de usarse (cosa que no era necesario en BASIC) y además hemos de decir de qué tipo van a ser. En BASIC, por ejemplo, podemos poner en cualquier punto del programa una instrución del tipo "A = 2" y le asignaremos a la variable A el valor 2, sin importarnos si la variable A es de 8 bits o de 16 o con signo o sin signo. (Si no consigues acabar de entender esto no te preocupes, de las variables ya hablaremos detenidamente cuando llegue si momento ;)

El otro elemento he dicho que eran los procedimientos. ¿Qué es un procedimiento? Respuesta fácil y sencilla: un pedazo de código. Al principio puede parecernos incómodo tener que partir un programa en cachos en vez de ponerlo todo seguido, tal y como podría hacerse en BASIC, pero en cuanto veamos algún ejemplo os dareis cuenta de que es muy práctico. Realmente un procedimiento es bastante más completo que "un pedazo de código", pero por ahora estamos en la introdución y lo que interesa es que saqueis una visión global de lo que es el pascal. Cuando vayamos pasando por cada uno de los puntos en cuestión ya veremos las cosas con todos sus detalles.

Para acabar, me queda decir que a la programación estructurada también se la llama "programación sin GOTO". Y esto es porque NO se usa NUNCA dicha instrución. A los que hayais programado en BASIC tal vez os parezca imposible programar sin esa instrución (tal y como

pensé yo) pero no es nada difícil. Para suplir el GOTO tenemos la estructuración de los procedimientos y diferentes bucles iterativos. (cosas que explicaré en su momento)

El Turbo Pascal incluye la función GOTO y puede utilizarse, pero yo no la voy a explicar. Al menos no hasta el final del curso.

Y esto es todo por hoy. El próximo lunes más. ;)
Empezaremos con la estructura general de un programa en Pascal y hablaremos un poco a cerca de las librerías básicas.

Cualquier duda o sujerencia que tengais podeis preguntármela por el area. (con lo de sujerencia me refiero a si quereis problemas para prácticar (aunque por ahora todabía no podemos) o cualquier otra cosa)

por ComaC

Notas
Introdución
Capítulo 1º
Principal
Capítulo 3º
Capítulo 4º
Capítulo 5º
Capítulo 6º
Capítulo 7º
Capítulo 8º

Capítulo 2º

VARIABLES (1ª PARTE)

Las variables son el "sitio" donde podemos guardar los datos que el programa necesite. En Pascal las variables hay que definirlas antes de usarlas y hay que aplicares un tipo.

Esto significa que al principio del programa tenemos que decir qué variables vamos a usar y qué tipo de datos van a contener. Para decla- rar una variable lo hacemos después de la palabra reservada "Var" y de la siguiente forma:

```
Var
  Variable1 : Tipo;
  Variable2 : Tipo;
  Variable3, Variable4 : Tipo;
```

Si dos o más variables van a ser del mismo tipo podemos ponerlas en lineas separadas o en una misma linea separadas por comas.

Por hoy veremos sólo los tipos simples

BYTE (8 bits, 0..255)

En tipo byte define una variable que ocupará 8 bits de memoria (un byte) y esta variable podrá contener valores desde el 0 hasta el 255, ambos incluidos.

SHORTINT (8 bits, -128..127)

Define una variable que ocupa 8 bits y puede tomar valores de -128 a 127. Con estas variables hay que tener cuidado porque 127+1=-128.

CHAR (8 bits, caracter)

Define una variable de 8 bits pero que no contiene valores sino cualquier caracter ASCII.

BOOLEAN (8 bits, verdadero-falso)

Define una variable de 8 bits que sólo puede contener dos valores: TRUE (verdadero) o FALSE (falso).

INTEGER (16 bits, -32768..32767)

Define una variable de 16 bits (2 bytes) que puede contener valores desde -32768 hasta 32767. Sucede lo mismo que con el tipo Shortint: 32767+1=-32768.

WORD (16 bits, 0..65535)

Define una variable de 16 bits que soporta valores de 0 a 65535.

LONGINT (32 bits, -2Gb..(2Gb-1))

Define una variable de 32 bits (4 bytes) que puede contener valores desde -2147483648 hasta 2147483647.

REAL, SINGLE, DOUBLE (punto flotante)

Estos tipos son también numéricos pero son un tanto peculiares. Están pensados para poder trabajar con números muy grandes pero no tienen mucha precisión, es decir 4+1 no siempre será 5. Una característica de estos tipos de datos, además de poder trabajar con números de hasta 1.1e4932, es que con ellos podemos usar números con decimales. A una variable SINGLE le podemos asignar el valor 1.5 (y tal vez lo que se guarde realmente sea 1.49, como ya he dicho) pero en cambio no le podemos asignar ese mismo valor a una variable de tipo INTEGER, por ejemplo.

Las operaciones con los tipos SINGLE y DOUBLE se realizarán bastante más rápido si se ejecutan en un ordenador con coprocesador matemático y se ha activado la opción del compilador para usarlo. Los REAL en cambio no irán más rápido ya que no son un formato standar sino que es un formato que se ha inventado Borland para sus compiladores de Pascal.

STRING (cadena)

Define una variable de 256 bytes que puede contener una cadena de hasta 255 caracteres. (Una frase de hasta 255 letras, para entendernos ;)

PUNTEROS

Los punteros los trataremos más adelante en un capítulo que ellos solos, ya que es un tema muy extenso. Por ahora prescindiremos de ellos.

Por hoy ya está. Este es uno de los capítulos que convendría imprimirlos y tenerlos delante de las narices a la hora de hacer prácticas. Para acabar os pongo un ejemplo para que veais cómo se declararían y usarían las variables de cada tipo, aunque para entender bien el programa hacen falta los capítulos sobre asignaciones y sobre funciones de librería, pero para hacerse una idea yo creo qye ya irá bien. ;)

```
Program Programa_Demo_De_Variables;

Const                   { Definimos constantes, como ya habíamos visto }
  Numero = 8;
  Cadena = 'Esto es una cadena ;)';

Var                     { Declaramos unas cuantas variables... }
  Tipo_Int  : Integer; { Usamos el Integer como ejemplo para: BYTE,
                         SHORTINT, INTEGER ,WORD y LONGINT }
  Tipo_Bool : Boolean;
```

```
  Tipo_Real : Single;  { Usamos el tipo SINGLE como ejemplo para: REAL,
                         SINGLE y DOUBLE }
  Tipo_Char : Char;

  Tipo_Cad  : String;

Begin
Tipo_Int := Numero;    { Asignamos el valor 8 a Tipo_Int }
WriteLn(Tipo_Int);     { Sacamos por pantalla el contenido de Tipo_Int }

Tipo_Int := 4;         { Asignamos el valor 4 a Tipo_Int }
WriteLn(Tipo_Int);     { Sacamos por pantalla el contenido de Tipo_Int }

Tipo_Bool := True;     { Asignamos el valor True a Tipo_Bool }
WriteLn(Tipo_Bool);    { Sacamos por pantalla el contenido de Tipo_Bool }

Tipo_Real := 87.724;   { Asignamos el valor 87.724 a Tipo_Real }
WriteLn(Tipo_Real:0:3);{ Sacamos por pantalla el contenido de Tipo_Real }

Tipo_Char := 'A';      { Asignamos la letra 'A' a Tipo_Char }
WriteLn(Tipo_Char);    { Sacamos por pantalla el contenido de Tipo_Char }

Tipo_Cad := Cadena;    { Asignamos "Esto es una cadena ;)" a Tipo_Cad }
WriteLn(Tipo_Cad);     { Sacamos por pantalla el contenido de Tipo_Cad }

End.
```

En el próximo capítulo hablaremos de las variables que se utilizan para trabajar con ficheros y de las variables "compuestas".

por ComaC

Notas
Introducción
Capítulo 1º
Capítulo 2º
Principal
Capítulo 4º
Capítulo 5º
Capítulo 6º
Capítulo 7º
Capítulo 8º

Capítulo 3º

VARIABLES (2ª PARTE)

En el último capítulo empezamos a explicar los tipos básicos de variables. En este seguimos con las variables para control de archivos y los tipos compuestos.

TEXT

Una variable de tipo TEXT declara una variable que servirá para contener los datos de un archivo. Esos datos los necesita el programa para controlar el archivo, pero nosostros nunca los trataremos. Las funciones de la librería se encargan de modificar/tratar todo lo necesario en este tipo de variables. Así, por ejemplo, al llamar a la función Reset, ésta rellenará la variable que le indiquemos con los datos del archivo que hemos abierto.

Las variables TEXT sirven para trabajar con archivo de texto. Todo lo referente a la gestión de archivos lo trataremos más adelante, cuando hablemos de las funciones de la librería para trabajar con archivos.

FILE

Una variable de tipo FILE tiene exactamente la misma función que una TEXT sólo que en vez de tratarse de archivos de texto nos permitirá trabajar con archivo binarios. Podemos definir variables FILE de varios tipos, dependiendo del tipo de elementos que integren el archivo o incluso podemos definir una variable FILE sin tipo, pero esto, como ya he dicho, lo trataremos más adelante.

ARRAY

Define una variable que es un conjunto de elementos de un tipo. Es decir, si definimos un array de CHARs, por ejemplo, la variable en cuestión no contendrá un solo elemento de tipo CHAR sino varios, la cantidad que nosotros le indiquemos. Pongamos un ejemplo:

```
Var
  Grupo : Array [1..10] of Integer
```

Este código define la variable Grupo como un array que contiene 10 datos de tipo Integer. Para acceder a uno de esos datos lo hacemos indicando el número de orden entre corchetes. Así, para acceder al dato nº 6 haríamos esto:

```
Grupo[6] := 9;
```

O para asignar a el elemento nº 6 el mismo valor que el elemento 9 podemos hacer esto:

```
Grupo[6] := Grupo[9];
```

Podemos imaginarnos un ARRAY como una fila de elementos del tipo que hemos especificado. El array de la variable Grupo sería esto:

Nº de orden:	1	2	3	4	5	6	7	8	9	10
Elemento:										

Pero también podemos definir un array como el siguiente:

```
Tabla : Array [1..10,1..10] of Byte;
```

En este caso se dice que tenemos un array bidimensional. Esto és: ahora el array ya no se representaría como un fila sino como una tabla de 10 filas y 10 columnas. Así:

	1	2	3	4	5	6	7	8	9	10
1										
2										
3						XX				
4										
5		YY								
6										
7										
8										
9										
10										

De esta forma, con Tabla[5,2] nos estaríamos refiriendo al elemento que he marcado con "YY" y con Tabla[3,6] nos referiríamos al elemento marcado con "XX".

También pueden hacerse array de más dimensiones símplemente añadiendo el tamaño de la nueva dimensión mediante comas. Para un array de 3 dimensiones lo declararíamos así:

```
Tabla3D : Array [1..10,1..10,1..10] of Word;
```

ESTRUCTURAS

Una estructura define una variable formada por varios campos, y cada campo puede ser de un tipo diferente. Por ejemplo, pongamos que en una variable queremos guardar los datos de un amigo. Podríamos definir una estructura como esta:

```
Var
  Amigo : Record
            Nombre   : String;
            Edad     : Byte;
            DNI      : Longint;
            Telefono : Longint;
```

```
                    End;
```

Con esto declaramos la variable amigo, que tiene 4 campos. El campo Nombre es una cadena de caracteres (String) el campo Edad puede contener valores de 0 a 255 y los campos DNI y Telefono son Longint para poder almacenar números grandes.

Para acceder a los campos lo hacemos escribiendo su nombre tras el de la variable separando ambos mediante un punto. Para rellenar la estructura podríamos hacer:

```
Amigo.Nombre   := 'Perico el de los palotes';
Amigo.Edad     := 29;
Amigo.DNI      := 12345;
Amigo.Telefono := 3589325;
```

En las estructuras podemos incluir CUALQUIER tipo de datos. Incluidos los arrays e incluso otras estructuras. Así, podemos declarar una variable con:

```
Amigo_Cambia_Cromos : Record
                        Nombre : String;
                        Edad   : Byte;
                        Tengui : Array [1..300] of Word;
                        Falti  : Array [1..300] of Word;
                        Colecs : Record
                                   BolaDeDrac : Boolean;
                                   SonGoku    : Boolean;
                                   DragonBall : Boolean;
                                   End;
                      End;
```

De esta manera, en Amigo_Cambia_Cromos.Tengui[20] guardaríamos si el amigo tiene o no el cromo 20, y en Amigo_Cambia_Cromos.Colecs.SonGoku pondríamos TRUE o FALSE dependiendo de si tiene hecha la coleción de cromos del Goku o no.

Y más útil todabía es poder definir una estructura dentro de un array. Es decir, algo como esto:

```
Amigos : Array [1..100] of Record
           Nombre : String;
           Edad   : Byte;
           Colecs : Record
                      SonGoku      : Array [1..300] of Boolean;
                      BulmaEnBolas : Array [1..300] of Boolean;
                      End;
           End;
```

Con esto, definimos una "lista" en la que podemos guardar los datos de 100 amigos diferentes. Podemos guardar su Nombre, su Edad y los cromos que tienen de las dos coleciones.

TIPOS DE DATOS

Para acabar con las estructuras veremos cómo se define un tipo de datos. Los tipos de datos se definen a continuación de la palabra reservada "Type". (al igual que las variables bajo "Var" o las constantes bajo "Const") Y a un tipo que nosotros definamos le podemos asignar cualquier tipo de variable de los que hemos visto. Definir el tipo siguiente, por

ejemplo, no tiene gran utilidad:

```
Type
  Octeto = Byte;
```

Con él podríamos definir una variable como esta:

```
Var
  Edad : Octeto;
```

Y sería equivalente a escribir:

```
  Edad : Byte;
```

Pero cuando se trata de estructuras esto sí que puede ser muy útil, sobretodo para crear un código claro y legible. Podemos definir un tipo que contenga la estructura que queramos y a partir de entonces trabajar con ese tipo de datos. Por ejemplo:

```
Type
  Estructura = Record
                 Nombre : String;
                 Edad   : Byte;
                 End;

Var
  Variable1 : Estructura;
  Variable2 : Array [1..10] of Estructura;
```

OTROS

Existen más tipos de datos (el PChar, por ejemplo) pero de momento no los explicaré porque o bien no son básicos o bien están incluidos en el Turbo Pascal por motivos de compatibilidad.

VARIABLES ESTATICAS

Todas las variables que hemos visto hasta ahora son variables estáticas. Esto quiere decir que las variables existen ya al empezar el programa y no cambian durante la ejecución de este. Podemos cambiar el contenido de las variables, pero no podremos cambiar el tamaño ni el tipo ni nada de eso.

Estas variables se almacenan en una zona reservada para datos. En el segmento de datos, concretamente. No entraré en la explicación de lo que es un segmento, pero en un segmento sólo caben 64 Kbs. (que son 65536 bytes y NO 64000 bytes) Por lo tanto, si escribimos una declaración de variables como esta, el compilador nos dará un error:

```
Var
  Tabla1 : Array [1..40000] of Byte;
  Tabla2 : Array [1..40000] of Byte;
```

¿Por qué da el error? Pues porque para almacenar todas esas variables necesitamos 80000 bytes, que son más de los 65536 que nos proporciona el segmento de datos. Entonces ¿sólo podemos usar 64 Kbs de memoria en Pascal? ¿Qué pasa con las 32 megas de Ram que tiene tal ordenador? ¿Para qué me mato yo en conseguir liberar 630 Kbs de memoria convencional si

ahora resulta que sólo puedo usar 64 Kbs? Bueno, para algo están las variables dinámicas.

VARIABLES DINAMICAS

Las variables dinámicas son, para decirlo de manera simple, los punteros. La memoria que necesita un puntero no se reserva desde el comienzo del programa sino que seremos nosotros quienes decidiremos cuándo ha de asignarse memoria a un determinado puntero. (También deberemos acordarnos de liberarla una vez hayamos terminado)

Además, la memoria para los punteros no se saca del segmento de datos sino que se saca de la memoria que quede libre después de cargar el programa. Si usamos un compilador para modo real, podremos usar lo que quede de memoria hasta los 640 Kbs, y si usamos un compilador para modo protegido podremos usar toda la memoria que quede libre hasta el final de ésta. (que pueden ser 4, 8, 16... la Ram que tengamos) Como ejemplo en el cursillo usaremos un compilador en modo real (El Turbo Pascal 7)

Cada puntero tiene su propio segmento, por lo tanto a un puntero no le podremos asignar más de 64Kbs de memoria, pero como no es necesario que los punteros tengan el mismo segmento, podemos usar toda la memoria que quede libre a base de asignar una parte a varios punteros. Esto es un tema complicado que trataremos en el capítulo sobre punteros.

Espero no haberos liado demasiado con esta explicación sobre los arrays, las estructuras, las variables estáticas y las dinámicas. Si algo no se entiende no teneis más que preguntarme, claro está. ;) Ahora ya sólo nos quedan 2 capítulos para poder hacer un programa completo.

por ComaC

Notas
Introducción
Capítulo 1º
Capítulo 2º
Capítulo 3º
Principal
Capítulo 5º
Capítulo 6º
Capítulo 7º
Capítulo 8º

Capítulo 4º

EXPRESIONES

Bueno... antes de nada esta vez quiero decir que he estado a punto de cargarme el HD enterito y que hace poco me cargué más de la mitad de la base de mensajes. Con lo cual he perdido todos los mensajes sobre el cursillo y el índice que tenía hecho. Pero por suerte, todabía tengo un montón de hojas sueltas en las que tengo todos los esquemas que me planteé antes de empezar así que por ahora parece que pese a todo podré continuar. O:)

Hoy nos toca hablar sobre las expresiones. Es un asunto sencillo pero que es importante que quede claro. Una expresión, para decirlo de manera sencilla, podríamos decir que es un cálculo. Por ejemplo, una expresión podría ser esto: (7+8*15-34)/10

Puesto que es un cálculo podemos indicar operaciones. Tenemos dos tipos de operaciones las numéricas y las lógicas. Las numéricas las realizamos con números y dan como resultado un número. Las lógicas las podemos realizar con números o con valores booleanos y dan como resultado un valor booleano.

OPERACIONES NUMERICAS

Empecemos por las numéricas. Como operaciones numéricas tenemos estas:

+ : Suma.
- : Resta.
*** : Producto.**
% : Módulo. (es decir, haya el resto de la división) EJ: 4 % 3. Da el resto de 4 / 3, que es 1.
/ : División de reales. Esta división da como resultado un número real. Acepta como operandos tantos números reales como enteros.
Div : División de enteros. Esta división da como resultado un número entero y no acepta como operandos números reales.

Los operandos que podemos utilizar son: valores inmediatos, constantes o variables. Por ejemplo, en este caso:

```
Const
  Constante = 7;

Var
  Tabla : Array [0..10] of Byte;
  A     : Word;

Begin
A := 8 + Tabla[9] - Constante;
End.
```

Aquí "8" sería un valor inmediato, "Tabla[9]" una variable, y "Constante" una constante. Así de sencillo.

Un punto importante a tener en cuenta es la preferencia de las operaciones. Es decir, que " 8 + 3 * 2 " NO es lo mismo que " (8 + 3) * 2 ". En el primer caso se calcula primero 3*2 y al resultado se le suman 8 mientras que en el segundo se calcula primero 8 + 3 y el resultado se multiplica por 2. La preferencia es la misma que utilizamos normalmente para hacer operaciones matemáticas: primero los paréntesis, luego las multiplicaciones/divisiones/módulos y por último las sumas.

Una expresión numérica puede utilizarse en una asignación, como en el ejemplo anterior, para pasar un parámetro a una función, etc... Por ejemplo, en lugar de "A := 8 + Tabla[9] - Constante" podríamos cambiarlo y poner "A := 8 + Tabla[Tabla[3]+Constante*2] - Constante".

Con esto se calcularía 7 * 2 (7 es el valor de "constante"), se le sumaría el valor almacenado en la posición 3 de la tabla y ese resultado se utilizaría como índice de la tabla. Se leería en valor que indique esa posición y se le sumaría 8 y se le restaría 7. (la constante)

OPERACIONES NUMERICAS

Las operaciones lógicas devuelven siempre verdadero (true) o falso (false). Se utilizan para hacer comprobaciones.

Las operaciones lógicas de que disponemos son estas:

= : Devuelve verdadero si los dos operandos son iguales
<> : Devuelve verdadero si los dos operandos son diferentes
> : Devuelve verdadero si el primer operando es mayor que el segundo
< : Devuelve verdadero si el segundo operando es mayor que el primero
>= : Verdadero si el primer operando es mayor o igual que el segundo
<= : Verdadero si el primer operando es menor o igual que el segundo
and : Devuelve verdadero si las dos expresiones lógicas son verdaderas
or : Verdadero si una o las dos expresiones lógicas son verdaderas
not : Devuelve verdadero si la expresión es falsa

Y ahora expliquemos esto un poquito:

Las primeras 6 operaciones son muy sencillas. Sirven para comparar dos expresiones numéricas y nos devuelven el valor booleano correspondiente. Así, por ejemplo, la expresión "3 = 2" siempre nos

devolverá falso, pero la expresión "A = 2" tal vez devuelva falso o tal vez devuelva verdadero, ya que dependerá de si la variable A está almacenando o no el valor 2 en el momento en que se hace la comprobación.

Como siempre, podemos complicar las cosas y escribir cosas como esto:

```
(Tabla[3] - Constante + 8 * Tabla[ Tabla[ 9 - 3 ] + 9) >= Tabla[5]
```

(Si alguien no acaba de ver claro esto, que pruebe a desglosar en pasos esta expresión, tal y como he hecho yo con las anteriores, y me lo comente)

Las 3 últimas operaciones de la lista ya no trabajan con expresiones numéricas sino con expresiones lógicas. Teniendo arriba la función de cada una creo que lo mejor es explicarlo mediante ejemplos.

(3 = 2) or (1 = 4) daría siempre falso porque la expresión (3 = 2) da falso y la expresión (1 = 4) también. Y "or" sólo devuelve verdadero cuando al menos una de las expresiones es verdadera.

(3 = 2) or (1 = 1) daría verdadero porque aunque (3 = 2) es falso, (1 = 1) es verdadero.

((3 = 2) or (1 = 1)) and (9 = 3) devolvería falso porque ((3=2) or (1=1)) da verdadero pero (9 = 3) da falso. Y para que "and" devuelva verdadero han de ser verdaderas las dos expresiones.

not (1=3) daría verdadero. Porque (1=3) da falso y "not" devuelve lo contrario de lo que vale la expresión dada.

Hasta aquí es sencillo, pero hay que tener en cuenta que no nos vamos a limitar a hacer que el Turbo Pascal nos diga que 1 no es igual a 3. La finalidad de esto es mucho más útil, por supuesto. Hay que recordar que para construir las expresiones lógicas podemos valernos de cualquier expresión numérica. Esto es, si tenemos dos expresiones numéricas como estas:

```
(Tabla[4] - 8) * 34
(Contante + A) % Tabla[3]
```

Podemos hacer una expresión lógica que compare esas dos expresiones, como por ejemplo:

```
( (Tabla[4] - 8) * 34 )  >=  ( (Constante + A) % Tabla[3] )
```

Y una vez tenemos esta expresión lógica podemos combinarla con otras expresiones lógicas para hacer expresiones lógicas más complejas, por ejemplo algo como esto:

```
((( Tabla[4] - 8) * 34 )  >=  ((Constante + A) % Tabla[3])) or (2=(1+A))
```

(si alguien quiere entretenerse, podría ser un buen ejercicio calcular para qué valores de Tabla[4], Tabla[3] y A la expresión ENTERA es

verdadera.

Uhm... creo que por hoy no tenía que decir nada más. :m ... Tampoco os puedo asegurar que aquí esté todo lo que tenía que decir, porque he perdido los esquemas, pero... bueno... creo que sí. O:)

por ComaC

Notas
Introdución
Capítulo 1º
Capítulo 2º
Capítulo 3º
Capítulo 4º
Principal
Capítulo 6º
Capítulo 7º
Capítulo 8º

Capítulo 5º

EL CODIGO DEL PROGRAMA

Bueno... este capítulo sale con un pelín de retraso, pero... bueno... es que entre que hoy tenía el global de física y que ayer me quedé viciado al Need for Speed... O:)

Pero bueno, no importa porque este capítulo es uno de los más interesantes. No vamos a meter ya con lo que es el código principal del programa!! ;)

El código es el conjunto de órdenes que el ordenador debe seguir. Básicamente estas órdenes serán o bien asignaciones o bien llamadas a funciones. El código está encerrado entre las palabras claves BEGIN y END.

Las asignaciones consisten símplemente en asignarle un valor a una variable. Para ello se usa algo como esto:

```
Variable := <expresión> ;
```

Las expresiones ya las vimos en el capítulo anterior. No hay que olvidar los dos puntos antes del igual ni el punto y coma al final de la linea. Como variable podemos poner cualquier variable de las que vimos en el capítulo sobre variables. (un array, una estructura, etc...)

Y a parte de las asignaciones podemos hacer llamadas a funciones, cosa que ya veremos cuando hablemos de las funciones.

Pero si sólo pudiésemos hacer esto un programa no podría ser complejo. Lo que tenemos, además de lo que ya he dicho, son sentencias condicionales y bucles.

SENTENCIAS CONDICIONALES

Una sentencia condicional es algo como esto:

```
IF <expresión lógica> THEN
  <código1>
ELSE
  <código2>;
```

De manera que si la expresión lógica es cierta se ejecuta el código1 mientras que si es falsa se ejecuta el código2. También puede escribirse así:

```
IF <expresión lógica> THEN <código>;
```

De esta manera si la expresión es cierta se ejecutará el código y si es falsa no se hará nada. Normalmente lo que se hace es encerrar entre un BEGIN y un END todo el código que será afectado por la sentencia condicional, así, un ejemplo sería este:

```
IF Variable1 = 0 THEN
  BEGIN
    Variable2 := 1;
    Variable1 := 4;
  END
ELSE
  BEGIN
    Variable2 := 2;
    Variable1 := 7;
  END;
```

Y aquí ATENCION a dónde están puestos los puntos y coma. Los puntos y coma se usan en Pascal para marcar el final de una instrución. Así después de cada una de las asignaciones hay uno. Y también debe ir otro al final de la sentencia condicional. (Después del último END)

BUCLES

Los bucles nos sirven para repetir unas mismas instruciones varias veces. Para esto tenemos 5 tipos de bucles:

FOR...TO

Este tipo de bucle sigue esta estructura:

```
FOR <variable> := <inicio> TO <fin> DO
  <código>;
```

Donde variable es cualquier numérica. (no podemos usar un String), donde es el número a partir del que empezamos a contar y donde es el número al que tenemos que llegar para parar de ejecutar .

Pongamos un ejemplo:

```
FOR Contador := 1 TO 5 DO
  BEGIN
  Variable1 := Contador;
  Variable2 := Variable2 + Contador;
  END;
```

Aquí la sentencia FOR empezaría a contar desde el 1 y ejecutando una vez el código antes de sumarle uno a . Cuando contador llegue a 5 ya no se ejecutará más el código. Por lo tanto se ejecutará una vez para Contador=1, otra para Contador=2, para Contador=3, para Contador=4 y para Contador=5. Por lo tanto el código se ejecutará 5 veces.

Cuidado a la hora de usar 0 como inicio para el contador, porque de 0 a 5 el código se ejecutaría 6 veces!

También podemos usar DOWNTO en lugar de TO en esta sentencia y entonces lo que hará será restar uno al contador en vez de sumarlo. Entonces el número al que ha de llegar tiene que ser MENOR que el inicial, es decir, ha de ser algo como esto:

```
FOR Contador := 5 DOWNTO 1 DO
  Variable := Contador;
```

DO...WHILE

Este otro bucle sigue esta estructura:

```
DO
  <código>
WHILE <expresión>;
```

En este caso no hace falta encerrar entre BEGIN-END el código porque ya queda delimitado por el DO-WHILE y no hay lugar a confusión. Este bucle es similar al FOR. Lo que hace es repetir el código mientras la expresión sea cierta. PERO la comprobación la hace después de haber ejecutado el código. Esto es importante porque gracias a eso el código se ejecutará SIEMPRE una vez, como mínimo. Así, esto se ejecutaría una vez:

```
A := 8;
DO
  A := 8;
WHILE A = 9;
```

En cambio esto se ejecutaría 3 veces:

```
A := 1;
DO
  A := A + 1;
WHILE A <= 3;
```

Al acabar la primera ejecución A sería igual a 2, al acabar la 2ª A sería igual a 3 y al acabar la tercera A sería igual a 4. Como 4 no es ni menor ni igual a 3 el bucle de para.

WHILE...DO

Este es prácticamente idéntico al anterior. Tiene esta estructura:

```
WHILE <expresión> DO
  <código>;
```

En este caso vuelve a ser necesario el BEGIN-END si queremos que lo que se repita sea más de una

instrución.

En este bucle la comprobación se hace ANTES de ejecutar el código y por eso puede que el código no se ejecute ninguna vez en el caso de que la condición sea ya al principio falsa.

REPAT...UNTIL

Este bucle sigue la siguente estructura:

```
REPEAT
  <código>
UNTIL <expresión>
```

En este caso no es necesario el BEGIN-END y lo que hace este bucle es repetir el código hasta que la expresión sea cierta, o lo que es lo mismo: lo repite mientras la expresión sea falsa. Es decir, algo como esto no pararía nunca de ejecutarse:

```
REPEAT
  A := A + 1;
UNTIL 2 = 3;
```

Aquí le estamos diciendo que repita el código hasta que 2 sea igual a 3. Y... como yo creo que va a tardar un ratito en que 2 sea igual a 3 pues... este bucle no acabaría nunca y tendríamos que hacer un reset o pulsar CTRL+C si estamos ejecutándolo desde el IDE.

Bueno. Y esto es todo por hoy. Tal vez parezca que no he dicho gran cosa, pero ahora ya sólo nos falta dar las funciones para poder empezar a hacer programas completos. Por supuesto, en el próximo capítulo toca funciones. ;)

por ComaC

Notas
Introducción
Capítulo 1º
Capítulo 2º
Capítulo 3º
Capítulo 4º
Capítulo 5º
Principal
Capítulo 7º
Capítulo 8º

Capítulo 6º

PROCEDIMIENTOS Y FUNCIONES

Bien... en el último capítulo vimos en qué consistía el código de un programa. Hoy veremos "otro tipo" de código: los procedimientos y las funciones. A este "otro tipo" de código también se le suela llamar rutina, subrutina, a veces se le llama módulo... y algún que otro nombre más que ahora se me debe olvidar. Lo más comú suele ser referirse genéricamente a procedimientos y funciones con el término fución o rutina.

Un procedimiento o una función es, por decirlo sencillamente, un trozo de código. Pero no cualquier trozo de código. Sino un trozo que nosostros hemos separado por algún motivo.

Pongamos un ejemplo: estamos haciendo un programa que ha de calcular muchos factoriales. Hemos hecho un código que calcula el factorial de un número, pero no queremos tener que repetir todo ese código cada vez que haya que calcular un factorial. Entonces lo que hacemos es separar ese código y darle un nombre, de manera que después, con poner sólamente el nombre ya no nos haga falta copiarlo todo.

También podría ser que hiciésemos un programa que dibuja muchos cuadrados en la pantalla. Igualmente "aislaríamos" el trozo de código que dibuja el cuadrado para poder usarlo más adelante.

Y, como no he puesto dos ejemplos sólo por entretenerme, expliquemos en qué se diferencian estos dos casos. En el primero, el del factorial, el código de la función tiene que devolver un valor: el resultado de calcular el factorial, mientras que en el segundo caso, para dibujar un cuadrado no nos hace falta que nos devuelva ningún resultado; símplemente ha de dibujarlo y listos.

Por este motivo, el primer caso sería una Función y el segundo sería un Procedimiento. La única diferencia entre la función y el procedimiento es que la función devuelve un valor y el procedimiento no.

DECLARACION

Ahora vemos cómo nos las apañamos para separar el código en cuestión y que el Turbo Pascal se entere.

Para un procedimiento usaríamos este esquema:

```
Procedure Nombre ( parámetros );
```

Y para una función este otro:

```
Function Nombre ( parámetros ) : Tipo;
```

Donde "Nombre" es el nombre que queremos darle a la función. Normalmente suele corresponder a la función que hace, así, por ejemplo, la función que calcula el factorial podría llamarse "Calcula_Factorial", aunque nada nos impide llamarla "Inicia_Sound_Blaster".

Los parámetros nos sirven para pasarle información a la función. El formato que hay que seguir es igual al que usábamos para declarar las variables:

```
Procedure BlaBla ( Nombre1 : Tipo1; Nombre2 : Tipo2 );
```

Podemos poner cualquier número de parámetros y de cualquier tipo.

Podemos usar enteros, cadenas, números en punto flotante, estructuras, punteros... cualquier tipo de datos.

Y en las funciones, por último, tenemos que indicar un "Tipo" ¿Qué tipo? Pues el tipo de datos del valor que devuelve la función. Esto es, como la función Calcula_Factorial devolverá un número entero que puede ser bastante grande nos convendrá poner que dicha función devuelve un dato de tipo LONGINT. La declararíamos así:

```
Function Calcula_Factorial ( Numero : Longint ) : Longint;
```

De esta manera, pasaríamos como parámetro el número del cual hay que calcular el factorial y la función nos devolvería el resultado en un longint.

VALOR Y REFERENCIA

Ahora bien, no es tan sencillo el paso de parámetros. Podemos pasarlos por valor o por referencia.

Por valor es la manera que hemos usado en los ejemplos anteriores. De esta manera lo que se pasa a la función es, como si dijésemos, una "copia" de la variable original. Con lo que, si modificamos el valor de ese parámetro no variará el contenido de la variable original.

En cambio, al pasar parámetros por referencia lo que se hace es pasar la variable original directamente. (esto no es muy exacto que digamos, pero ya lo explicaré mejor cuando hayamos explicado los punteros) Así, si modificamos el parámetro pasado por referencia TAMBIEN modificamos la variable original. (al final pondré unos ejemplos para que quede más claro)

Para que un parámetro se pase por referencia tenemos que declararlo de una manera especial: tenemos que preceder su nombre de la palabra reservada "var". Pongamos un ejemplo:

```
Procedure Calcula_Factorial (Numero : Longint; var Resultado : Longint);
```

En este ejemplo pasamos "Numero" por valor y "Resultado" por referencia. De esta manera modificando "Numero" no modificamos la variable original que se ha pasado como parámetro a la función, mientras que modificando "Resultado" sí que modificamos el contenido original de la variable. De esta manera no necesitaríamos declarar esta rutina como función. Podemos hacerlo como procedimiento y devolver el resultado en "Resultado".

TIPOS, CONSTANTES Y VARIABLES LOCALES

Después de la declaración de la función podemos definir tipos de datos locales, constantes locales y variables locales. Estos son como los tipos, constantes y variables normales (que ya vimos) sólo que no podemos usarlos en otro lugar más que en la función en la que han sido definidos. Es decir: desde una función podemos usar los tipos/variables/constantes globales y también los locales que hayamos definido en la función, pero desde el código principal (que explicamos el pasado capítulo) sólo podemos acceder a las variables/constantes/tipos globales.

Y aquí nos puede aparecer un conflicto: ¿qué pasa si tenemos una variable local que se llama igual que una variable global? ¿A cual hace

caso el Turbo Pascal? Pues siempre que se dé este caso, cuando usemos el nombre de esas variables, el Turbo Pascal asume que nos estamos refiriendo a la variable local y no a la global.

EL CODIGO DE LA FUNCION

Una vez hemos puesto todo lo anterior podemos poner un BEGIN...END; y escribir en medio el trozo de código que queríamos separar del resto. Atención a que en este caso después del END no va un punto sino un punto y coma. (En el código principal sí que va un punto).

Este código es idéntico al código global sólo que tiene acceso a los tipos/constantes/variables globales y locales. Desde una función podemos llamar a otra función o incluso a la misma función, pero llamar a una misma función es ya otro tema; se le llama recursividad y ya la explicaremos más adelante. Por ahora dejémoslo en que si una función se llama a si misma lo que sucederá será que obtendremos un runtime error por desbordamiento de pila. ¿Que significa esto? Pues sencillo: que la función se llama a ella misma sin parar. Cada vez que se llama perdemos un poco de memoria, que se usa para guardar las "copias" de los parámetros pasados por valor y algunas otras cosas... si la rutina se llama a si misma infinitamente llega un momento en que no queda memoria para pasar los parámetros y por eso se produce el error.

En el caso de las funciones (refiriéndonos ahora a las rutinas que devuelven un valor) tenemos que devolver el valor antes de acabar el código. Esto se hace asignando un valor al nombre de la función. Es decir: tratamos a la función como si fuese una variable y le asignamos un valor "a saco". ;) Ejemplillo al canto:

```
Function Devuelve_5 : Longint;
Begin
  Devuelve_5 := 5;
End;
```

Así de sencillo. :)

EJEMPLOS

Ahora que ya sabemos toda la teoría sobre las funciones podemos hacer el ejemplo del factorial. Pongamos que el código para hayar el factorial fuera este:

```
Factorial := 1;
For Contador := 1 To Numero Do
  Resultado := Resultado * Contador;
```

Donde Factorial, Contador y Numero son números enteros LONGINT.

Pues bien, o bien copiamos ese trozo de código en todos los sitios donde necesitemos calcular el factorial o bien lo aislamos en una función. La función podría quedar así:

```
Procedure Factorial ( Numero : Longint ) : Longint;
Var
  Resultado, Contador : Longint;

Begin
Resultado := 1;
```

```
For Contador := 1 to Numero Do
  Resultado := Resultado * Contador;

Factorial := Resultado;
End;
```

Y después llamaríamos a esta función desde el código principal:

```
Var
  A : Longint;

Begin
A := Factorial( 8 );
A := Factorial( A );
A := Factorial( A div 10 );
End.
```

Como parámetro podemos pasr cualquier expresión. Pero sólo en este caso. Podemos pasar cualquier expresión porque el parámetro está declarado como parámetro pasado por valor. En el caso de parámetros por referencia no podemos usar un expresión sino que tenemos que indicar una variable.

Por ejemplo, podemos hacer que la función Factorial devuelva el resultado mediante un parámetro pasado por referencia:

```
Procedure Factorial (Numero: Longint; var Resultado : Longint);
Var
  Contador : Longint;

Begin
Resultado := 1;

For Contador := 1 To Numero Do
  Resultado := Resultado * Contador;

End;
```

Y esta función la llamaríamos así desde el código principal:

```
Var
  A : Longint;

Begin
Factorial ( 8, A );
End.
```

Pero algo de este estilo provocaría un error de compilación:

```
Factorial ( 8, A + 5 );
```

Porque no podemos pasar una expresión como parámetro por referencia. En cambio, ya que el primer parámetro se pasa por valor, SI que podemos poner cosas que estas:

```
Factorial ( A + 8, B );
```

Y más cosas... Un detallito que parece no tener importancia: si os fijais vereis que al escribir la "Function Factorial" el resultado se va calculando en una variable local y después se asigna como resultado de la función. En cambio en el "Procedure Factorial" el resultado se calcula directamente en el parámetro por referencia en el que se ha de devolver el resultado.

¿Por qué? Pues porque en la función, para ir calculando el valor directamente como resultado de la función deberíamos poner una linea como esta:

```
Factorial := Factorial * Contador;
```

Y una estructura del tipo := ... no es otra cosa que la manera de llamar a la función que se indica. (tal y como hemos hecho en el código principal que llama a la función Factorial). Por lo tanto, si pusiéramos esa linea en la función, el compilador entendería que queremos llamar a la función Factorial dentro de la misma función Factorial, con lo cual se produciría la "llamada infinita" de la que he hablado antes y acabaría con un runtime error.

En cambio en el procedimiento no corremos ese riesgo, ya que los parámetros se comportan exactamente igual que si fuesen variables globales, con lo cual podemos usarlos en cualquier sentencia de asignación normal.

Bueno... esto es todo por hoy. Espero no haberos liado demasiado y espero también no retrasarme tanto para el próximo capítulo, que si no luego me pego unos atracones de escribir el último día que... O:)

por ComaC

Notas
Introducción
Capítulo 1º
Capítulo 2º
Capítulo 3º
Capítulo 4º
Capítulo 5º
Capítulo 6º
Principal
Capítulo 8º

Capítulo 7º

FUNCIONES DE LAS LIBRERIAS

Hasta ahora hemos ido viendo cómo hacer funciones y procedimientos en Pascal, ahora que ya sabemos bien lo que son explicaré las funciones más básicas que ya trae hechas el TP.

Un librería no es más que un conjunto de funciones metidas juntas en un archivo. Y el TP nos trae unas cuantas librerías que nos ayudan bastante a la hora de hacer las funciones más básicas, como por ejemplo escribir en la pantalla o leer las pulsaciones del teclado.

Yo no las explicaré todas ni mucho menos. Para obtener información sobre ellas basta con ir a los contenidos de la ayuda del compilador, coger "units" y escoger la librería de la que queremos chafardear las funciones.

Algunas de las funciones básicas están en una librería llamada SYSTEM.TPU que se incluye siempre en todos los programas que hagamos. Si usamos esas librerías no necesitamos hacer nada, pero si usamos funciones de otra librería tendremos que añadir el nombre de esta librería a la linea USE. Es decir, si usamos funciones de las librerías DOS.TPU y CRT.TPU tendríamos que poner esto al principio del programa:

```
USES Crt, Dos;
```

FUNCIONES DE ENTRADA SALIDA

Para empezar tenemos Write y WriteLn, que se encargan de escribir algo en la pantalla. La diferencia entre ellas es que WriteLn pasa a la linea siguiente después de escribir mientras que Write deja el cursor en la posición donde quede.

Podemos pasarles casi cualquier tipo de datos, así, un programa como este:

```
Var
   A : Integer;
   B : String;

Begin
A := 8;
B := ' duros.';
WriteLn('Tengo ',a,b);
End.
```

escribirá "Tengo 8 duros." (sí. ya sé que las frases de ejemplo no son lo mio. ;)

Estas dos funciones están tanto en SYSTEM.TPU como en CRT.TPU. La diferencia es que las funciones de CRT.TPU pueden escribir en color. Si os interesa buscad en la ayuda sobre TextColor.

Para la entrada (lectura del teclado) tenemos Keypressed y ReadKey. La primera es una función que nos devuelve un valor de tipo BOOLEAN, es decir, nos devuelve cierto o falso según si se ha pulsado una tecla o no. Esta función ni lee la tecla ni la saca del buffer del teclado.

La segunda también es una función, pero nos devuelve un valor de tipo

char que contiene el código ASCII de la tecla que se ha pulsado. Esta función sí que saca la pulsación del buffer del teclado. En el caso de que no haya nada en el buffer del teclado cuando se llama a esta función, el programa se detendrá hasta que se pulse una tecla.

También tenemos Read y ReadLn. Su uso es inverso a Write y WriteLn. Con Read(A), por ejemplo, el programa leerá UNA tecla y la almacenará en A (supongamos que A es un CHAR). Mientras que ReadLn leería una linea entera y la almacenaría en el parámetro. Esto no sólo puede usarse para caracteres y cadenas sino también para números, por ejemplo:

```
Var
   A : Integer;

Begin
ReadLn(A);
A := A + 1;
WriteLn('Has escrito el número anterior a ',A);
End.
```

Supongo que no hace falta que explique lo que hace este programa. Sólo tener en cuenta que si como número escribimos, por ejemplo, "HOLA", el programa dará un error de ejecución porque no sabrá como parar HOLA a Integer.

FUNCIONES DE LECTURA/ESCRITURA DE ARCHIVOS DE TEXTO

Para trabajar con archivos de texto tenemos que definir una variable de tipo TEXT. A esta variable le "asignaremos" el archivo que queremos tratar y lo abriremos o lo crearemos o continuaremos escribiendo en él. Para asignarle un archivo a una variable haremos algo como esto:

```
ASSIGN(VariablaTEXT, 'Nombre.Ext');
```

Y después podemos escoger si queremos abrir un archivo (en este caso el archivo debe existir, en caso contrario el programa se detendrá produciendo un error de ejecución), podemos crearlo (con lo que si ya existe un archivo con ese nombre lo sobreescribiremos) o podemos abrirlo para continuar escribiendo al final del archivo. (en este último caso el archivo también debe existir)

Si abrimos el archivo sólo podremos leerlo. Si lo creamos sólo podremos escribirlo y si lo abrimos para continuar escribiendo sólo podremos escribir a partir del final del archivo.

Para abrirlo usaremos RESET(VariableTEXT), para crearlo usaremos REWRITE(VariableTEXT) y para continuar escribiendo usaremos APPEND(VariableTEXT).

Una vez tenemos abierto el archivo podremos leer/escribir. Para eso usaremos Write/WriteLn y Read/ReadLn. Las usaremos exactamente igual que antes sólo que el primer parámetro que pasaremos será la variable de tipo TEXT que identifica al archivo, por ejemplo:

```
A := 3;
WriteLn(VariableTEXT,'Hola!! He estado ',A,' veces en Madrid.);
```

Esto escribiría Hola!! He estado 3 veces en Madrid en el archivo de texto. Y si queremos leer 3 valores enteros que están en una misma linea

de un archivo de texto podemos usar:

```
ReadLn(VariableTEXT, Valor1, Valor2, Valor3);
    {Siendo Valor1, Valor2, Valor3 de tipo INTEGER}
```

Para cuando leamos disponemos de la función EOF(VariableTEXT), que nos devolverá TRUE cuando hayamos llegado al final del archivo. Esto nos sirve para saber cuándo hay que parar de leer.

Cuando hayamos terminado con el archivo debemos cerrarlo. Para ello usamos CLOSE(VariableTEXT). Es importante no olvidarse de esto, ya que si no puede que no se graben los últimos cambios que se hayan escrito al archivo.

FUNCIONES DE LECTURA/ESCRITURA DE ARCHIVOS CON TIPO

En lugar del tipo TEXT podemos usar el tipo FILE OF para declarar un archivo que no será ya de texto sino que contendrá elementos del tipo . Por ejemplo, si queremos hacer un archivo que esté lleno de datos de tipo INTEGER lo declararíamos así:

```
Var
   Archivo : FILE OF Integer;
```

Lo asignaremos igual que los archivos de texto, pero a la hora de abrirlo sólo podemos usar RESET y REWRITE. Pero en este caso en ambos modos podemos leer y escribir indistintamente. La diferencia está en que RESET abre un archivo existente mientras que REWRITE crea/sobreescribe el archivo.

Para leer/escribir en el archivo usaremos Write y Read, PERO NO WriteLn ni ReadLn. Y, por supuesto, ahora ya no podemos escribir cualquier tipo de datos sino que sólo podemos leer/escribir datos del tipo del que es el archivo. Así, para llenar de ceros un archivo de Integers haríamos esto:

```
Var
   Archivo : FILE OF Integer;
   Contador : Integer;
   Valor    : Integer;

Begin
Assign(Archivo, 'Prueba.Pr');
ReWrite(Archivo);

Valor := 0;
For Contador := 1 to 10000 do
  Write(Archivo,Valor);

Close(Archivo);
End.
```

Para archivos de este tipo disponemos de dos funciones muy cucas. Por un lado tenemos FileSize(Archivo), que nos devuelve un Longint con el número de elementos que cfontiene el archivo. Al final de el ejemplo anterior FileSize devolvería que tenemos 10.000 Integers. (que ocuparían 20.000 bytes, pero FileSize no devuelve los bytes sino el número de elementos).

La otra instrución es SEEK(Archivo, Posicion) que lo que hace es movernos al elemento del archivo. De esta manera, por ejemplo, podemos

leer salteados los elementos de un archivo, o podemos sobreescribir un elemento concreto.

Estas dos instruciones también nos sirven para emular un APPEND. Para eso sólo tendríamos que abrir un fichero e irnos a la última posición para seguir escribiendo a partir de ahí. Algo como esto:

```
Begin
Assign(Archivo, 'Prueba.Pr');
Reset(Archivo);
Seek(Archivo,FileSize(Archivo));
End.
```

Atención a que saltamos a la posición Filesize(Archivo) y no a FileSize(Archivo)+1, porque Seek considera que la primera posición del archivo es el elemento 0.

En estos archivos no funciona EOF y hemos de cerrarlos igualmente.

Bien. Hasta aquí llegamos hoy. Hay montones de funciones más, pero estás son las más basiquísimas para ir haciendo cosas. Si alguien está interesado en hacer algo en concreto no tiene más que decírmelo y esperemos que haya funciones del pascal para hacer eso. O:)

Ale... para la próxima "clase" nos toca un plato fuerte: LOS PUNTEROS!!!! 8-O ;)

por ComaC

Notas
Introdución
Capítulo 1º
Capítulo 2º
Capítulo 3º
Capítulo 4º
Capítulo 5º
Capítulo 6º
Capítulo 8º
Principal

Capítulo 8º

LOS PUNTEROS

Aisshhh... que pocas ganas de escribir, con el calor que hace... pero bueno, si no lo escribo antes de agosto ya no lo escribiré, así que... venga, antes de empezar a estudiar pa septiembre acabaremos el cursillo...

Lo primero: hay que entender muy bien lo que es un puntero, porque al principio puede se run poco complicado. Un puntero es un tipo de datos (como un INTEGER, por ejemplo) que lo que almacena no es ni un número ni una letra sino una direción de memoria. Una posición de memoria es una "cosa" del tipo 0xxxxh:0xxxxh que identifica una posición de la memoria CONVENCIONAL del ordenador. (Recordemos que el TP 7 sólo sabe hacer programas para 286, o sea: para modo real, por lo tanto sólo podemos usar la memoria convencional [esto en realidad es falso, se puede usar memoria XMS y EMS desde Pascal, pero eso ya es otro tema]) Se dice que un puntero APUNTA a la direción de memoria.

Para declararlo tenemos el tipo POINTER, con lo que lo declaramos igual que cualquier otro tipo de datos:

```
Var
  Puntero : Pointer;
```

¿Y esto para qué nos sirve? Pues así, con lo que sabemos ahora, de nada. Pero lo curioso de los punteros es que podemos decirle qué es lo que hay en la posición de memoria a la que está apuntando. Me explico: el puntero "señala" a un lugar de la memoria de nuestro PC. Pues ese lugar podemos tratarlo como un BYTE, o como un WORD, como un STRING, como una estructura de datos o incluso como otro puntero!

En este caso la declaración cambia. En este caso se declara igual que si declarásemos una variable de un tipo concreto sólo que precedemos el nombre del tipo por el símbolo ^. (ALT+94)

Así para declarar un puntero de manera que la memoria a la que apunta sea tratada como un byte haríamos esto:

```
Var
  Puntero : ^Byte;
```

Y lo mismo para una estructura:

```
Type
  Tipo = record
    Nombre : String;
    DNI    : Longint;
    NIF    : Char;
    Calle  : String;
    Edad   : Word;
    End;
```

```
Var
  Puntero : ^Tipo;
```

Este tipo de punteros se denominan de una manera especial, son punteros A el tipo de datos. Es decir, este último ejemplo sería un puntero A una estructura (o a la estructura Tipo), el anterior sería un puntero a un byte, etc... Y también pueden hacerse punteros a punteros, punteros a arrays de punteros, etc... pero eso es más complicado y vendrá más adelante.

Pero... ¿a dónde apunta un puntero? Eeeeiiinngg? ;) Me refiero: un puntero, nada más empezar el programa ¿apunta a alguna parte? ¿a dónde apunta? Pues lo más normal es que apunte a la direción 0000:0000, ya que las variables suelen inicializarse a cero. Por lo tanto antes de usar el puntero tenemos que hacer que apunte al sitio que nosotros queremos. ¿Qué pasa si no lo hacemos? Pues si modificamos un puntero que apunta a 0000:0000... en esa zona de la memoria está la tabla de los vectores de interrupción, si lo modificamos nos cepillaremos esa tabla, con lo cual, en cuanto se genere una interrupción del temporizador (una cada 55 milisegundos) nuestro programa nos brindará un precioso cuelgue.

Este tipo de punteros puede ser más útil, porque... Ya que un puntero puede apuntar a cualquier parte de la memoria convencional... podemos modificar cualquier posición de memoria, tanto si pertenece a nuestro programa como si no. Por ejemplo, el WORD de la posición 040h:02h contiene la direción del puerto base para el COM2, para leerla sólo tenemos que hacer que un puntero a un word apunte a esa direción (040h:02h) y después leer lo que hay allí. Para hacer que un puntero apunte a un sitio determinado tenemos la función Ptr. La función Ptr acepta como parámetros el segmento y el offset de la direción de memoria y nos devuelve como resultado un puntero a esa direción. Así, es programa que lee la direción del COM2 quedaría así:

```
Var
  Puntero_a_COM2 : ^Word;

Begin
Puntero_a_COM2 := Ptr($40,2);
WriteLn('La direción del COM2 es ',Puntero_a_COM2^);
End.
```

Una cosa importante a tener en cuenta es que no es lo mismo la direción de memoria que lo que hay en esa direción. El puntero sólo contiene la direción, no contiene lo que hay en ella. Es decir: el puntero sólo señala al WORD donde está la direción del COM2. El puntero NO almacena la posición del COM2. Por eso estas dos lieas son bien diferentes:

```
Puntero := 8;
Puntero^ := 8;
```

La primera intenta asignar el valor 8 como direción de memoria a la cual tiene que apuntar el puntero. Mientras que la segunda lo que hace es asignar el valor 8 a la direción de memoria a la que apunta el puntero. Es muy importante que esto quede claro.

Podríamos decir que Puntero^ es el elemento y Puntero es la direción del elemento.

Pero la utilidad de los punteros no acaba aquí ni mucho menos. Creo que ya antes había dicho que las variables de un programa sólo podían ocupar 64 Kbs como mucho. Y... ¿qué ha pasado con en resto de los 640 Kbs de memoria? Pues esa memoria se puede usar mediante punteros.

Para pedir memoria tenemos la función Getmem y para liberarla una vez ya no la necesitamos tenemos Freemem. A ambas hay que pasarles un puntero y el tamaño de memoria que queremos reservar. Por ejemplo:

```
Getmem(Puntero, 30000);
```

Esto lo que haría sería reservar 30000 bytes de memoria y hacer que el puntero apunte al inicio de esos 30000 bytes. ¿A qué tipo de datos tiene que apuntar este puntero? A cualquiera. Si hemos declarado Puntero como un puntero a un byte podremos usar "Puntero^ := valor;" para modificar el primer byte del bloque de memoria que hemos pedido.

¿Y qué pasa con el resto? Pues muy sencillo. Podemos "mover" el puntero para que apunte al segundo byte del bloque, o al tercero... Para eso tenemos las funciones INC y DEC. Con INC(PUNTERO); hacemos apuntar el puntero a el siguiente elemento. PERO OJO!!! El siguiente elemento no siempre será el siguiente byte. Si hemos definido el puntero como un puntero a words cada vez que usemos INC avanzaremos 2 bytes (que es lo que ocupa un WORD). Y DEC se usa igual y tiene la función inversa.

La pega que tiene este método es que nadie nos impide hacer 600.000 INCs y poner el puntero apuntando a una parte de la memoria que no hemos reservado. Y es que para bloques de memoria hay maneras más prácticas de manejarlos.

Para esto son muy útiles los punteros a arrays. Podemos definir un así:

```
Type
  P = Array [0..29999] of Byte;

Var
  Puntero : ^P;

Begin
Getmem(Puntero, 30000);
End.
```

De esta manera definimos un array que empieza en la direción a la que apunta el puntero. Esta direción, después del Getmem, será la direción del bloque de memoria. Y a partir de esa direción tenemos 29999 elementos más en el array. Cada elemento es un Byte, por lo tanto, a cada posición del array le corresponde un byte del bloque de memoria que hemos reservado.

Ahora, para leer/escribir en ese bloque sólo tenemos que hacer cosas como esta:

```
Puntero^[8] := Puntero^[15003] - 80;
```

Fácil ¿no? Pero... ¿qué pasa si queremos poner todo el bloque a cero? Podríamos hacer un bucle FOR que fuese poniendo las posiciones a cero una por una, pero hay una manera mejor. Tenemos la instrución FillChar. Esta instrución llena con un valor dado el número de posiciones que le digamos a partir de la posición que le demos. Se usaría así:

```
FillChar(Puntero^, 30000, 0);
```

Atención a que ponemos "Puntero^" y no "Puntero". ¿Por qué? Pues porque lo que queremos llenar de ceros no es el puntero sino la memoria a la que apunta ese puntero. (Ya vereis que cuelgues más majos os salen cuando se os olvide esto ;)

Y... ¿qué pasa si tenemos 2 bloques de memoria y queremos copiar todo lo que haya en uno al otro? Pues podríamos hacer un FOR, pero hay una función específica para eso: Move. Move se usaría así:

```
Move(Puntero_Fuente^, Puntero_Destino^, Tamaño);
```

Cuando ya hayamos hecho todo lo que queríamos con un bloque de memoria debemos liberarlo. Para eso usamos Freemem igual que si fuese GetMem: Freemem(Puntero, Tamaño). Hay que tener en cuenta que el puntero debe apuntar exactamente al mismo sitio que después de hacer el GetMem. Es decir: si hemos usado algún INC o DEC con él debemos asegurarnos que lo dejamos tal y como estaba antes de llamar a Freemem.

Ahora supongamos que tenemos un puntero que apunta a una estructura y queremos reservar la memoria justa para que quepa esa estructura. Podemos contar los bytes que ocupa esa estructura (no estaría mal para practicar ;) o podemos usar New en lugar de Getmem. A New sólo hay que pasarle el puntero: New(Puntero);

Y el equivalente a Freemem entonces será Dispose, que usaremos igual que New.

Y ahora el punto más interesante. Antes de empezarlo os presento a NIL. NIL es una constante, pero no es una constante numérica sino un puntero. Y ¿a dónde apunta este puntero? En la ayuda dice que no apunta a ninguna parte. Y... aunque un puntero SIEMPRE apunta a alguna parte haremos como si fuese verdad que no apunta a nada. Así podemos decir que NIL es el puntero nulo.

Ahora empecemos: declaramos un puntero así:

```
Type
  PEsctructura = ^Estructura;

  Estructura = record
    Nombre    : String;
    Siguiente : PEstructura;
    End;

Var
  Puntero : PEstructura;
```

Bien... y reservamos memoria para el puntero:

```
New(Puntero);
```

En este momento ya tenemos un puntero apuntando a una estructura que consta de una cadena de caracteres y de otro puntero.

Rellenamos el primer campo:

```
Puntero^.Nombre := 'Pepito';
```

Y pedimos memoria para el segundo (ya que un puntero no podemos usarlo si no hemos pedido memoria para él o le hemos hecho apuntar a alguna parte).

```
New(Puntero^.Siguiente);
```

Ahora el puntero Siguiente apunta a una estructura de datos que consta de 2 campos: una cadena de caracteres y un puntero. ¿os suena de algo? ;)

Podemos volver a pedir memoria para el puntero de esta segunda estructura y luego pedir memoria para el puntero de la tercera, etc, etc...

¿Y qué ventaja tiene esto? Pues que podemos irlo repitiendo todas las veces que queramos hasta que se nos acabe la memoria. En un array hay que definir el número de elementos a la hora de escribir el programa, pero de esta manera podemos ir llenanado más y más elementos sin tener un límite prefijado.

La pega ahora es... ¿Cómo se trabaja con esto? Pues bien, para dejarlo lo más claro posible os pongo un ejemplo sencillo:

```
Type
  P2 = ^Estruc;

  Estruc = record
    Cadena    : String;
    Siguiente : P2;
    End;

Var
  Puntero, PAux : P2;
  Cadena        : String;

Begin
WriteLn('Memoria libre: ',maxavail);
WriteLn('Escribe nombres. (una linea en blanco para terminar)');

New(Puntero);
PAux := Puntero;
PAux^.Siguiente := Nil;

Repeat

  ReadLn(Cadena);
  PAux^.Cadena := Cadena;
  if Cadena <> '' Then
    Begin
```

```
      New(PAux^.Siguiente);
      PAux := PAux^.Siguiente;
      PAux^.Siguiente := Nil;
      End;

  Until Cadena = '';

  WriteLn;
  WriteLn('Has escrito:');

  PAux := Puntero;

  While PAux^.Siguiente <> Nil do
    Begin
    WriteLn(Paux^.Cadena);
    Puntero := Paux;
    PAux := Paux^.Siguiente;
    Dispose(Puntero);
    End;

  Dispose(Paux);
  WriteLn('Memoria libre: ',maxavail);

  End.
```

Y esta ha sido la última entrega del curso. Si quereis más sólo teneis que pedirlo, pero... a juzgar por el entusiasmo que ha levantado el curso no creo que abrais la boca... bueno... Pos eso, que me voy a afeitar, a pelarme tranquilo ahora que ya he acabado el cursillo... y a tudiar pa septiembre... :'(Maldita historia... :'(

por ComaC