Manual de FreePascal 1.0 para Win32

Versión 1.2

Este manual de FreePascal es gratuito pero ha requerido un esfuerzo considerable. Si encuentras algún fallo en el código de los ejemplos de los programas o bien algún aspecto que consideres que hay que corregir no dudes en ponerte en contacto conmigo en **pomuri@eresmas.net**

Gracias por tu colaboración.

El autor.

Roger Ferrer Ibáñez

1.Introducción a los lenguajes de programación

1.1.Etapas de la programación

El proceso de programación consta, teóricamente, de tres partes. La primera, que recibe el nombre de especificación, consiste en detallar que tiene que hacer el programa o, dicho de otra forma, concretar que problemas tiene que resolver nuestro programa. Una vez determinado qué hará nuestro programa empieza la fase de diseño. En la fase de diseño se escribe el algoritmo que es el conjunto de pasos que hay que realizar para resolver el problema planteado en la fase de especificación. De algoritmos pueden haber varios que hagan lo mismo, algunos serán más rápidos, otros más lentos, más simples o más complejos. Finalmente hay que implementar el algoritmo en un lenguaje de programación. En la mayoría de los casos la implementación se realizará en un lenguaje de programación de alto nivel. Los lenguajes de programación son un conjunto de reglas i símbolos que permiten la implementación de los algoritmos. En el caso de los lenguajes de alto nivel, estos son más cercanos a la sintaxis humana y al programador mientras que los de bajo nivel son más cercanos a las órdenes que entiende la máquina y en general más alejados del lenguaje natural. A consecuencia de esto los lenguajes de alto nivel se pueden emplear en más de un sistema mientra que los de bajo nivel tienen un ámbito más restringido.

Los programas escritos en lenguajes de alto nivel tienen que ser convertidos a lenguajes de bajo nivel para poder ser ejecutados en un sistema. Con esta finalidad existen los compiladores y los intérpretes. El compilador transforma el lenguaje de alto nivel en un lenguaje de bajo nivel que suele ser código máquina (o sea, directamente ejecutable por el procesador). Mientras que el intérprete codifica una o más instrucciones de forma secuencial, a medida que las va leyendo. Son lenguajes interpretados el BASIC y el LISP, mientras que el PASCAL, el C/C++ y muchos otros son compilados. Finalmente existe una opción híbrida introducida por JAVA que dispone de compilador e intérprete de los bytecodes que el compilador genera.

1.2.El lenguaje Pascal y el compilador FreePascal 1.00

El lenguaje de programación de alto nivel Pascal fue diseñado el 1968 por Niklaus Wirth con una finalidad eminentemente pedagógica. El 1983, el Pascal fue estandarizado llamandose ISO Pascal, justo en el mismo año en qué Borland lanzaría el Turbo Pascal 1.0, que sería el origen de una saga de éxitos que constó de 7 versiones para el entorno Ms-Dos. En estas sucesivas versiones, Borland aprovechó para corregir algunas deficiencias del Pascal original, que Niklaus Wirth había corregido ya en su otro lenguaje MODULA-2, volviendose un lenguaje de propósito general, fácil de aprender, potente (si bien a un nivel inferior que C/C++) y que se ha adaptado a los nuevos paradigmas de programación como son la programación orientada a objetos y clases.

El compilador FreePascal nació el julio de 1993 impulsado por su autor Florian Klaempfl. El 1996 fue lanzado a Internet y el julio del 2000, casi 7 años después de su inicio, salió la versión 1.00 del compilador, suficientemente estable como para poder desarrollar aplicaciones. Se distribuye bajo licencia GNU GPL que permite, básicamente, su distribución gratuita y del código fuente sin ningún coste aunque los autores retienen el copyright.

El compilador FreePascal existe para plataformas Ms-Dos, Windows de 32-bits, Linx, OS/2 y AmigaOs y recientemente se ha añadido FreeBSD. Está limitado a las arquitecturas Intel y Motorola. Como característica interesante hay que decir que soporta muchas de las características del Pascal de Borland y de ObjectPascal de Delphi y además incluye nuevas posibilidades inexistentes en estos compiladores, como es la sobrecarga de operadores, por poner un ejemplo. También aporta una librería estándar que funciona en todas las plataformas que da soporte el compilador.

En este manual se explicará como programar en Pascal y como aprovechar al máximo todas las posibilidades que el compilador FreePascal brinda al usuario, sin olvidar las que tratan del entorno Win32.

2.Antes de compilar ningún programa

2.1.Preparación del compilador

A diferencia de los productos comerciales, FreePascal no es un sistema diseñado para que los usuarios inexpertos lo empleen sin tener algún que otro problema antes de conseguir compilar su primer programa. En este apartado daremos algunas recomendaciones de forma que la tarea de instalación y preparación del entorno sea lo más fácil para los usuarios inexpertos.

2.1.1.Descarga e instalación del compilador

El compilador se puede descargar gratuitamente desde el área de descarga del web principal de FreePascal o desde alguno de los mirrors más cercanos. Para la mayoría de usuarios descargando el archivo completo sin fuentes, llamadas sources, és más que suficiente. El archivo en cuestión para la plataforma Win32 ocupa algo más de 8 Mb y viene comprimido en formato Zip. Para descomprimirlo podemos emplear WinZip o cualquier otro programa que de soporte a este formato.

Una vez descomprimido en una carpeta temporal, habrá que ejecutar el programa INSTALL.EXE. Este programa se encarga de descomprimir el resto de archivos Zip que incorpora el archivo de distribución. En la primera ventana que aparece, veremos tres páginas tabuladas. La primera, General, es donde tenemos que especificar el directorio donde se instalará FreePascal (Base Path). Recomendamos que active la opción de crear el archivo de configuración del compilador (Create ppc386.cfg).

En la siguiente página tabulada, Win32, se pueden escoger los elementos que queremos instalar, en nuestro caso una instalación mínima exige los componentes Required que son los 2 primeros, aunque una instalación completa sólo ocupa 20 Mb.

Finalmente, en la página Common podemos escoger instalar los elementos comunes a todas las distribuciones. Recomendamos que active la instalación de la documentación en formato PDF.

Al pulsar *Continue*, empieza la instalación. Finalmente, el programa nos avisará que es recomendable ampliar la variable `PATH` para poder trabajar con el compilador sin problemas.

Para hacerlo, ejecute la aplicación SysEdit que viene con Windows (escoja Ejecutar del Menú Inicio y escriba `sysedit` y pulse Intro). De las ventanas que aparecerán escoja la ventana con el nombre AUTOEXEC.BAT (puede cerrar todas las otras) y al final del archivo añada la línea `SET PATH=%PATH%;C:\PP\BIN\WIN32` suponiendo que ha instalado el compilador al directorio C:\PP (que es el directorio por defecto). En todo el manual se supondrá que el directorio de instalación es este. Guarde el archivo y reinicie el ordenador (no se limite a reiniciar el Windows).

2.2.Compilación del primer programa

Para emplear el compilador FreePascal tendrá que abrir una sesión de Ms-Dos en Windows. En este ejemplo compilaremos uno de los programas de ejemplo que lleva FreePascal (siempre que los haya instalado, claro). Este ejemplo es el típico *Hola mundo*, y se encuentra al directorio `C:\PP\SOURCE\DEMO\TEXT` en el archivo `HELLO.PP`.

Para compilar el archivo escriba :

```
PPC386 HELLO.PP
```

En unos instantes, si todo ha ido bien durante la compilación y el enlazado, obtendrá un archivo ejecutable con el nombre `HELLO.EXE`. Si lo ejecuta obtendrá la salida :

```
Hello world
```

En caso que hubiera obtenido el error de "Comando o nombre de archivo incorrecto" revise que el directorio `C:\PP\BIN\WIN32` se encuentra en la variable `PATH`.

2.3.Buscando un IDE para FreePascal

FreePascal incorpora un entorno de desarrollo integrado (IDE) en modo texto. En la distribución 1.00 de FreePascal el IDE era aún muy inestable y se encontraba aun en fase beta. Por este motivo habrá que buscar algún sustituto para este IDE y que incorpore al menos el resaltado de sintaxis.

Una de las soluciones pasa por emplear el IDE de Turbo Pascal pero también se pueden emplear por ejemplo el *Programmers Notepad* para Windows (`http://www.alternate.demon.co.uk/pn/`), de Echo Software. También está muy bien el IDE totalmente integrado BloodShed's Dev-Pascal (`http://www.bloodshed.net`) bajo licencia GNU (hay que indicar que ya lleva su propia distribución del compilador FreePascal).

En cualquier caso es muy recomendable que el IDE o editor permita :

- Resaltar la sintaxis de Pascal.
- Ejecutar el compilador directamente desde el editor. Consulte la documentación del editor sobre como aprovecharse de esta característica.

3.Primeros pasos con Pascal

3.1.Programas y units

Pascal define dos tipos básicos de archivos que se pueden compilar : los programas y las units. En estos primeros capítulos nos limitaremos a trabajar con programas y dejaremos el tema de creación de units para más adelante.

Los archivos de código fuente que se escriben con un editor (o IDE) conviene que tengan extensión `.PP` o bien `.PAS`. Al compilar programas a Win32 obtendremos archivos `.EXE` mientras que si compilamos units obtendremos archivos `.PPW` que no son ejecutables.

3.2.Estructura básica de un programa

3.2.1.Cabecera

Todos los programas tienen la misma estructura. Para indicar al compilador que se trata de un programa tenemos que escribir la palabra `program` seguida de un espacio y de un nombre (que puede tener letras y números) y que tiene que terminar en un punto y coma. Por ejemplo :

```
program EjemploNumero6;
```

Es importante indicar que el nombre que demos al programa tiene que seguir unas reglas ya que sino en caso contrario no será válido para el compilador :

• El primer carácter tiene que ser una letra y no una cifra ya que en caso contrario el compilador se pensaría que es un numero. Por ejemplo, no es valido el nombre siguiente :

```
program 6Ejemplo;
```

• El único carácter no alfanumérico válido es el guión bajo. No se admiten caracteres como exclamaciones o signos de interrogación, etc.

• Tampoco se admiten vocales acentuadas o letras que no pertenezcan al alfabeto inglés. Por ejemplo la ç y la ñ no se admiten.

• Finalmente, el nombre del programa no tiene que coincidir con otras variables o constantes del programa. Éstas las veremos más adelante.

3.2.2.Utilización de units

En algunos casos nos puede interesar emplear diferentes funciones y procedimientos que están definidos en otros archivos. Esta especie de *almacenes* de rutinas se llaman *units* y hay que explicitarlas cuando queremos emplear alguna de sus rutinas en nuestro programa.

Las units estándar que incorpora FreePascal son las siguientes :

PARA TODAS LAS PLATAFORMAS	
GETOPTS	Permite recuperar los parámetros pasados en la línea de ordenes de un programa.
MMX	Aporta funcionalidad básica para el uso de las extensiones multimedia de los procesadores INTEL.
OBJECTS	Incorpora rutinas para gestionar objetos.
OBJPAS	Se emplea automáticamente cuando se activa el soporte Delphi.
PORTS	Da acceso a los puertos I/O del PC. No se recomienda su uso en Win32.
STRINGS	Permite emplear el tipo de cadena PCHAR.
SYSUTILS	Implementación alternativa de la unit del mismo nombre en Delphi.
TYPINFO	Permite acceder a la información de tipo en tiempo de ejecución, como en Delphi.
PARA WINDOWS 32-BIT	
DOS	Lleva rutinas de emulación de acceso al sistema DOS mediante funciones del sistema de Windows.
CRT	Rutinas básicas de gestión de la pantalla en modo texto.
GRAPH	Gestión básica de gráficos.
WINDOWS	Esta unit permite acceder a las funciones de la API de 32-BIT de Windows.
OPENGL	Accede a les funciones de bajo nivel OpenGL de Windows 32-BIT.
WINMOUSE	Funciones para emplear el ratón en Windows 32-BIT.
OLE2	Implementa las posibilidades OLE de Windows 32-BIT.
WINSOCK	Interfaz a la API winsock de Windows 32-BIT.
SOCKETS	Encapsulación de WINSOCK de forma que sea compatible en Windows 32-BIT y en Linux.

En la mayoría de ejemplos sólo emplearemos la unit CRT para mostrar datos en modo texto. Para incluir una unit en nuestro programa emplearemos la palabra reservada `uses`. Por ejemplo, si queremos incluir las units CRT y GRAPH sólo habrá que añadir la siguiente cláusula debajo de program :

```
uses CRT, GRAPH;
```

Cuando se emplee más de una unit se tendrá que separarlas en comas. La línea siempre hay que finalizarla en punto y coma.

3.2.3.Declaración de constantes

Muchas veces puede ser útil emplear un valor numérico o un texto determinado varias veces en un programa. Para evitar tener que definirlo en cada punto que lo necesitemos emplearemos una referencia única en todo el programa al que llamaremos constante.

Las constantes no varían en todo el programa y se declaran seguidas de la palabra reservada `const`, un signo igual y el valor que representan. Por ejemplo :

```
const
  maximo = 1000;
  e = 2.71828;
  nombre = 'Roger Ferrer';
  letra = 'a';
```

Lo que va después del signo igual recibe el nombre de literal. Un literal es un valor incluido directamente en el código. En el primer caso es el numero entero 1000, la constante `maximo` vale 1000. En el caso de la constante `e`, es el número real `2.71828`. Finalmente `nombre` representa una cadena de caracteres con el texto `Roger Ferrer`.

La declaración de literales se rige por unas normas sencillas :

• Las cifras que no lleven decimales son automáticamente entendidas por el compilador como un entero.

• Las cifras con notación decimal inglesa, o sea, un punto decimal en vez de una coma decimal, son siempre cifras reales.

• Las cadenas de caracteres siempre empiezan y terminan con un apóstrofe o comilla simple ('). Para escribir el signo apóstrofe dentro de una cadena de caracteres hay que duplicar el apóstrofe. Ej : `'Tira p''alante'`

• Si una cadena de caracteres tiene un sólo carácter, el compilador lo entiende también como un tipo carácter. Los caracteres son compatibles con las cadenas de caracteres pero no a la inversa.

• Podemos escribir caracteres con su código ASCII en decimal añadiendo el símbolo # delante del número. Por ejemplo #7 es el carácter BEEP que al escribirse emite un ruido por el altavoz, #65 es la letra A mayúscula. Podemos combinar caracteres escritos así y literales normales, por ejemplo `'L'#65` equivale a `'LA'`.

• Los tipos enteros que empiecen con un signo de dólar $ se interpretarán como cifras hexadecimales. Las cifras que empiecen con un signo de porcentaje % se interpretarán como nombres binarios. En el caso de los hexadecimales se permite el uso de todas las cifras en base 16 (0..9, A..F) y en el caso de los binarios sólo son validas la cifras 0 y 1.

```
const
  NumeroBase10 = 1522;
  NumeroBase16 = $5F2;
  NumeroBase2 = %10111110010;
```

En este caso las tres constantes NumeroBase10, NumeroBase16 y NumeroBase2 representan el mismo número, aunque escrito en bases distintas.

En Pascal, los nombres de constantes es indiferente de que estén en mayúsculas o minúsculas. Por tanto, en el ejemplo anterior maximo, MAXIMO y maXIMo representan lo mismo para el compilador. Hay que ir con cuidado para no duplicar nombres. Por ejemplo, esto es incorrecto.

```
const
  maximo = 1000;
  MAXIMO = 5000;
```

Ya que MAXIMO y maximo son lo mismo y se está redeclarando una constante ya declarada. Esta independencia sintáctica a las mayúsculas está presente en todos los aspectos de Pascal excepto, claro está, en las cadenas de caracteres. Es usual, debido a la notación de C, que las constantes se declaren totalmente en mayúsculas y en los programas se empleen en mayúsculas para distinguirlas de las variables, como veremos más adelante, aunque no sea necesario hacerlo.

3.2.4.Declaración de variables. Tipos de datos fundamentales

Si pensamos en muchas fórmulas matemáticas veremos que emplean variables en su expresión. Por ejemplo el área del triángulo es AT= ½ b h. Cuando queremos calcular el arara de un triángulo dado nos limitamos a sustituir estas variables. En Pascal, y programación ocurre algo parecido.

Pascal exige que se declaren las variables que se emplearán. Las variables pueden modificar su valor a lo largo del programa. Cada variable tiene que ser de un tipo determinado. Los tipos de variables, llamados tipos de datos, se dividen en cuatro grandes grupos fundamentales :

• Numéricos. Enteros o reales.

• Alfanuméricos. Un carácter o una cadena de caracteres.

• Booleanos. Sólo pueden valer cierto o falso.

Los tipos numéricos enteros que incorpora `FreePascal` son los siguientes :

Tipos	**Rango numérico**	**Bytes (Bits)**
Byte	0..255	1 (8)
Shortint	-128..127	1 (8)
Integer	-32768..32767 En modo *compatibilidad Delphi* este tipo es igual a Longint.	2 (16)
Word	0..65535	2 (16)
Longint	-2147483648..2147483647	4 (32)
Cardinal	0..4294967296	4 (32)
Int64	-9223372036854775808..9223372036854775807	8 (64)

También tenemos tipos reales :

Tipos	**Rango real**	**Decimales significativos**	**Bytes (Bits)**
Single	1.5^{-45} .. 3.4^{38}	7-8	4 (32)
Real	5.0^{-324} .. 1.7^{308}	15-16	8 (64)
Double	5.0^{-324} .. 1.7^{308}	15-16	8 (64)
Extended	1.9^{-4951} .. 1.1^{4932}	19-20	10 (80)
Comp	$-2^{64}+1$ .. $2^{63}-1$	19-20	8 (64)

El tipo Comp es un tipo especial de entero con propiedades de real. Los decimales significativos indican cuantos decimales se tienen en cuenta a la hora de hacer cálculos.

Los tipos alfanuméricos que implementa `FreePascal` son los siguientes :

Tipos	**Contenido**	**Bytes**
Char	Un solo carácter	1
ShortString	Una cadena de 255 caracteres	256
`String`[n]	Una cadena de n caracteres $(1 \leq n \leq 255)$	n+1
AnsiString	Nombre ilimitado de caracteres	Concepto no aplicable en este caso

PChar	Cadena finalizada en NULL	Concepto no aplicable en este caso

Los tipos AnsiString y PChar permiten almacenar datos alfanuméricos pero su uso funcionamiento y uso es más especial.

El tipo `String` puede representar un ShortString o un AnsiString en función del modo en qué se encuentre el compilador.

Finalmente, los tipos booleanos implementados son Boolean, ByteBool, WordBool y LongBool. Los tres últimos sólo tienen sentido en programación de la API de Windows y el tipo Boolean es el que se emplea más habitualmente.

Para declarar variables emplearemos la palabra reservada var, los nombres de las variables seguidas de dos puntos y del tipo de variable. Por ejemplo :

```
var
  Base : Integer;
  Altura : Integer;
  NombreUsuario : ShortString;
```

Todo lo que era aplicable a los nombres de las constantes también lo es para los nombre de variable. Por tanto altura, ALTURA y altura serian equivalentes. La declaración anterior se podía haber simplificado de la forma siguiente :

```
var
  Base, Altura : Integer;
  NombreUsuario : ShortString;
```

Podemos declarar dos o más variables del mismo tipo a la vez separandolas entre comas.

Hay que tener en cuenta de que no se pueden emplear las siguientes palabras para nombres de variables o constantes ya que son palabras reservadas para Pascal.

```
absolute        const           export          implementation
abstract        constructor     exports         in
alias           continue        external        index
and             default         false           inherited
array           destructor      far             initialization
as              dispose         file            inline
asm             div             finalization    interface
assembler       do              finally         is
begin           downto          for             label
break           else            forward         library
case            end             function        mod
cdecl           except          goto            name
class           exit            if              near
```

```
new                 private             repeat              try
nil                 procedure           saveregisters       type
not                 program             self                unit
object              property            set                 until
of                  protected           shl                 uses
on                  public              shr                 var
operator            published           stdcall             virtual
or                  raise               string              while
override            read                then                with
packed              record              to                  write
pascal              register            true                xor
popstack
```

En esta lista se incluyen todas las palabras reservadas de FreePascal, Turbo Pascal y Delphi. Aunque la lista parece muy grande, no es nada complicado encontrar nuevos nombres para constantes y variables.

3.2.5.El bloque principal del programa

Una vez hemos declarado la etiqueta program, la clausula uses y las declaraciones de constantes y variables sólo falta el bloque principal de código del programa. El bloque donde se ejecutarán las ordenes. Este bloque, a diferencia de los anteriores, no se puede dejar de poner en un programa Pascal. El bloque principal del programa se define entre las palabras `begin` y `end`. Por ejemplo estas dos líneas ya son un programa válido para el compilador aunque no haga nada :

```
begin
end.
```

Nótese que el end tiene que llevar un punto final indicando que termina el programa. Es entre estas dos palabras que incluiremos nuestro código. Totas las órdenes que se incluyen en este bloque reciben el nombre de sentencias.

3.3.Los primeros programas

Vamos a hacer un programa que calcule el área de un triángulo dadas la base y la altura. El código podría ser el siguiente :

```
program Calcular_Area_del_triangulo;
var
  base, altura, area : real;
begin
  base := 2.4;
  altura := 5.3;
  area := 0.5*base*altura;
```

```
  Writeln(area);
end.
```

Antes de comentar nada obsérvese que todas las líneas de Pascal suelen terminar en un punto y coma. Como excepciones notables tenemos en nuestro caso a `var` y `begin`.

Declaramos las variables base, altura y area de tipo real. Dentro del bloque de código asignamos el valor 2.4 a la variable base mediante el operador de asignación. Este operador `:=` permite asignar valores a las variables. En la siguiente línea se hace lo mismo con la variable altura.

En la variable area se almacena el valor del producto de ½ por la base por la altura. Finalmente esta variable la mostramos en la pantalla mediante la función Writeln.

Las funciones que necesitan parámetros se especifican entre paréntesis. Si tuviéramos que añadir más de un parámetro lo separaríamos entre comas.

El resultado que obtenemos en pantalla no es muy profesional, pero más adelante ya veremos como especificar los decimales que queremos obtener.

```
6.359999999999999E+000
```

Ya que el resultado de la operación `0.5*base*altura;` es un real y Writeln puede mostrar reales podíamos haber simplificado el programa de la forma siguiente :

```
program Calcular_Area_del_triangulo;
var
  base, altura : real;
begin
  base := 2.4;
  altura := 5.3;
  Writeln(0.5*base*altura);
end.
```

Hemos empleado el asterisco * para indicar producto. Pero FreePascal incorpora más operadores aritméticos. Los operadores se distinguen por unarios, que sólo trabajan con un sólo elemento, o binarios, que trabajan con dos elementos.

Operadores unarios	**Operación**
+	Identidad de signo.
-	Inversión del signo.

Operadores binarios	**Operación**	**Tipo de dato resultante**

+	Suma aritmética	Admite operandos reales y enteros. Si los dos operandos son enteros el resultado es un entero. En caso contrario es un real.
-	Resta aritmética	
*	Producto aritmético.	
**	Potencia. Ex 2**3 = 2^3	Sólo admite operandos enteros y el resultado es un entero.
div	División entera	
mod	Residuo de la división entera	
/	División real	Admite operandos enteros y reales. El resultado siempre es un real.

Los operadores tienen las mismas prioridades que en las operaciones matemáticas habituales. Si deseamos calcular 3+4 y multiplicarlo por 5 tendremos que hacer (3+4)*5 = 35 ya que si hacemos 3+4*5 = 23. Los paréntesis se pueden colocar enmedio de la operación para modificar el orden de prioridad de los operadores.

```
begin
  Writeln(3+4*5);
  Writeln((3+4)*5);
  Writeln((3+4)*5+6);
  Writeln((3+4)*(5+6));
end.
```

En este aspecto hay que remarcar que los operadores con mayor prioridad son **, *, /, div y mod. Mientras que la suma y la resta son los que menos tienen. En el caso que todos tengan la misma prioridad la operación se lleva a cabo de izquierda a derecha.

```
Writeln(4*20 div 5);
Writeln(4*(20 div 5));
Writeln((4*20) div 5);
```

En este caso los tres resultados son idénticos y los paréntesis sólo tienen una finalidad esclarecedora. En los dos últimos casos la operación empezará siempre por el elemento que tenga paréntesis o que tenga más que los otros.

3.3.1.Introducción de datos

El programa de cálculo de áreas que hemos hecho antes funciona correctamente pero siempre devuelve el mismo resultado. Por ejemplo, vamos a hacer un programa que calcule la suma de n potencias de r > 1. O sea que nos diga el resultado de $r^0 + r^1 + r^2 ... + r^{n-1} = (r^n - 1) / (r-1)$.

```
program Calcular_n_potencias;
var
  r, n : Integer;
begin
  Writeln('R :');
  Readln(r);
  Writeln('N :');
  Readln(n);
  Writeln('Resultado :');
  Writeln((r**n-1) div (r-1));
end.
```

En el anterior ejemplo hemos empleado la orden Readln para leer una entrada del teclado y almacenarla en la variable especificada en los parámetros de Readln. El usuario tiene que introducir un par de números. Por ejemplo si ejecutamos el programa e introducimos los datos siguientes obtendremos :

```
R :
5
N :
4
Resultado :
156
```

3.3.2.Comentarios en el código

Los programadores pueden situar comentarios en el código que no se compilará. Para indicar al compilador que un elemento es un comentario es necesario rodearlo de llaves { } o de los conjuntos siguientes (* y *). Finalmente si el comentario va desde un punto cualquiera hasta el final de la linea podemos emplear dos barras oblicuas //. No mezcle los caracteres { y } con (* y *) ni tampoco escriba un espacio entre el paréntesis y el asterisco porque entonces no se entiende como un comentario.

```
program Calcular_Area_del_triangulo;
{ Este comentario está hecho con llaves }
var
  r, n : Integer;
(* Este, en cambio, emplea la notación antigua de Pascal *)
begin
  Writeln('R :'); // Desde las 2 barras hasta el final de linea es un comentario
  Readln(r);
  Writeln('N :');
  Readln(n);
  Writeln('Resultado :');
  Writeln((r**n-1) div (r-1));
end.
```

4.Profundizando aspectos

4.1.Trabajando con variables y tipos de datos

Hemos visto en ejemplos anteriores que trabajar con variables es tan sencillo como declararlas y después referirse en el código, por ejemplo, en una asignación o en una operación. Aún así, hay otras cuestiones que debemos conocer cuando trabajemos con variables.

4.1.1.Gestión de las variables

Los tipos de variables que estamos empleando no precisan de ningún tipo de operación de inicialización o de finalización. El compilador se encarga de todos estos aspectos reservando la memoria necesaria para las variables.

4.1.2.Compatibilidad y tipos de variables

Pascal es un lenguaje fuertemente tipificado. Esto quiere decir que las comprobaciones de tipos de datos son bastante estrictas. Por este motivo no podemos asignar cualquier valor a las variables sino variables o literales que sean compatibles. El programa siguiente es incorrecto sintácticamente :

```
var
  cadena : string;
  numero : Integer;
begin
  cadena := 10; // No podemos asignar un numero a un string
  numero := '10'; // No se puede asignar un string a un numero
  cadena := numero; // Tipos incompatibles por asignación
end.
```

Para resolver algunos de los problemas de tipos que se presentan en Pascal tenemos dos formas. Escoger una u otra dependerá de la situación en la que nos encontremos :

- Si los dos datos incompatibles representan una misma estructura de datos podemos probar un typecasting. Por ejemplo, los diferentes tipos de cadenas o de nombres enteros.
- Si los dos datos son de naturaleza distinta podemos recurrir a alguna función que transforme los datos. Por ejemplo, pasar un entero a cadena o un real a entero.

Algunas veces no será necesario realizar ninguna transformación. Es el caso de los enteros. Los enteros son compatibles “de abajo arriba”. O sea, podemos emplear enteros pequeños como el Byte en asignaciones de enteros mayores como Integer o Cardinal. Habrá que tener en cuenta también si el entero

es con signo o sin signo. Hay que observar también que el proceso contrario, "de arriba abajo", no siempre es posible.

```
var
  pequeno : Byte;
  grande : Longint;
begin
  pequeno := 200;
  grande := pequeno;
  Writeln(grande); // 200
  grande := 500;
  pequeno := grande;
  Writeln(pequeno); // 244 !
end.
```

La causa por la cual obtenemos 244 en el último caso es debido al rango del Byte 0..255 que no llega a 500. Los enteros, los caracteres y los booleanos, son tipos de datos llamados ordinales. Esto quiere decir que sus valores tienen un orden cíclico (en los reales, por ejemplo, no tiene sentido hablar de ordinalidad) y por tanto en un Byte después de 255 viene un 0. Por tanto 255 + 1 = 0. Cuando asignamos 500 mediante el Longint resulta que 500 = 256 + 224 = 0 + 224 = 224. O Expresado de otra forma 500 mod 256 = 224. Es importante tener en cuenta este hecho ya que es un error bastante habitual asignar tipos enteros demasiado pequeños que suelen dar comportamientos erráticos y difíciles de detectar. A menos que sea imprescindible (que podría ser), es recomendable emplear Integer o Longint en vez de Byte o Shortint.

4.1.3.Typecasting

En muchos casos trabajar con un tipo u otro de entero no reviste ninguna importancia excepto por el rango en el que trabajemos. Algunas veces es necesario que nuestro entero tenga el tamaño adecuado. Para "amoldar" el tamaño de las variables podemos emplear el typecasting.

Para realizar un typecasting sólo hay que rodear la variable a amoldar con el nombre del tipo de dato al cual queremos ajustar. Los typecasting son habituales cuando se trabaja con clases y objetos. He aquí un ejemplo de typecasting con booleanos :

```
var
  var_bool : Boolean;
begin
  var_bool := Boolean(0);
  Writeln(var_bool); // Escribe FALSE
  var_bool := Boolean(1);
  Writeln(var_bool); // Escribe TRUE
  var_bool := Boolean(12);
  Writeln(var_bool); // Escribe TRUE
```

```
end.
```

Esto es sintácticamente correcto ya que un booleano es nada más que un Byte que se interpreta cierto cuando tiene un valor distinto a cero. Otro typecasting muy usual es el de la conversión de caracteres a su valor ASCII. Por ejemplo :

```
var
  caracter : Char;
begin
  Write('Introduzca un carácter : ');
  Readln(caracter);
  Writeln('Su código ASCII es ', Integer(caracter));
end.
```

La instrucción Write es idéntica a Writeln pero no hace saltar el cursor a la línea siguiente. Write y Writeln permiten concatenar más de un parámetro separados por comas, en este caso un string y un Integer. Esta característica es única de Write, Writeln, Read y Readln aunque sólo se suele emplear en Write y Writeln.

En el caso anterior, si introducimos una A mayúscula obtendremos el valor 65 del código ASCII. Esto también es correcto ya que un carácter no es nada más que un Byte. Escribiendo Integer hacemos que Writeln lo interprete como si de una variable Integer se tratara.

4.1.4.Funciones de conversión de tipo

Qué tenemos que hacer cuando necesitamos convertir una cadena de texto que tiene un número a un entero o convertir un real a string ?

Para esto tendremos que emplear las funciones que incorpora la unit SysUtils. Las funciones que nos interesarán más son IntToStr, StrToInt, FloatToStr, FloatToStrF y FloatToText.

Convertir de entero a string

Nos puede interesar convertir un entero a string ya que muchas funciones sólo permiten strings como parámetros. Además, los strings se pueden concatenar con el operador +.

```
begin
  Writeln('Hola '+'qué tal?');
end.
```

En el caso de Writeln no sería necesario pero muchas funciones sólo admiten un string y esta es una forma sencilla de añadir más de una o como veremos añadir enteros.

La función IntToStr toma como parámetro un entero y devuelve un String. Veamos el caso siguiente.

```
uses SysUtils; // IntToStr está definida en SysUtils
var
  cadena : string;
  anyos : Integer;
begin
  Write('Cuantos años tienes ? '); Readln(anyos);
  cadena := 'Tienes ' + IntToStr(anyos) + ' años';
  Writeln(cadena);
end.
```

El punto y coma del final de cada sentencia nos permite agruparlas en una sola línea aunque no es recomendable, generalmente. Obsérvese la enorme diferencia entre anyos y 'años' pues la primera es una variable y la segunda una cadena de texto..

Ya que IntToStr devuelve un string lo puedo operar como si de un string se tratara, por ejemplo concatenandolo con el operador +.

Convertir de string a entero

En este caso la conversión no es tan simple pues corremos el riesgo de que la cadena de texto no pueda ser convertida a un entero. En este caso la función StrToInt lanzará una excepción. La gestión de excepciones la veremos más adelante, a pesar de esto veamos un ejemplo sencillo :

```
uses SysUtils;
var
  sum_a, sum_b, total : string;
begin
  sum_a := '100';
  sum_b := '400';
  total := IntToStr( StrToInt(sum_a) + StrToInt(sum_a) );
  Writeln(total);
end.
```

Obsérvese la tercera sentencia que tiene una estructura muy interesante. Primero convertimos los string sum_a, sum_b a enteros mediante StrToInt. Ahora la suma es aritmética ya que los dos valores son enteros. Una vez hecha la suma transformamos de nuevo el entero en un string y lo almacenamos en una variable de tipo string. En este ejemplo, los enteros contienen números válidos que pueden ser convertidos a enteros sin problemas.

Convertir un real a string

Para convertir un real a string tenemos tres funciones : FloatToStr, FloatToStrF i FloatToText. Las dos últimas son más complejas ante la simplicidad de FloatToStr..

```
uses SysUtils;
var
  prueba : real;
  cadena : string;
begin
  prueba := 10.5;
  cadena := 'El valor es '+FloatToStr(prueba);
  Writeln(cadena);
end.
```

4.1.5.Trabajar con constantes

A las constantes se les puede aplicar muchas de las cuestiones expuestas para las variables. Mientras la función de conversión no modifique el parámetro, se verá más adelante cómo, es posible sustituir una variable por una constante.

```
uses SysUtils;
const
  NUMERO = 1000;
begin
  Writeln('El numero es '+ IntToStr(NUMERO));
end.
```

4.1.6.Asignaciones aritméticas de variables

Antes hemos visto el operador := que es el operador de asignación por excelencia. Habitualmente, se emplean construcciones parecidas que se han simplificado en otros operadores de asignación. La tabla siguiente muestra la lista de operadores de asignación aritméticos de FreePascal. Estos operadores están disponibles sólo cuando añadimos el parámetro `-Sc` al compilador.

Expresión	Operación	Equivalencia
`x += y;`	Suma “y” a “x” y lo guarda en “x”.	`x := x + y;`
`x -= y;`	Resta “y” a “x” y lo guarda en “x”.	`x := x - y;`
`x *= y;`	Multiplica “x” por “y” y lo almacena en “x”.	`x := x * y;`
`x /= y;`	Divide “x” entre “y” y lo almacena en “x”.	`x := x / y;`

Veamos un ejemplo :

```
var
  a : Integer;
begin
  a := 10;
  a+=1;
  Writeln(a); // 11
  a := a + 1;
  Writeln(b); // 12
end.
```

4.2.Expresiones e instrucciones

Hasta ahora hemos visto muchos ejemplos de expresiones (por ejemplo a*b + c*d) combinadas con instrucciones (por ejemplo a:= 10;). Una expresión es siempre algo que tiene un valor, ya sea una operación matemática simple como las que hemos visto antes. El valor de la expresión viene determinado por los elementos que intervienen en la operación que suelen ser variables, literales, constantes y operadores. Así el producto de dos enteros es un entero mientras que un entero por un real es una expresión real.

Las instrucciones son acciones que se realizan en el programa. Las instrucciones en Pascal reciben el nombre de sentencias. Hasta ahora hemos visto dos sentencias muy habituales : las sentencias de asignación y la sentencia de llamada de funciones y procedimientos. Pascal, afortunadamente, dispone de un conjunto de sentencias más elaborado que veremos en el siguiente capítulo.

5.Sentencias estructuradas

5.1.Introducción

Las sentencias estructuradas, basadas en más de una sentencia, son de hecho sentencias con construcciones especiales que permiten, básicamente, alterar la ejecución secuencial de las sentencias de un programa.

5.2.La estructura condicional if

Antes hemos visto por ejemplo calcular $r^0 + r^1 + r^2 ... + r^{n-1} = (r^n - 1) / (r-1)$ cuando $r > 1$. En el código anterior no hemos comprobado en ningún momento que r fuera mayor que 1. Si r = 1 el programa fallaba ya que se produce una división entre cero. Para evitar que el usuario introduzca un valor menor o igual que 1 tendremos que comprobar es mayor que 1. Con la finalidad de comprobar la certeza o falsedad de una condición Pascal (y muchos lenguajes de programación) dispone de la sentencia `if`.

La estructura de if es la siguiente :

```
if expresionBooleana then sentencias;
```

Es importante que la expresión entre if y then sea una expresión booleana. Vamos a ver el código del programa una vez modificado :

```
program Calcular_n_potencias;
var
  r, n : Integer;
begin
  Writeln('R :');
  Readln(r);
  if r > 1 then
  begin
    Writeln('N :');
    Readln(n);
    Writeln('Resultado :');
    Writeln((r**n-1) div (r-1));
  end;
end.
```

Obsérvese que en este caso la expresión booleana r > 1 sólo es cierta cuando r es mayor que 1. Si es cierta se ejecuta lo que se encuentra después del then. En este caso es un bloque de sentencias. En Pascal los bloques de sentencias, aparte del principal, van entre un begin y un end con punto y coma. Si r > 1 no es cierto entonces no se ejecutará nada más.

Habitualmente, precisaremos de código que se ejecutará cuando la condición no se cumpla. Para hacerlo hay que emplear la estructura siguiente :

```
if expresionBooleana then sentencias else sentencias;
```

Por ejemplo modificando el programa anterior tendríamos :

```
program Calcular_n_potencias;
var
  r, n : Integer;
begin
  Writeln('R :');
  Readln(r);
  if r > 1 then
  begin
    Writeln('N :');
    Readln(n);
    Writeln('Resultado :');
    Writeln((r**n-1) div (r-1));
  end
  else
  begin
    Writeln('R tiene que ser un valor superior a 1');
  end;
end.
```

Si el usuario introduce un valor menor o igual a 1 entonces obtiene el mensaje que "R tiene que ser un valor superior a 1". Obsérvese también que sólo en este caso end tiene que ir sin ningún punto y coma pues va seguido de else.

Si el número de sentencias de then o else es sólo una es posible omitir begin y end. Pero sólo en este caso. Si deseamos que se ejecute más de una sentencia no tendremos más remedio que emplear begin y end. En cualquier caso, el uso de begin y end es mejor pues evita muchos errores típicos.

```
var
  numero : Integer;
begin
  Write('Introduzca un número : '); Readln(numero);
  if numero mod 5 = 0 then Writeln('Es divisible por 5')
  else Writeln('No es divisible por 5');
end.
```

Observe que tampoco la sentencia anterior a else tiene que llevar un punto y coma. Un número es divisible por otro si su residuo (operación mod) es cero.

De las estructuras condicionales anteriores hemos visto unos pocos operadores llamados relacionales que permiten construir expresiones booleanas más o menos complicadas.

5.2.1.Operadores lógicos booleanos

Pascal incorpora un conjunto de operadores booleanos lógicos que permiten modificar las expresiones booleanas. Las tres operaciones booleanas que se pueden aplicar a expresiones booleanas son las clásicas de la lógica : `not` (negación), `and` (conjunción) y `or` (disyunción).

Not es un operador unario (con un solo operando) y devuelve el valor contrario a la expresión booleana.

	not
true (verdadero)	false (falso)
false (falso)	true (verdadero)

Los operadores and y or son operadores binarios, trabajan sobre dos operandos. And devuelve true si los dos operandos son true y or retorna true si alguno de los operandos lo es.

		and	**or**
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Finalmente disponemos de un tercer operador binario `xor`, disyunción exclusiva, este devuelve cierto cuando los dos elementos tienen valores diferentes.

		xor
false	false	false
false	true	true
true	false	true
true	true	false

Los operadores booleanos combinados con los operadores relacionales, de los cuales hemos visto antes el operador > y el =, permiten construir expresiones booleanas muy eficientes. Los operadores relacionales relacionan dos elementos y devuelven una expresión booleana.

Operador	Tipos de datos	Resultado
=	Todos los ordinales, reales y string.	Enteros y Reales. Cierto si los dos operandos son idénticos.
		Caracteres. Si el valor ASCII de los dos caracteres es el mismo.
		Booleanos. Si las dos expresiones booleanas tienen el mismo valor.
		String. Si las dos cadenas comparadas son idénticas incluso en las mayúsculas.
<>	Todos los ordinales, reales y string.	Devuelve true cuando = ha devuelto false y viceversa.
<	Todos los ordinales, reales y string.	Enteros y reales. Devuelve cierto si el primer operando es menor que el segundo.
		Devuelve cierto si el valor ASCII del primer carácter es menor al valor del segundo.
		String. Devuelve cierto si la primera cadena es alfabéticamente anterior a la segunda. El alfabeto de referencia es el ASCII[1].
		Booleanos. Si el primer operador es false y el segundo true entonces el resultado es cierto. Téngase en cuenta de que false es menor que true y true es mayor que false ya que los booleanos son ordinales.
>	Todos los ordinales, reales y strings.	Enteros y reales. Devuelve cierto si el primer operando es mayor al segundo.
		Caracteres. Devuelve cierto si el valor ASCII del primer operando es mayor al segundo.
		String. Devuelve cierto si la primera cadena es alfabéticamente posterior a la segunda. El alfabeto de referencia es el ASCII.

[1]A consecuencia de esto los caracteres no ingleses como la ç o la ñ se consideran siempre los últimos. Por ejemplo 'caña' < 'cata' es falso. En esta cuestión también están incluidas las vocales acentuadas y los caracteres tipográficos.

		Booleanos. Si el primer operador es true y el segundo false entonces el resultado es cierto.
<=	Todos los ordinales, reales y string.	Enteros y reales. Devuelve cierto si el primer operando es menor o igual al segundo.
		Caracteres. Devuelve cierto si el valor ASCII del primer operando es menor o igual al segundo.
		String. Devuelve cierto si la primera cadena es alfabéticamente anterior o idéntica a la segunda. El alfabeto de referencia es el ASCII.
		Booleanos. Sólo devuelve falso si el primer operador es true y el segundo false. En otros casos devuelve cierto.
>=	Todos los ordinales, reales y string.	Enteros y reales. Devuelve cierto si el primer operando es mayor o igual al segundo.
		Caracteres. Devuelve cierto si el valor ASCII del primer operando es mayor o igual al segundo.
		String. Devuelve cierto si la primera cadena es alfabéticamente posterior o idéntica a la segunda. El alfabeto de referencia es el ASCII.
		Booleanos. Sólo devuelve false si el primer operando es false y el segundo es true. En otros casos es siempre cierto.

Vamos a ver ejemplos de combinación de operadores relacionales. Por ejemplo, para determinar si un numero es $0 \leq x \leq 10$ (mayor o igual que cero y menor o igual que diez). Podríamos hacerlo de la forma siguiente :

:

```
if numero >= 0 then
begin
  if numero <= 10 then
  begin
    ...
  end;
```

```
end;
```

Este ejemplo funcionaría perfectamente pero es mucho mejor combinar operadores booleanos y relacionales para obtener un código más eficiente :

```
if (numero >= 0) and (numero <= 10) then
begin
  ...
end;
```

En este caso hemos empleado el operador booleano and ya que nuestro número es "mayor o igual que cero **y** menor o igual que 10". Obsérvese que en este caso se necesitan paréntesis para separar las diferentes operaciones ya que los operadores relacionales tienen la misma prioridad que los booleanos y es necesario que los relacionales sean primeros pues se intentaría hacer un and con un numero y no es un booleano.

Supongamos que disponemos de una variable de tipo carácter. Queremos comprobar que la variable contiene una 'a' minúscula o una 'A' mayúscula, entonces una forma de hacerlo es :

```
if (Caracter = 'a') or (Caracter = 'A') then
begin
  ...
end;
```

Finalmente, como ejemplo, vamos a definir la operación xor con and y or. En este caso las dos estructuras siguientes son equivalentes :

```
var
  a, b : Boolean;
begin
  if a xor b then
  begin
    ...
  end;

  if ((not a) or b) or (a or (not b)) then
  begin
    ...
  end;
end;
```

5.3.La estructura iterativa for

Muchas veces nos interesará realizar una misma o parecida operación dentro de un conjunto limitado de valores. Con esta finalidad Pascal posee la estructura `for`.

Para poder realizar la estructura for es necesario emplear una variable contadora que variará en 1 y que tiene que ser de tipo entero necesariamente. Veamos un ejemplo simple :

```
var
  numero : Integer;
begin
  for numero := 1 to 4 do
  begin
    Writeln('Repetición número ', numero);
  end;
end.
```

La salida del programa es la que sigue :

```
Repetición número 1
Repetición número 2
Repetición número 3
Repetición número 4
```

Como se ve, el código de dentro del bucle se ha ejecutado empezando desde el valor 1 de la variable numero hasta el valor 4, momento en el cual el bucle termina.

En el caso anterior podíamos haber escrito este código equivalente pues el bloque begin y end sólo tiene una sentencia, pero sólo en este caso :

```
for numero := 1 to 4 do Writeln('Repetición número ', numero);
```

Obsérvese que si hace **`for`** `numero := 1` **`to`** `4` **`do;`** con punto y coma después de do, el compilador entenderá que es una sentencia vacía y no hará nada más que incrementar la variable, pero nada más.

Si en vez de incrementar la variable queremos decrementarla hay que cambiar `to` por `downto` a la vez que cambiar el origen y el final.

```
for numero := 4 downto 1 do Writeln('Repetición número ', numero);
```

Si los parámetros origen y final del contador son incorrectos, es decir que el origen es superior al final y tenemos un to, el for ya ni empezará y continuará con la sentencia siguiente después del for.

```
var
  contador : Integer;
begin
  for contador := 9 to 0 do
  begin
    Writeln('Hola'); // Esto no se ejecutará nunca
  end;
end.
```

Como ejemplo final vamos a codificar un programa que calcule la suma de potencias de forma manual y que compruebe que el valor coincide con el de la fórmula antes mostrada, $r^0 + r^1 + r^2 ... + r^{n-1} = (r^n -1) / (r-1)$.

```
program Calcular_n_potencias;
var
  r, n, i, suma : Integer;
begin
  Writeln('R :');
  Readln(r);
  if r > 1 then
  begin
    Writeln('N :');
    Readln(n);
    Writeln('Resultado :');
    Writeln((r**n-1) div (r-1));
    // Método mecánico
    suma := 0;
    for i := 0 to n-1 do
    begin
      suma := suma + r**i;
    end;
    Writeln('Resultado Mecánico :');
    Writeln(suma);
  end
  else
  begin
    Writeln('R tiene que ser un valor mayor que 1');
  end;
end.
```

5.4.Estructuras iterativas condicionales

Las estructuras iterativas condicionales basan su potencial en que el bucle finaliza en función del valor que toma una expresión booleana.

5.4.1.El bucle while..do

Este bucle no finaliza hasta que la condición que ha permitido iniciarlo se vuelve falsa. La estructura de `while..do` es la siguiente.

```
while expresionBooleana do Sentencia;
```

Veamos de ejemplo, el programa siguiente. Este programa no finaliza hasta que el usuario no ha introducido un punto como cadena de entrada.

```
var
  cadena : string;
begin
  cadena := ''; // Una cadena vacía
  while cadena <> '.' do
  begin
    Write('Escriba algo (Un punto para terminar) : ');
    Readln(cadena);
  end;
  Writeln('Fin del programa');
end.
```

El bucle while ejecuta la sentencia o sentencias hasta que la condición del while se vuelve falsa. Hasta que no se ha terminado un ciclo no se evalúa de nuevo la condición, por tanto, aunque enmedio del código la condición ya sea falsa se seguirán ejecutando sentencias hasta que al finalizar el bloque se tenga que repetir el bucle. Entonces la evaluación daría falso y el código seguiría al punto siguiente de después de la sentencias.

En el ejemplo anterior, es una expresión booleana que se vuelve falsa cuando cadena es igual a un punto. Cuando el usuario introduce un punto, el bucle ya no se repite y el usuario ve impreso en la pantalla “Fin del programa”.

Si la condición del bucle ya fuera falsa al iniciarse por primera vez la sentencia while, el bucle ya no empezaría. El código siguiente, por ejemplo, no se ejecutaría nunca :

```
while false do
begin
  Writeln('Esto no se ejecutará nunca');
end;
```

El literal false denota una expresión booleana falsa, por lo que el bucle no empezará nunca. Vemos el caso siguiente :

```
while true do
begin
  Writeln('Esto no terminará nunca...');
end;
```

En este caso la expresión es siempre cierta por tanto el bucle sería infinito. De momento no sabemos como salir de un bucle infinito. Es importante asegurase de que la condición de salida se cumplirá alguna vez cuando diseñemos el algoritmo.

5.4.2.El bucle repeat..until

El bucle `repeat..until` es más o menos el contrario del bucle while..do. El bucle repeat..until, va ejecutando sentencias hasta que la condición del bucle es cierta. Otra diferencia se encuentra en el hecho de que la evaluación de la condición del bucle se hace al final del bucle y no al principio de éste. Esto provoca que el código se ejecute al menos una vez antes de salir del bucle. Cuando la condición del bucle se cumple entonces éste termina.

Algunos autores consideran el uso de repeat..until inapropiado para una buena programación. En cualquier caso, lo cierto, es que toda estructura de tipo repeat..until es convertible en una estructura de tipo while..do. Veamos un ejemplo de repeat..until en el siguiente programa :

```
var
  nombre : string;
begin
  repeat
    Write('Escriba su nombre : ');
    Readln(nombre);
  until nombre<>'';
end.
```

En este caso podemos observar que si el usuario pulsa Intro sin haber introducido nada la variable nombre no es diferente de la cadena vacía ('') y por tanto el bucle vuelve a empezar.

Tal como hemos comentado anteriormente todos los repeat..until son transformables a while..do. La versión con while..do del programa anterior sería la siguiente :

```
var
  nombre : string;
begin
  nombre := '';
  while nombre = '' do
  begin
    Write('Escriba su nombre : ');
    Readln(nombre);
  end;
end.
```

En este caso hay que asegurarse de que el código del bucle se ejecutará al menos una vez. Para esto, realizamos la asignación nombre := '' para que la condición del bucle `while` se cumpla al menos la primera vez.

5.4.3.Las funciones break y continue.

Algunas veces, necesitaremos alterar el flujo de ejecución de bucles for, while o repeat. Para este cometido disponemos de las funciones break y continue.

La función break hace que el programa salga del bucle y la función continue hace que el bucle procese la siguiente iteración sin terminar la iteración actual. Estas funciones sólo se pueden emplear dentro de bucles for, while y repeat y no se pueden emplear fuera de estos ámbitos.

```
for numero := 1 to 10 do
begin
  Writeln('Hola #', numero);
  Continue;
  Writeln('Adiós');
end;

while true do
begin
  ...
  if Salir then break;
end;
```

En el primer caso 'Adiós' no se verá impreso nunca en la pantalla pues el bucle continúa con la siguiente iteración aunque el bloque de sentencias no haya terminado.

En el segundo ejemplo podemos salir del bucle infinito mediante una condición de salida Salir, que es booleana, y la función break.

Hay que tener en cuenta de que aunque es posible poner sentencias después de un break o continue, si estos forman parte de un mismo bloque de sentencias éstas ya no se ejecutarán, tal como hemos visto en el ejemplo anterior.

5.5.El selector case

Muchas veces necesitaremos modificar el comportamiento de un programa en función a los valores que toma una variable. Con lo que conocemos de Pascal, la única forma de hacerlo sería mediante if. Pero si la variable en cuestión puede tomar varios valores y necesitamos diferentes comportamientos entonces la sentencia if es algo limitada. Siempre podríamos imaginar el caso siguiente :

```
var
```

```
  diasemana : Integer;
begin
  Write('Introduzca el día de la semana : ');
  Readln(diasemana);
  if diasemana > 7 then begin Writeln('Número no válido') else
    if diasemana = 1 then begin Writeln('Lunes') else
      if diasemana = 2 then begin Writeln('Martes')
                        .
                        .
                        .
                    if diasemana = 7 then begin Writeln('Domingo');
end.
```

Como se adivinará, esta no es la mejor forma de resolver este problema y termina por confundir al programador. Para este tipo de casos Pascal posee la estructura selectora `case`. Case sólo puede trabajar con ordinales (booleanos, caracteres y enteros) pero es una forma muy elegante de implementar lo que queremos. La estructura de case es algo más enrevesada que las anteriores pero muy sencilla a la larga :

```
Case expresionOrdinal of
  Constante1 : Sentencia1;
  Constante2 : Sentencia2;
        .
        .
        .
  ConstanteN : SentenciaN;
else SentenciaElse;
end;
```

Pasemos a ver un ejemplo de como se implementaría el problema de los días de la semana con un case :

```
var
  diasemana : Integer;
begin
  Write('Introduzca el día de la semana : ');
  Readln(diasemana);
  case diasemana of
    1 : begin
          Writeln('Lunes');
        end;
    2 : begin
          Writeln('Martes');
        end;
    3 : begin
          Writeln('Miercoles');
        end;
    4 : begin
          Writeln('Jueves');
```

```
        end;
    5 : begin
          Writeln('Viernes');
        end;
    6 : begin
          Writeln('Sábado');
        end;
    7 : begin
          Writeln('Domingo');
        end;
    else
        begin
          Writeln('Número no válido');
        end;
  end;
end.
```

La estructura introducida por else se ejecutará si el valor de diasemana no tiene un selector, por ejemplo si introducimos un 0 o un 8.

Hay que tener en cuenta varias cuestiones a la hora de escribir un case. Los valores ordinales de cada selector tienen que ser valores literales o valores constantes o una expresión evaluable en tiempo de compilación. Así, no se permiten las variables. Por ejemplo :

```
var
       Numero1, Numero2 : Integer;
const
       Numero4 = 4;
begin
       Numero1 := 3;
       case Numero2 do
             Numero1 : begin
                            ... // Esta construcción no es valida
                       end;
             Numero4 : begin
                            ... // Esta sí que lo es
                       end;
             Numero4 + 1 : begin
                               ... // Esta también lo es
                           end;
       end;
end.
```

Numero4+1 es válido pues es una expresión evaluable en tiempo de compilación, en nuestro caso tiene el valor 5. Como se puede ver, la estructura else se puede omitir sin ningún problema. En este caso si Numero2 tiene un valor sin selector no se hace nada. En una estructura case es posible especificar más de un tipo ordinal para cada selector, separándolos entre comas, o bien especificando un rango ordinal para el selector, separados con dos puntos seguidos (..).

```
Case Numero of
    0..10 : begin
              // Esto se ejecutaría con numero del 0 al 10 incluidos
            end;
    11, 13 : begin
               // Sólo los valores 11 y 13
             end;
    12 : begin
           // Sólo el valor 12
         end;
    14..80, 82..90 : begin
                          // Los valores de 14 a 80 y de 82 a 90
                     end;
    81, 91..100 : begin
           // El valor 81 y del 91 al 100
         end;
end;
```

No importa el orden en el que escribamos los selectores. Lo importante es que no se repitan, si fuera el caso el compilador nos advertiría del fallo. Este es uno de los motivos por los cuales los valores de los selectores tienen que ser evaluables en tiempo de compilación y no de ejecución.

En todos los ejemplos anteriores hemos empleado enteros ya que son ordinales por excelencia pero también podemos emplear booleanos y caracteres, que como sabemos también son ordinales.

```
uses Crt;
var
 caracter : Char;
begin
 Write('Introduzca un carácter...');
 caracter := Readkey;
 case caracter of
  'A'..'Z', 'a'..'z' : begin
                          Writeln('Es una letra');
                       end;
  '0'..'9' : begin
               Writeln('Es un número');
             end;
 else begin
        Writeln('Otro tipo de carácter');
      end;
end.
```

La función ReadKey es una función de la unit Crt que nos devuelve el carácter de la tecla que el usuario ha pulsado.

6.Tipos de datos estructurados

6.1.Los datos estructurados

Hasta el momento hemos trabajado todo el rato con tipos de datos básicos de Pascal : booleanos, caracteres, string, enteros y reales. La mayoría de aplicaciones de la vida real no emplean datos de uno en uno sino que emplean grupos de datos. Para resolver este problema Pascal incorpora un conjunto de datos estructurados : array (vectores o tablas), registros y conjuntos. También son tipos de datos estructurados las clases y los objetos pero que debido a su complejidad trataremos en otros capítulos. Finalmente, los archivos también son un tipo especial de datos estructurados que trataremos aparte.

6.2.Declaraciones de tipo

Antes hemos hablado de tipos fundamentales, pero también es posible definir nuevos tipos de datos que podremos utilizar en nuestras variables. Con la palabra reservada `type` podemos declarar nuevos tipos que podremos emplear cuando declaremos variables, de forma que estas variables sean del tipo definido. Veamos un ejemplo asaz inútil pero que servirá para introducirse a los tipos de datos. Los tipos se tendrían que declarar antes de las variables pero dentro de la declaración de tipo no es necesario seguir un orden especial.

```
type
 Numero = Integer;
 NumeroNoIdentico = type Integer;
var
  N1 : Numero; // Esto es lo mismo que un Integer
  N2 : NumeroNoIdentico; // Esto no es exactamente un Integer aunque funciona igual
begin
  ...
end.
```

Como se puede ver, la declaración de un tipo es semejante a la declaración de una constante. En el primer caso Numero no es más que otro nombre para Integer mientras que en el segundo caso NumeroNoIdentico es un símbolo nuevo para el compilador. Este último caso sirve para forzar al compilador que sea capaz de distinguir entre un Integer y NumeroNoIdentico mientras que Numero es imposible de distinguir de Integer. Es un caso algo especial y se emplea pocas veces. Sólo veremos este ejemplo.

6.3.Enumeraciones

El tipo enumerado permite construir datos con propiedades ordinales. La característica del tipo enumerado es que sólo puede tomar los valores de su enumeración.

```
type
 TMedioTransporte : (Pie, Bicicleta, Moto, Coche, Camion, Tren, Barco, Avion);
var
  MedioTransporte : TMedioTransporte;
begin
  MedioTransporte := Camion;
  { Sólo es posible asignar a identificadores de la enumeración }
  case MedioTransporte of // Es posible ya que un enumerado es un tipo ordinal
    Pie : begin
          end;
    Bicicleta : begin
                end;
          ...
  end;
end.
```

Por convención, los tipos de datos se suelen declarar con una T seguido del nombre. De esta forma, los identificadores que empiezan por T se entiende que son tipos y no variables.

Las enumeraciones se declaran con paréntesis y un seguido de identificadores separados por comas la utilidad más importante de las enumeraciones se encuentra especialmente en los conjuntos.

6.4.Conjuntos

Un conjunto es una colección de elementos de un mismo tipo ordinal. De esta forma podemos añadir y suprimir elementos de un conjunto, comprobar si un elemento determinado pertenece al conjunto, etc.

Los conjuntos, además, permiten realizar operaciones propias de conjuntos como la unión, con el operador suma, o la intersección con la operación producto.

Los literales de conjuntos se expresan con claudátores y los identificadores del conjunto. Si deseamos más de uno hay que separarlos entre comas. El literal de conjunto vacío es [].

Dado el conjunto siguiente de comida rápida y dos variables de tipo de este conjunto definimos las operaciones siguientes :

```
type
  TJunkFood = (Pizza, Donut, Frankfurt, Burguer); // Enumeración
  TSetJunkFood = set of TJunkFood; // Conjunto de la enumeración anterior
var
  JunkFood, JunkFood2 : TSetJunkFood;
```

Operación	Sintaxis	Qué hace?

Asignación directa	`JunkFood := [Pizza, Donut];`
	Si queremos el conjunto vacío :
	`JunkFood := [];`
		Asigna un conjunto de valores al conjunto de forma que estos pertenezcan a él.

Unión de conjuntos o elementos.	Si añadimos elementos :
	`JunkFood := JunkFood + [Burguer];`
	Si unimos conjuntos del mismo tipo :
	`JunkFood := JunkFood + JunkFood2;`
		Une un conjunto con otro de forma que el resultado es el conjunto unió.

Diferencia	Si suprimimos elementos :
	`JunkFood := JunkFood - [Burguer];`
	Suprimimos los elementos de JunkFood2 que hay en JunkFood :
	`JunkFood := JunkFood - JunkFood2;`
		Suprime los elementos del segundo conjunto que se encuentran en el primero y devuelve el conjunto resultado.

Intersección	Intersección de un conjunto y un literal :
	`JunkFood := JunkFood * [Burguer];`
	(Si en JunkFood no hubiera Burguer obtendríamos el conjunto vacío).
	Intersección de dos conjuntos.
	`JunkFood := JunkFood * JunkFood2;`
		Devuelve un conjunto que contiene los elementos que pertenecen a ambos conjuntos intersecados.

Para comprobar que un elemento pertenece a un determinado conjunto hay que emplear el operador `in` :

```
if Burguer in JunkFood then { este operador devuelve cierto
                              si en JunkFood hay el elemento Burguer }
begin
  ...
end;
```

6.5.Arrays o vectores

Llamamos array, vector o tabla a una estructura de datos formada por un conjunto de variables del mismo tipo a los cuales podemos acceder mediante uno o más índices.

Se declaran de la forma siguiente :

```
array [minElemento..maxElemento] of tipoDato;
```

Por ejemplo, en el caso siguiente :

```
var
 MiArray : array [1..9] of Integer;
```

Hemos declarado un array de 9 elementos, del 1 al 9, al cual podemos acceder en un momento determinado del programa de la forma siguiente :

```
begin
  MiArray[2] := 100;
  MiArray[3] := 23;
  MiArray[4] := MiArray[2] + MiArray[3];
end;
```

Hay que ir con cuidado a la hora de leer un elemento del array porque puede ser que nos estemos leyendo un punto que no está en el array :

```
begin
  MiArray[10] := 4;  // Error !!!
end.
```

Para saber el elemento máximo y mínimo de un array podemos emplear las funciones High y Low sobre el array :

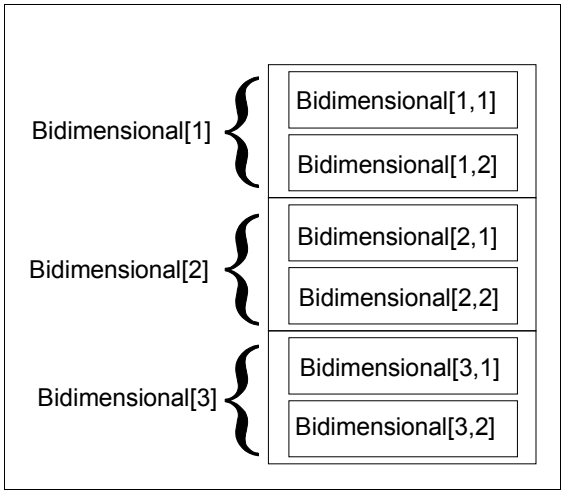
```
begin
  Writeln(Low(MiArray));     // Devuelve 1
  Writeln(High(MiArray));    // Devuelve 9
end.
```

Podemos definir arrays multidimensionales concatenando dos rangos, por ejemplo, definiremos dos arrays de Integer. El primero será bidimensional de 3x2 y el segundo tridimensional de 4x3x2.

```
var
  Bidimensional : array [1..3, 1..2] of Integer;
  Tridimensional : array [1..4, 1..3, 1..2] of Integer;
```

Llegados aquí puede ser algo complicado entender como es un array bidimensional o un array multidimensional. Es fácil de interpretar. Cada elemento del array indicado por el primer índice es otro array el cual podemos acceder a sus elementos mediante la combinación del primer índice y el segundo.

Por ejemplo si hacemos Bidimensional[2] estamos accediendo al segundo array de 2 elementos. Podemos acceder al primer elemento de este array de 2 elementos mediante la construcción siguiente :




```
Bidimensional[2, 1]
```

En este caso, el segundo índice sólo es valido para los valores 1 y 2. Observese que si hace Low y High directamente a Bidimensional obtendrá 1 y 3, respectivamente, mientras que si los aplica a Bidimensional[1] obtendrá 1 y 2.

En el caso de que necesitemos asignar el contenido de un array a otro sólo es posible cuando tengan el mismo tipo estrictamente, hay que tener en cuenta que para el compilador igual declaración no implica el mismo tipo. Véase el programa siguiente :

```
program CopiarArray;
type
  TMiArray : array [0..3] of Integer;
var
  Array1 : TMiArray;
  Array2 : TMiArray;
  Array3 : array [0..3] of Integer;
  Array4 : array [0..3] of Integer;
  Array5, Array6 : array [0..3] of Integer;
begin
  Array1 := Array2; // Compatible ya que el tipo es el mismo
  Array3 := Array4; { Incompatible, tiene la misma declaración pero diferente
                      tipo. Esto da un error de compilación }
  Array5 := Array6; { Compatible, la declaración obliga al compilador a entender
                      que las dos variables son idénticas }
end.
```

En los ejemplos hemos empleado rangos numéricos con enteros positivos pero también es posible definir el rango del array con números negativos como por ejemplo [-2..2] o bien [-5..-3]. Es importante recordar que el rango inicial no puede ser nunca superior al final. Por tanto un array con rango [5..3] es incorrecto.

6.5.1.Los string y los arrays

Es posible acceder a los caracteres de un string (tanto si es ShortString, AnsiString o String[n]) mediante una sintaxis de array. Los string están indexados en cero pero el primer carácter se encuentra en la posición 1.

```
Cadena := 'Hola buenos días';
Writeln(Cadena[2]); {o}
Writeln(Cadena[14]); {í}
```

El nombre de caracteres de una cadena se puede obtener con la función Length. El valor que devuelve es el nombre de caracteres dinámicos, o sea el nombre de caracteres que tiene la cadena. En cambio no devuelve el máximo de caracteres que puede obtener la cadena :

```
var
 Cadena : string[40];
begin
  Cadena := 'Hola buenos días';
  Writeln(Length(Cadena)); // Devuelve 16
end.
```

Para saber la longitud máxima de la cadena hay que emplear la función High. La función Low siempre devuelve cero con una cadena. Obsérvese que los elementos de una cadena son caracteres y por tanto podemos asignar a un Char el elemento de una cadena.

6.6.Registros o records

Un registro es un tipo de dato que contiene un conjunto de campos que no son más que un conjunto de identificadores reunidos bajo un mismo nombre.

La declaración de un registro es de la forma siguiente :

```
record
  Variable1 : Tipo1;
          .
          .
          .
  VariableN : TipoN;
end;
```

Por ejemplo si queremos almacenar datos referentes a un cliente podemos emplear el registro siguiente :

```
type
```

```
  TDatosCliente = record
    Nombre : string;
    Apellidos : string;
    NumTelefono : Integer;
  end;
```

Ahora sólo hay que declarar una variable de este tipo y emplear sus campos :

```
var
  DatosCliente : TDatosCliente;
begin
  Writeln('Introduzca los datos del cliente');
  Write('Nombre : '); Readln(DatosCliente.Nombre);
  Write('Apellidos : '); Readln(DatosCliente.Apellidos);
  Write('Núm. Teléfono : '); Readln(DatosCliente.NumTelefono);
end.
```

Como se puede ver, empleamos el punto para distinguir con qué variable queremos trabajar. Así `DatosCliente.Nombre` representa el campo `Nombre` que como hemos visto en la declaración es de tipo String. Téngase en cuenta que DatosCliente es de tipo TDatosCliente pero que los campos de éste tienen su propio tipo, tal como se ha declarado en el registro.

Podemos indicar al compilador que los identificadores que no conozca los busque en el record mediante la palabra reservada `with`. Podíamos haber escrito el código así :

```
var
  DatosCliente : TDatosCliente;
begin
  Writeln('Introduzca los datos del cliente');
  with DatosCliente do
  begin
    Write('Nombre : '); Readln(Nombre); // Esto sólo puede ser DatosCliente.Nombre
    Write('Apellidos : '); Readln(Apellidos); // DatosCliente.Apellidos
    Write('Núm. Teléfono : '); Readln(NumTelefono); // DatosCliente.NumTelefono
  end;
end.
```

A menos que sea necesario, no se recomienda el uso de with ya que confunde al programador. La utilización del punto permite cualificar el identificador y evita confusiones. También podemos cualificar el identificador en función de su unit. Por ejemplo la función ReadKey de la unit Crt se puede indicar como `Crt.ReadKey`. En este caso, el punto es útil para determinar exactamente qué variable o qué función estamos llamando, especialmente cuando haya más de una en diferentes units.

6.6.1.Registros variantes

Pascal permite definir lo que se llaman registros con partes variantes. Un registro con una parte variante consta de un campo que llamaremos selector. El valor de este selector es útil para el programador pues le permite saber cual de los campos posibles se están empleando. El compilador no tiene en cuenta el valor de este campo selector ya que permite el acceso a todos los campos declarados en la parte variante.

```
type
  TConjuntoDatos = ( TCaracter, TCadena, TNumero, TCadenayCaracter );
  TCombinado = record
  case ConjuntoDatos : TConjuntoDatos of
    TCaracter : ( Caracter : Char);
    TCadena : ( Cadena : string );
    TNumero : ( Numero : integer);
  end;

var
  Combinado : TCombinado;
```

Por ejemplo podemos realizar lo siguiente `Combinado.Caracter := 'z';` y después `Combinado.ConjuntoDatos := TCaracter;` para indicar qué campo hay que leer. Por ejemplo, si hacemos la asignación `Combinado.Numero := 65;` y después hacemos `Writeln(Combinado.Caracter);` se escribirá en la pantalla la letra `A`.

Este comportamiento, en apariencia tan raro, es debido a que un registro con la parte variante sitúa los distintos campos en una misma región de memoria de forma que podamos trabajar con un Byte y un Boolean o trabajar con un Cardinal y leer sus dos Word, el superior y el inferior.

El uso de registros variantes está restringido a usos muy concretos. Esta es la forma Pascal de implementar un `union` de C/C++.

7.Archivos

7.1.La necesidad de los archivos

Los programas que hemos desarrollado hasta ahora se caracterizan por poseer una memoria volátil que se concretaba con las variables. Al finalizar el programa, el valor de las variables, sea útil o no, se pierde sin que haya forma de recuperarlo en una posterior ejecución del mismo programa. Los archivos permiten resolver este problema ya que permiten el almacenaje de datos en soportes no volátiles, como son los disquetes, los discos duros y su posterior acceso y modificación.

7.2.Cómo trabajar con archivos en Pascal

Todos los sistemas de archivos permiten asignar un nombre a los archivos de forma que lo podamos identificar de forma única dentro de un directorio, por ejemplo.

En Pascal, en vez de operar directamente con los nombres de archivos trabajaremos con un **alias** que no es nada más que una variable que ha sido asignada a un nombre de archivo. Las posteriores operaciones que queramos llevar a cabo sobre este archivo tomarán como parámetro este alias y todas las operacions se realizarán al archivo al cual se refiere este alias.

7.2.1.Tipos de archivos en Pascal

Los tres tipos de archivos con los cuales podemos trabajar en Pascal son : archivos con tipo, archivo sin tipo y archivos de texto.

Los archivos con tipo almacenan un único tipo de dato y permiten los que se llama **acceso aleatorio**, en contraposición con el **acceso secuencial** que obliga a leer (o al menos pasar) por todos los datos anteriores a uno concreto. De esta forma podemos situarnos en cualquier parte del archivo de forma rápida y leer o escribir datos.

Los archivos sin tipo no almacenan, a la contra, ningún tipo en concreto. El programador es quien decide qué datos se leen y en qué orden se leen. El acceso a datos de este tipo de archivos es totalmente secuencial aunque es más versátil pues se permiten almacenar datos de casi cualquier tipo.

Los archivos de texto permiten la lectura y escritura de archivos ASCII y la conversión automática a cadenas String[255], o ShortString, que en realidad no son cadenas ASCII. El acceso a estos archivos también es secuencial.

7.3.Trabajar con archivos de texto

Empezaremos con archivos de texto pues son más simples que otros tipos de archivo.

El tipo de archivo de texto se define con el tipo `Text`. Por tanto declarar un alias de archivo de texto es tan simple como la declaración siguiente :

```
var
  ArchivoTexto : Text;
```

Para asignar a un alias un nombre de archivo emplearemos la función Assign[2]

```
Assign(ArchivoTexto, 'PRUEBA.TXT');
```

En este ejemplo hemos asignado la variable ArchivoTexto al archivo PRUEBA.TXT. Hay que tener en cuenta de que en Windows los archivos no son sensibles a las mayúsculas, podíamos haber puesto prueba.txt y sería el mismo archivo. Téngase en cuenta que en algunos sistemas operativos (como Linux) esto puede no ser cierto.

Ahora habrá que abrir el archivo para realizar alguna operación. Básicamente distinguimos entre dos tipos de operaciones en un archivo : operaciones de lectura, leemos el contenido, y operaciones de escritura, donde escribimos el contenido.

Pascal, además permite abrir el archivos con diferentes finalidades y con diferentes funciones. Si lo queremos abrir de sólo lectura, sin poder realizar escritura, emplearemos la función Reset. Si lo que queremos es rescribir de nuevo el archivo, sin posibilidad de leer, emplearemos Rewrite. Además, los archivos de texto permiten añadir datos al final del archivo si este se abre con la orden Append, la cual es exclusiva de los archivos de texto. Las tres funciones toman como único parámetro el alias del archivo.

```
Reset(ArchivoTexto); // Sólo podemos leer
// o bien
Rewrite(ArchivoTexto); // Borramos el archivo y sólo podemos escribir
// o bien
Append(ArchivoTexto); // Sólo podemos escribir al final del archivo
```

La diferencia fundamental entre Rewrite y Append se encentra en que Rewrite borra todos los datos del archivo si ya existiese, de lo contrario lo crea, mientras que Append crea el archivo si no lo existe, de lo contrario se sitúa al final del archivo. Reset sólo funciona con archivos que ya existen, en caso contrario genera un error.

El llamado cursor de lectura/escritura se refiere al punto desde el cual empieza nuestra lectura o escritura. Al abrir con Reset o Rewrite el cursor siempre se encuentra al principio del archivo. En el caso de Append el cursor se encuentra en el final del archivo de texto de forma que la única cosa que

[2]La función AssignFile hace lo mismo que Assign.

podemos hacer es escribir del final del fichero en adelante, o sea alargándolo. Al leer o escribir en un archivo el cursor de lectura/escritura adelanta hacia el fin del fichero tantas posiciones como el tamaño del dato escrito o leído. En caso de lectura, leer más allá del fin del archivo es incorrecto mientras que escribir a partir del fin del fichero hace que éste crezca.

Los únicos datos que podemos escribir en un archivo de texto son efectivamente cadenas de texto. La estructura de los archivos de texto ASCII es bien simple. Constan de líneas formadas por una tira de caracteres que terminan con los caracteres ASCII número 13 y 10 (fin de línea y retorno de carro).

Para leer o escribir una línea desde un archivo de texto emplearemos las funciones Readln o Writeln respectivamente, pero teniendo en cuenta de que la primera variable especificada como parámetro sea el alias del archivo y el segundo parámetro sea un String. Sólo en el caso de Writeln, el segundo parámetro puede ser un literal de String mientras que Readln exige una variable String como segundo parámetro.

Una vez hemos finalizado el trabajo con el archivo conviene cerrarlo para asegurarse de que los datos escritos se escriben correctamente y para indicar que ya no lo vamos a emplear más. La función Close[3], que toma como parámetro el alias del archivo, se encarga de esta tarea.

Por ejemplo este programa escribe la fecha actual a un archivo de texto :

```
program EscribirFecha;
uses Dos, SysUtils;
{ La unit DOS se necesita para GetDate
  La unit SysUtils es necesaria para IntToStr }
const
  DiasSemana : array[0..6] of string =
                              ('Domingo', 'Lunes', 'Martes',
                              'Miercoles', 'Jueves', 'Viernes', 'Sábado');
var
  ArchivoTexto : Text;
  Anyo, Mes, Dia, DiaSemana : Integer;
begin
  Assign(ArchivoTexto, 'FECHA.TXT');
  Rewrite(ArchivoTexto); // Lo abrimos para escritura borrandolo todo.
  GetDate(Anyo, Mes, Dia, DiaSemana);
  Writeln(ArchivoTexto, DiasSemana[DiaSemana]+' '+IntToStr(Dia)+
              '/'+IntToStr(Mes)+'/'+IntToStr(Anyo));
  Close(ArchivoTexto);
end.
```

Antes de seguir hay que hacer un comentario sobre el array DiasSemana que hemos empleado. Es posible declarar constantes con tipo de forma que se especifica de qué tipo son y su valor inicial, ya que las constantes con tipos pueden ser modificadas a lo largo del programa, aunque no suele ser muy

[3]En vez de Close se puede emplear CloseFile.

usual. En el caso de los arrays la sintaxis obliga a separar los literales entre paréntesis. En caso de de arrays multidimensionales hay que emplear la misma sintaxis y separarla entre comas :

```
const
  ArrayBi : array [0..1, 0..1] of integer = ((-3, 4), (2, 7));
```

El array que hemos declarado lo empleamos para saber el nombre del día de la semana. Los valores del día actual se obtienen gracias a la función GetDate a la cual le pasaremos cuatro variables : día, mes año y una para el día de la semana. Mediante el array DiasSemana podemos obtener el nombre de los días de la semana ya que GetDate devuelve un valor entre 0 y 6 dónde cero es domingo y el seis es sábado. De esta forma el día de la semana devuelto por GetDate coincide con los nombres del array.

En el Writeln posterior hemos concatenado el día de la semana, el día, el mes y el año, aunque por ser Writeln podríamos haber separado estos elementos por comas como si fueran más parámetros.

Vamos a leer el archivo que hemos creado con otro programa que lea una línea del archivo y la imprima por la pantalla.

```
program LeerFecha;
var
  ArchivoTexto : Text;
  CadenaFecha : string;
begin
  Assign(ArchivoTexto, 'FECHA.TXT');
  Reset(ArchivoTexto); // Lo abrimos para sólo lectura
  Readln(ArchivoTexto, CadenaFecha);
  Writeln(ArchivoTexto);
  Close(ArchivoTexto);
end.
```

Leemos una sola línea de el archivo de texto y la almacenamos en la variable CadenaFecha. Después la mostramos en pantalla para comprobar que el dato leído es el que realmente había en el archivo. Finalmente cerramos el archivo.

El uso de Read en vez de Readln no es recomendable ya que puede tener un comportamiento extraño. En cambio, si lo que queremos es añadir una cadena sin saltar de línea podemos emplear Write con la misma sintaxis Writeln.

Hay otras funciones muy interesantes y útiles para los archivos de texto, todas toman como parámetro el alias del archivo de texto. Eof (End of file) devuelve true si el cursor del archivo ha llegado al final de éste. Eoln (End of line) devuelve true si el cursor se encuentra al final de una línea. Esta última función se suele emplear cuando se lee con Read carácter a carácter. La función Eof permite parar la lectura del archivo ya que leer más allá del archivo genera un error de ejecución.

La función Flush descarga los buffers internos del archivo de texto al disco de forma que el buffer, o memoria intermedia, se vacía y se actualiza el archivo.

Las funciones SeekEof y SeekEoln son parecidas a las respectivas Eof y Eoln pero sin tener en cuenta los caracteres en blanco, de forma que si desde la posición del cursor del archivo hasta el final del archivo o final de línea todo son blancos SeekEof y SeekEoln, respectivamente, devuelven true.

Truncate suprime todos los datos posteriores a la posición actual del cursor del archivo.

7.4.Trabajar con archivos con tipo

Para declarar que un archivo está formado por datos de un sólo tipo hay que declararlo con las palabras reservadas `file of` y el tipo del archivo :

```
type
  TAgenda = record
    Nombre, Apellidos : string;
    NumTelefono : Integer;
  end;
var
  Agenda : file of TAgenda;
```

De esta forma podríamos implementar una agenda muy rudimentaria con sólo tres campos (Nombre, Apellidos y NumTelefono). Para crear, escribir y leer del archivo el funcionamiento es idéntico a los archivos de texto con la diferencia que hay que leer los datos en una variable del tipo del archivo. También hay pequeñas diferencias respecto a la apertura del archivo. Reset permite la lectura y la escritura, en archivos de texto sólo permitía lectura. Append no se puede emplear en archivos que no sean de texto y Rewrite se comporta igual que con los archivos de texto.

Vamos a ver un ejemplo de como podemos añadir datos al archivo de agenda :

```
program EscribirArchivoTipo;
uses SysUtils; // Para la función FileExists
const
  NombreArchivo = 'AGENDA.DAT';
type
  TAgenda = record
    Nombre, Apellidos : string;
    NumTelefono : Cardinal;
  end;
var
  Agenda : file of TAgenda;
  DatoTemporal : TAgenda;
  NumVeces, i : Integer;
begin
  Assign(Agenda, NombreArchivo); // Asignamos el archivo
  if not FileExists(NombreArchivo) then
```

```
  begin
    Rewrite(Agenda); // Si no existe lo creamos
  end
  else
  begin
    Reset(Agenda); // Lo abrimos para lectura escritura
  end;
  Write('Cuantas entradas desea añadir en la agenda ? ');
  Readln(NumVeces);

  for i := 1 to NumVeces do
  begin
    Write('Introduzca el nombre : '); Readln(DatoTemporal.Nombre);
    Write('Introduzca los apellidos : '); Readln(DatoTemporal.Apellidos);
    Write('Introduzca el teléfono : '); Readln(DadaTemporal.NumTelefono);
    Write(Agenda, DatoTemporal); // Escribimos DatoTemporal al archivo
    Writeln; // Equivale a Writeln(); hace un salto de línea
  end;
  Close(Agenda); // Cerramos la agenda
end.
```

Observamos en el código de que cuando el archivo no existe entonces lo creamos automáticamente con la función Rewrite. En caso contrario lo podemos abrir con Reset. La función FileExists permite saber si el archivo especificado existe ya que devuelve true en caso afirmativo.

Pedimos al usuario que nos diga cuantas fichas desea introducir y solicitamos que introduzca los datos. Después almacenamos la variable DatoTemporal al archivo agenda con una llamada a Write. No es posible emplear Writeln en archivos con tipo. Cuando llamamos a Writeln sin parámetros en pantalla se ve un salto de línea. Lo empleamos para distinguir los diferentes registros que vamos introduciendo al archivo.

Un archivo con tipo se entiende que está formado por un numero determinado de datos del tipo del archivo, en nuestro caso son registros de tipo TAgenda.

Para saber el número de registros que tiene nuestro archivo emplearemos la función FileSize como parámetro el alias del archivo. El resultado que nos devuelve esta función es el nombre de registros del archivo. Emplearemos esta función en el ejemplo siguiente de lectura del archivo.

```
program LeerArchivoTipo;
uses SysUtils; // Para la función FileExists
const
  NombreArchivo = 'AGENDA.DAT';
type
  TAgenda = record
    Nombre, Apellidos : Shortstring;
    NumTelefono : Cardinal;
  end;
var
  Agenda : file of TAgenda;
```

```
  DatoTemporal : TAgenda;
  i : Integer;

begin
  Assign(Agenda, NombreArchivo); // Asignamos el archivo
  if not FileExists(NombreArchivo) then
  begin
    Writeln('El archivo ', NombreArchivo, ' no existe.');
  end
  else
  begin
    Reset(Agenda); // Lo abrimos para lectura y escritura

    for i := 1 to FileSize(Agenda) do
    begin
      Read(Agenda, DatoTemporal);
      Writeln('Nom : ', DatoTemporal.Nombre);
      Writeln('Cognoms : ', DatoTemporal.Apellidos);
      Writeln('Telèfon : ', DatoTemporal.NumTelefono);
      Writeln;
    end;
    Close(Agenda);
  end;
end.
```

En este caso, vamos leyendo los registros guardando los valores a la variable temporal `DatoTemporal`. Después escribimos esta información en la pantalla. Siempre que el archivo exista, claro.

Tal como hemos comentado antes, los archivos con tipo son archivos de acceso aleatorio y podemos escribir y leer datos en el orden que queramos. Para poder acceder a una posición determinada del archivo hay que emplear la función Seek.

Esta función necesita dos parámetros : el primero es el alias del archivo y el segundo el número del elemento al cual queremos acceder. Hay que tener en cuenta que el primer elemento es el cero y el último es el valor de FileSize menos 1.

Por ejemplo, para leer el archivo en sentido inverso al habitual, puesto que Read adelanta el cursor de lectura automáticamente que vamos leyendo, necesitaremos la función Seek.

```
program LeerAlReves;
// Aquí van las declaraciones del ejemplo anterior : TAgenda, Agenda, ...
begin
  Assign(Agenda, NombreArchivo); // Asignamos el archivo
  if not FileExists(NombreArchivo) then
  begin
    Writeln('El archivo', NombreArchivo, ' no existe.');
  end
  else
```

```
  begin
    Reset(Agenda); // Lo abrimos para lectura y escritura

    for i := FileSize(Agenda)-1 downto 0 do
    begin
      Seek(Agenda, i); // Nos movemos al punto especificado por i
      Read(Agenda, DatoTemporal);
      Writeln('Nombre : ', DatoTemporal.Nombre);
      Writeln('Apellidos : ', DatoTemporal.Apellidos);
      Writeln('Telefono : ', DatoTemporal.NumTelefono);
      Writeln;
    end;
    Close(Agenda);
  end;
end.
```

Si en algún momento queremos saber la posición del cursor del archivo tenemos que emplear la función FilePos y el alias del archivo. Esta función devuelve un entero que, a diferencia de Seek, puede estar comprendido entre 1 y el valor de FileSize.

7.5.Archivos sin tipo

Los archivos sin tipo no tienen ningún formato especial y por tanto podemos leer de ellos los datos que queramos. Hay que tener en cuenta, que en ninguno de los tres tipos de archivos se hace ningún tipo de comprobación de los datos que se escriben o se leen. De forma que si abrimos un archivo de un tipo determinado, por ejemplo de tipo Longint, con un alias de tipo Byte los datos que leamos corresponderán a los bytes del Longint.

Las funciones Write y Read no se pueden emplear con archivos sin tipo. Es por este motivo que tenemos que emplear las funciones BlockWrite y BlockRead. Estas funciones leen o escriben un numero n, o menos en caso de lectura, de bloques de tamaño b, en bytes, que conviene especificar en las funciones Rewrite y Reset. Si no especificamos el valor del bloque a Rewrite o Reset por defecto vale 128 bytes.

Como ejemplo, implementaremos dos programas. En el primero escribiremos en un archivo un string de 35 caracteres y después un entero. En el segundo nos limitaremos a recoger estos datos y a escribirlos en pantalla.

```
program EscribirArchivoSinTipo;
const
  NombreArchivo = 'DATOS.DAT';
var
  Datos : file;
  Nombre : string[35];
  Edad : Integer;
begin
```

```
  Assign(Datos, NombreArchivo); // Asignamos el archivo
  Rewrite(Datos, 1); { Especificamos 1 byte como tamaño
                       del bloque por comodidad }
  Write('Introduzca un nombre : '); Readln(Nombre);
  Write('Introduzca una edad : '); Readln(Edad);
  BlockWrite(Datos, Nombre, SizeOf(Nombre));
  BlockWrite(Datos, Edad, SizeOf(Edad));
  Close(Datos);
end.
```

Obsérvese, que especificamos el tamaño del registro en 1 ya que después necesitaremos especificar el tamaño de los datos que vamos a escribir. Para esto emplearemos la función SizeOf que toma como parámetro cualquier tipo de variable y devuelve su tamaño en bytes. Por tanto `Edad` tiene un tamaño de 2 bytes y el tamaño de `Nombre` es de 36 bytes. Este es el valor que se pasa como tercer parámetro y que entiende que queremos copiar 36 bloques de un byte y después 2 bloques de un byte. Si no hubieramos especificado el tamaño del bloque, se tomaría por defecto 128 bytes de forma que al escribir el entero coparíamos 2 bloques de 128 bytes (o sea 256 bytes). La función BlockWrite admite cualquier tipo de variable como segundo parámetro.

Para leer, la función BlockRead funciona de forma parecida a BlockWrite. En este caso el segundo parámetro también puede ser de cualquier tipo, pero no una constante.

```
program LeerArchivoSinTipo;
const
  NombreArchivo = 'DATOS.DAT';
var
  Datos : file;
  Nombre : string[35];
  Edad : Integer;

begin
  Assign(Datos, NombreArchivo); // Asignamos el archivo
  Reset(Datos, 1); { Especificamos el tamaño del bloque de 1
                     byte por comodidad }
  BlockRead(Datos, Nombre, SizeOf(Nombre));
  BlockRead(Datos, Edad, SizeOf(Edad));
  Writeln(Nombre, ' tiene ', Edad, ' años');
  Close(Datos);
end.
```

Otras funciones que podemos emplear para los archivos sin tipo incluyen : FileSize, FilePos y Seek. Todas tres funciones trabajan con el valor del bloque, el tamaño del cual hemos especificado en Reset o Rewrite. También podemos emplear Eof para detectar el fin del archivo.

7.6.Gestión de archivos

A diferencia de crear archivos, escribir o leer datos también podemos realizar otras operaciones como pueden ser eliminar un archivo o cambiarle el nombre.

7.6.1.Eliminar un archivo

Eliminar un archivo es tan simple como emplear la función Erase y el alias del archivo. Hay que tener en cuenta de que el archivo tiene que estar cerrado para evitar un error en tiempo de ejecución. Además, cerrar un archivo ya cerrado da error de ejecución. Para asegurarse de que un archivo está cerrado podemos abrirlo con Reset y cerrarlo con Close. Nuevas llamadas a Reset con un archivo ya abierto no dan error, simplemente sitúan el cursor del archivo al inicio. Una vez está cerrado ya podemos llamar a Erase.

7.6.2.Renombrar un archivo

La función Rename permite renombrar un archivo. Hay que tener en cuenta de que el archivo tambien tiene que estar cerrado.

```
var
  archivo : file;
begin
  Assign(archivo, 'NombreViejo.dat'); // Suponemos que existe
  Rename(archivo, 'NombreNuevo.dat');
end.
```

7.6.3.Capturar errores de archivos con IOResult

Hasta ahora hemos operado con archivos suponiendo que el archivo existía o bien no había ningún error durante las operaciones de lectura o de escritura.

Existe una forma muy sencilla de capturar los errores de las operaciones con archivos mediante la variable IOResult y la directiva de compilador {$I}

Una directiva de compilador es un comentario que justo después de la llave hay el símbolo de dólar $ y una o más letras. Las directivas modifican el comportamiento del compilador en tiempo de compilación. Después de una directiva el compilador puede modificar su comportamiento de muchas formas. Muchas veces van asociadas a la llamada compilación condicional, que veremos más adelante, en otras se refiere a como se estructuran los datos, etc.

En el caso de $I es una directiva de activación/desactivación. Por defecto $I está activada con lo cual los errores de entrada/salida de archivos provocan un error de ejecución (runtime error). Los errores runtime son irrecuperables, o sea, provocan la finalización inmediata y anormal del programa. Muy a menudo, el error puede ser debido al usuario con lo cual hay que procurar de no genera un error

runtime ya que inmediatamente terminaría el programa. Podemos evitar estos errores con la desactivación de la directiva $I. Para activar una directiva de activación/desactivación (llamada *switch*) tenemos que incluir la siguiente directiva enmedio del código.

```
{$I+}
```

Para desactivarla :

```
{$I-}
```

Es importante reactivar la directiva $I cuando ya no sea necesaria. Cuando está desactivada podemos acceder a la variable IOResult. Esta variable tiene el valor cero si la operación con archivos se ha llevado a cabo con éxito. En caso contrario, almacenará un valor distinto de cero. Vamos a aplicar esta técnica al programa de cambio de nombre :

```
var
  archivo : file;
begin
  Assign(archivo, 'NombreViejo.dat'); // Assign no da nunca error
  {$I-} // Desactivamos la directiva $I
  Rename(archivo, 'NombreNuevo.dat');
  if IOResult <> 0 then
  begin
       Writeln('Este archivo no existe o no se ha podido renombrar');
  end;
  {$I+} // Importante reactivar la directiva cuando ya no sea necesaria
end.
```

Así de simple es capturar los errores de lectura/escritura. Suele ser habitual, por ejemplo, emplearlo en la función Reset ya que si el archivo no existe se produce un error que se notifica en IOResult.

Es posible referirse a la directiva $I con su nombre largo $IOCHECKS. Además FreePascal permite la activación y desactivación mediante ON y OFF en los nombres largos de directivas.

```
{$I+} // Activar
{$I-} // Desactivar
{$IOCHECKS ON} // Activar, cuidado el espacio
{$IOCHECKS OFF} // Desactivar
{$IOCHECKS +} // Activar, cuidado el espacio
{$IOCHECKS -} // Desactivar
```

Hay más directivas de compilador que veremos más adelante.

8.Funciones y procedimientos

8.1.Diseño descendiente

Se llama diseño descendiente el método de diseño de algoritmos que se basa en la filosofía de "divide y vencerás". Donde el programa se divide en partes más pequeñas que se van desarrollando con una cierta independencia respecto a las otras partes del programa.

De esta forma podemos definir una especie de subprogramas o subrutinas para ser ejecutada diversas veces dentro del programa en si. Los problemas pueden ser descompuestos en otros problemas más pequeños pero también más fáciles de resolver. Esto permite, también, aprovechar mejor el código y permite modificarlo más cómodamente pues no habrá que modificar todo el programa, sólo el código del subprograma.

En Pascal el diseño descendiente se lleva a cabo con dos elementos muy importantes : las funciones y los procedimientos.

8.1.1.Funciones vs Procedimientos

La diferencia más importante entre un procedimiento y una función es el hecho de que una función es una expresión y como a tal se puede emplear dentro de otras expresiones. Un procedimiento, en cambio, representa una instrucción y no puede ser empleada dentro de un contexto de expresión.

Las funciones devuelven siempre un valor de un tipo determinado. Ya hemos visto el caso de la función FileExists de la unit SysUtils. Esta función devuelve un tipo booleano por lo que podíamos incluirla dentro de una expresión booleana como esta :

```
if not FileExists(NombreArchivo) then
...
```

La función Write es, por ejemplo, un procedimiento y por tanto no es sintácticamente legal incluirla dentro de una expresión.

Aun así, la sintaxis de Pascal se ha relajado a lo largo del tiempo y es posible emplear una función como si de un procedimiento se tratara. En este caso se pierde el valor que devuelve la función aunque se ejecuta igualmente.

Por ejemplo, si queremos que un programa se pare hasta que el usuario pulse una tecla podemos emplear la función Readkey de la forma siguiente :

```
Readkey; // El programa se parará hasta que el usuario pulse una tecla
```

El carácter que devuelve ReadKey, la tecla que ha pulsado el usuario, se perderá a menos que incluyamos ReadKey en una expresión.

8.1.2.Contexto de las funciones y los procedimientos

Las funciones y los procedimientos tienen que declararse siempre antes del módulo principal del programa. Si una función, o procedimiento, se emplea en otra ésta tendrá que estar declarada antes o bien encontrarse en una unit de la clausula uses.

8.2.Funciones

Tanto las funciones como los procedimientos tienen que tener un nombre que las identifique, que seguirá las reglas de los identificadores de Pascal, y opcionalmente pueden tener un conjunto de parámetros que modifique de alguna forma el comportamiento de la función.

La declaración de una función sigue el modelo siguiente :

```
function NombreFuncion ( listaParametros ) : tipoRetorno;
```

NombreFuncion es el identificador de la función y es el nombre que emplearemos para llamarla. ListaParametros representa el conjunto de parámetros que se pueden pasar a la función. Para declarar más de un parámetro los tenemos que separar entre puntos y comas y teniendo en cuenta de que el último parámetro no lleva nunca punto y coma. Estos parámetros reciben el nombre de **parámetros formales** y es importante distinguirlos de **parámetros reales** que representan los parámetros con los que se ha llamado la función. Finalmente, tipoRetorno representa el tipo de datos que devuelve la función.

Vamos a definir una función muy sencilla max que devuelva el valor del máximo de dos parámetros enteros a y b.

```
function max(a : integer; b : integer) : integer;
{ Podíamos haber escrito los parámetros así : (a, b : integer) }
begin
  if a >= b then
  begin
    max := a
  end
  else
  begin
    max := b;
  end;
end;
```

Obsérvese con detenimiento el código de la función max. Para empezar, las sentencias van entre un begin y un end con punto y coma ya que no es el bloque principal del programa. Nótese también que para especificar el valor que tiene que devolver la función hacemos una asignación al identificador de la función, max.

Como ejemplo de empleo de esta función, haremos un programa que pida dos enteros al usuario e indique cual de los dos es máximo.

```
program Funciones;
var
  n, m : integer;

function max(a, b : integer) : integer;
begin
  if a >= b then
  begin
    max := a
  end
  else
  begin
    max := b;
  end;
end;

begin
  Write('Introduzca el valor entero M : '); readln(m);
  Write('Introduzca el valor entero N : '); readln(n);
  Writeln('El máximo entre ', M, ' y ', N, ' es ', max(m, n));
end.
```

Como se puede ver en la última línea del código, a diferencia de la declaración y tal como hemos llamado hasta ahora las funciones o procedimientos con más de un parámetro, hay que separar los distintos parámetros entre comas. Hay que tener en cuenta que los parámetros reales m y n del ejemplo no tienen nada que ver con los parámetros formales de max (a y b). De hecho los parámetros formales sirven sólo para implementar la función e indicar el orden de estos. No hay que decir que en vez de una variable podemos emplear un literal.

Vamos a definir una nueva función min a partir de max.

```
function min(a, b : integer);
begin
  min := -max(-a, -b);
end;
```

Esta implementación se basa en que los números negativos se ordenan al revés que los positivos, por lo que cambiando dos veces el signo es suficiente.

Es importante siempre incluir una asignación al valor de retorno de la función ya que en caso contrario el valor de la función podría quedar indeterminado. Una forma segura es incluir la asignación al final de la función, si es posible, o asegurarse de que en todos los casos posibles de salida hay al menos una asignación de este tipo.

8.3.Procedimientos

Los procedimientos se declaran de forma parecida a las funciones pero sin especificar el tipo de salida ya que no tienen. Por ejemplo el procedimiento Creditos escribiría unos créditos en la pantalla :

```
procedure Creditos;
begin
  Writeln('(C) Manual de FreePascal 1.00'
  Writeln('Roger Ferrer Ibáñez - 2001');
end;
```

En vez de emplear la palabra reservada function emplearemos `procedure`. Tampoco hay que realizar ninguna asignación al identificador del procedimiento pues los procedimientos no devuelven nada. Tanto en funciones como en procedimientos, en caso de que no tengamos parámetros no habrá que poner los paréntesis. En el caso de las funciones antes del punto y coma tendrá que ir el tipo de retorno. Como ejemplo tenemos la definición de ReadKey es :

```
function Readkey : Char;
```

En el momento de llamar a una función o procedimiento sin parámetros podemos omitir los paréntesis de parámetros o, si los queremos escribir, sin nada enmedio (sin contar los caracteres en blanco).

```
Creditos; // Ambas llamadas son correctas
Creditos(); // Esta llamada tiene una sintaxis más parecida a C
```

8.4.Tipos de parámetros

Hasta ahora hemos visto que una función puede devolver un valor y que por tanto la podemos incluir dentro de una expresión. Muchas veces nos puede interesar especificar parámetros que impliquen un cambio en estos parámetros y esto no es posible de implementar en una simple función.

Supongamos un procedimientos en el cual pasando dos parámetros de tipo entero queremos que se intercambien los valores. Una forma de hacerlo es mediante sumas y restas. Nosotros implementaremos un caso más general mediante el uso de una tercera variable temporal.

```
procedure Intercambiar(a, b : integer);
var
  temporal : integer;
begin
  temporal := b;
  b := a;
  a := temporal;
end;
```

Si escribimos un programa para veri si el procedimiento funciona, nos daremos cuenta de que los parámetros no se han modificado después de la llamada. O sea, no ha habido intercambio.

```
a := 10;
b := 24;
Intercambiar(a, b);
// Ahora a aún vale 10 y b aún vale 24 !!!
```

Esto es porque hemos empleado un tipo de parámetro llamado **parámetro por valor**. Los parámetros por valor no son nada más que una copia de los parámetros originales que se crean sólo para la función o procedimiento. Una vez termina esta copia desaparece, por lo que toda modificación que hagamos dentro de la función o procedimiento se perderá.

Para resolver este problema podemos emplear **parámetros por referencia**. En este caso la función o procedimiento conoce exactamente la posición de memoria del parámetro por lo que las modificaciones que hagamos dentro de la función o procedimiento permanecerán después de la llamada. Para indicar que un parámetro es por referencia incluiremos la palabra reservada var delante del parámetro. El procedimiento intercambiar quedaría así :

```
procedure Intercambiar(var a : integer; var b : integer);
{ Podíamos haber escrito como parámetros (var a, b : integer); }
var
  temporal : integer;
begin
  temporal := b;
  b := a;
  a := temporal;
end;
```

Una de las ventajas de los parámetros por referencia es que no es necesario crear una copia cada vez que se llama la función o procedimiento, con lo cual ahorramos algo de tiempo sobretodo si el dato es grande (como arrays muy extensos o registros con muchos campos). El riesgo, pero, es que cualquier modificación (por accidental que sea) permanecerá después de la llamada. Para minimizar este riesgo disponemos de un tercer tipo de parámetro llamado parámetro constante. En vez de var llevan la palabra

reservada **const**. El compilador no permitirá ningún tipo de modificación fortuita de los datos pero los parámetros se pasaran por referencia y por tanto la función o procedimiento será más eficiente.

En el caso anterior intentar compilar procedure Intercambiar(const a, b : integer); daría un error de compilación pues entendería a y b como constantes.

Es importante remarcar que en caso de pasar variables por referencia no es posible especificar literales o constantes en este tipo de parámetros ya que si sufrieran alguna modificación dentro de la función o procedimiento no se podría almacenar en ningún sitio. Además, el compilador es más estricto a la hora de evaluar la compactibilidad del tipo. Por ejemplo si el parámetro por referencia es de tipo Integer y pasamos una variable Byte el compilador lo considerará ilegal aunque ambos sean enteros y, en general, un Byte es compatible con un Integer.

En el caso de parámetros constantes, el compilador no es tan estricto y permite pasar literales y constantes.

8.5.Variables locales i globales

En el ejemplo anterior de intercambiar hemos definido una variable temporal de tipo entero dentro del cuerpo del procedimiento. Estas variables declaradas dentro de una función o procedimiento reciben el nombre de variables locales y sólo son accesibles dentro de las funciones o procedimientos. Por lo que, intentar asignar un valor a temporal fuera de intercambiar es sintácticamente incorrecto ya que no es visible en este ámbito.

Las variables declaradas al principio del programa reciben el nombre de variables globales. Estas variables se pueden emplear tanto en funciones y procedimientos como en el bloque principal.

También el bloque principal puede tener sus propias variables locales si las declaramos justo antes de éste.

Esta consideración para variables es extensiva a tipos y constantes.

```
program Variables;
var
  a : integer;

procedure Asignar;
var
  b, c : integer;
begin
  a := 15; // CORRECTO a es una variable global
  b := 10; // CORRECTO b es una variable local
  c := b; // CORRECTO c es una variable local
  d := b; { INCORRECTO d es una variable del bloque principal que
              no se puede acceder desde aquí }
end;
```

```
var
  b, d : integer;
begin
  a := 10; // CORRECTO a es una variable global
  b := 5; // CORRECTO b es una variable local del bloque principal
  Asignar; // ATENCIÓN!! b no cambiará su valor después de la llamada (b=5)
  c := 4; { INCORRECTO!! c no és accesible desde aquí ya que es
            una variable local de Asignar }
end.
```

A banda de los diferentes errores sintácticos hay que remarcar la diferencia entre b del procedimiento Asignar y b del bloque principal. Ambas variables son totalmente independientes entre ellas. Este es el único caso en que es posible la existencia de dos variables con nombres iguales. Si la variable b hubiera sido declarada global, juntamente con a, el compilador no nos hubiera permitido declarar b en ningún otro punto pues habría una redeclaración de identificador y esto no es posible pues el compilador no sabría como distinguir las variables. Veremos más adelante, que en diferentes units se pueden declarar los mismos identificadores pues se puede especificar que identificador queremos.

Es posible emplear los mismos nombres en ámbitos diferentes pero no es recomendable a menos que estemos muy seguros de que no nos vamos a confundir.

8.7.Sobrecarga de procedimientos y funciones

Es posible definir más de una función o procedimiento con el mismo identificador pero con parámetros distintos. Esta característica permite ahorrar trabajo al programador ya que con un mismo nombre puede englobar más de una función que realice operaciones parecidas, por ejemplo.

Para sobrecargar una función o procedimiento sólo hay que redeclararlo y tener en cuenta de que tienen que cambiar los parámetros. Si no cambian los tipos de los parámetros el compilador no sabrá distinguir las dos funciones y presentará un error de compilación.

Vamos a implementar tres funciones sobrecargadas que compararan pares de datos de tipo string, enteros y booleanos. Las funciones devolverán cero cuando los dos elementos sean iguales, 1 cuando el primero sea mayor y -1 cuando el segundo sea mayor que el primero.

```
function Comparar(a, b : integer) : integer;
begin
       if a > b then Comparar := 1;
       if a < b then Comparar := -1;
       if a = b then Comparar := 0;
end;

function Comparar(a, b : string) : integer;
begin
       if a > b then Comparar := 1;
       if a < b then Comparar := -1;
```

```
        if a = b then Comparar := 0;
end;

function Comparar(a, b : boolean) : integer;
begin
        if a > b then Comparar := 1;
        if a < b then Comparar := -1;
        if a = b then Comparar := 0;
end;
```

En este caso el código de las tres funciones coincide pero no tiene porque ser así en otros casos. Podemos llamar a las funciones especificando dos enteros, dos strings o dos booleanos pero no ninguna otra combinación que no hayamos especificado. Incluso, podemos sobrecargar métodos con distinto número de parámetros. Por ejemplo :

```
function Comparar(a : integer) : integer;
begin
  Comparar := Comparar(a, 0);
end;
```

En este caso si no especificamos el segundo entero entonces la comparación se hará con cero. Suele ser habitual definir métodos sobrecargados con menos parámetros cuando se trata de casos más habituales o con parámetros por defecto.

La flexibilidad de la sobrecarga en FreePascal permite incluso a admitir la mezcla de funciones y procedimientos. Podemos definir otro procedimiento `Comparar` en el cual el resultado se guarde en el parámetro c.

```
procedure Comparar(a, b : integer; var c : integer);
begin
  if a > b then c := 1;
  if a < b then c := -1;
  if a = b then c := 0;
end;
```

En este caso Comparar no devuelve nada pues es un procedimiento y se empleará el parámetro c para guardar el resultado.

8.8.Pasar parámetros de tipo *open array*

FreePascal admite un tipo especial de parámetros que reciben el nombre de array abierto, open array, que permite especificar un array de mida variable y tipo determinado como parámetro de una función. Para hacerlo hay que emplear la sintaxis array of tipoDato al parámetro en cuestión.

Vamos a implementar una función que escriba una serie de enteros pasados como parámetros.

```
procedure EscribirEnteros(n : array of Integer);
var
  i : integer;
begin
  for i := Low(n) to High(n) do
  begin
    Write(' ', n[i]);
  end;
end;
```

Obsérvese que tenemos que emplear High y Low pues no sabemos el tamaño del array pasado como parámetro. Los datos que podemos pasar a estos parámetros son literales, constantes y variables. Los literales tienen una sintaxis parecida a los literales de conjuntos. Podemos llamar la función con el literal siguiente :

```
EscribirEnteros([1, 2, 8, 3]);
```

Esto escribiría en la pantalla :

```
 1 2 8 13
```

También podemos pasar una variable como parámetro. En este caso tendrá que ser un array de tamaño fijo.

```
var
  prueba : array[1..3] of integer;
begin
  prueba[1] := 4;
  prueba[2] := 2;
  prueba[3] := 6;
  EscribirEnteros(prueba);
end.
```

No hay ningún problema por pasar parámetros por referencia o constantes. Por ejemplo en el caso siguiente se llena el array con el valor del índice al cuadrado.

```
procedure Cuadrado(var n : array of Integer);
var
  i : integer;
begin
  for i := Low(n) to High(n) do
  begin
```

```
    n[i] := i*i;
  end;
end;
```

Al ser un parámetro por referencia no podremos pasar literales pero si variables. Es muy recomendable emplear parámetros constantes siempre que sea posible pues la función o procedimiento tardará menos tiempo en ejecutarse.

No hay que confundir los parámetros array abiertos con los array cerrados. En el caso siguiente se puede apreciar la diferencia.

```
program ArraysAbiertos;
type
  TArray3 = array[1..3] of Integer;

procedure Cuadrado(var n : array of Integer);
var
  i : integer;
begin
  for i := Low(n) to High(n) do
  begin
    n[i] := i*i;
  end;
end;

procedure Cuadrado3(var n : TArray3);
var
  i : integer;
begin
  for i := Low(n) to High(n) do
  begin
    n[i] := i*i;
  end;
end;

var
 prueba : array[1..4] of integer;
begin
  Cuadrado(prueba); // Correcto
  Cuadrado3(prueba); // Incorrecto pues prueba no es del tipo TArray3
end.
```

Si el array prueba hubiera sido de 3 elementos y de índice [1..3] (pero no [0..2] u otros) el compilador nos hubiera permitido pasarlo a Cuadrado3. En el caso de los arrays fijos no es posible pasar como parámetro un literal con la sintaxis de los arrays abiertos aunque el parámetro sea constante o por valor. De hecho, en funciones y procedimientos con parámetros de tipo array fijo sólo es posible pasar variables.

8.9.Funciones declaradas *a posteriori*

En algunos casos muy extraños podemos necesitar que dos funciones se llamen la una a la otra de forma recíproca. El problema se encuentra en que si declaramos A antes que B entonces A no puede llamar a B y viceversa. Para resolver este problema podemos posponer la implementación de la función a un punto inferior de forma que al existir la declaración sea sintácticamente correcto llamarla y el compilador permita la llamada mutua de funciones.

Para indicar al compilador de que una función será declarada más abajo hay que emplear la palabra forward después de su declaración Vamos a implementar un par de procedimientos llamados Ping y Pong que se irán pasando un entero que decrecerá. El ejemplo carece de utilidad práctica alguna.

```
procedure Ping(n : integer); forward; // Vamos a implementarlo más abajo

procedure Pong(n : integer);
begin
  if n > 0 then
  begin
    Writeln('Pong ', n);
    Ping(n-1); // Llamamos a Ping
  end;
end;

procedure Ping(n : integer); // Implementamos Ping
begin
  if n > 0 then
  begin
    Write('Ping ', n, ' ');
    Pong(n); // Llamamos a Pong
  end;
end;

begin
  Ping(5);
end.
```

El resultado del programa sería :

```
Ping 5 Pong 5
Ping 4 Pong 4
Ping 3 Pong 3
Ping 2 Pong 2
Ping 1 Pong 1
```

Es importante implementar, en algún punto del programa posterior a la declaración forward, la función o procedimiento. En caso contrario el compilador indicará un error sintáctico. Tampoco es posible declarar una misma función doblemente forward aunque tampoco tenga ningún sentido hacerlo.

8.10.La función *exit*

Muchas veces nos interesará salir de una función antes de que ésta termine o que no ejecute más instrucciones. Sobretodo cuando en un punto ya conocemos el resultado y no hay que realizar nada más. En estos casos podemos salir de la función o procedimiento con la orden Exit. Obsérvese, que después de llamar a Exit ya no se ejecuta ninguna instrucción más de la función o del procedimiento. No hay que llamar a Exit dentro del bloque principal puesto que se terminaría el programa, lo cual no suele ser recomendable.

```
function EsPrimo(n : integer) : boolean;
{ Devuelve true si n es primo, false en caso contrario
 Esta función es poco eficiente y es un ejemplo del uso de Exit }
var
  i : integer;
begin
  if n < 0 then n := Abs(n); // Trabajaremos con positivos
  if n = 1 then
  begin
    // En el número 1 no tiene sentido hablar de primalidad
    EsPrimo := false;
    Exit; // Salimos de la función
  end;
  // Comprobamos si es primo
  for i := 2 to (n-1) do
  begin
    if n mod i = 0 then
    begin // Es divisible por lo que no es primo
      EsPrimo := false;
      Exit; // Salimos de la función
    end;
  end;
  // Si llegamos aquí es porque no se ha dividido con los anteriores
  EsPrimo := true; // por tanto es primo
end;
```

8.10.1.Devolver al estilo de C/C++

Las funciones de C y C++ devuelven un valor mediante la palabra reservada `return` seguida de una expresión. Esto hace que la función termine y devuelva el valor de la expresión. En Pascal, la asignación al identificador de la función no hace que ésta termine por lo que muchas veces necesitaremos añadir un Exit después de la asignación. FreePascal extiende la función Exit con un parámetro que es el valor que devolverá la función.

Por ejemplo, podemos simplificar las dos instrucciones siguientes :

```
EsPrimo := false;
```

```
Exit; // Salimos de la función
```

en la siguiente :

```
Exit(false);
```

Esta sintaxis es mucho más parecida a la orden `return` de C/C++ y facilita el trabajo a la hora de portar funciones de C a Pascal.

9.Recursividad

9.1.Ventajas de la recursividad

Muchas operaciones requieren procesos repetitivos y se pueden implementar con sentencias del tipo for, while o repeat. Algunos casos, pero, no se pueden diseñar con sentencias iterativas puesto que ya no es posible o es muy complejo. Como alternativa disponemos de una herramienta muy potente, a la par que arriesgada y compleja, como es la recursividad.

Donde tiene más utilidad la recursividad es en procesos que podríamos llamar inductivos. Dada una propiedad P(n), donde n pertenece a los enteros, que se cumple para n_0 y que si se cumple para un n cualquiera implica que también se cumple para n+1, entonces es muy probable que se tenga que codificar mediante recursividad.

Qué es recursividad ? Hablamos de recursividad cuando una función se llama a si misma, con parámetros distintos, para llegar a un caso en el cual no haya que segur llamándose a si misma y se pueda resolver la llamada inicial.

La ventaja de la recursividad es poder escribir algoritmos muy eficaces y rápidos con pocas líneas de código.

9.2.Un ejemplo sencillo : el factorial

Sabemos que la operación factorial n! equivale a n! = n·(n-1)·(n-2)...2·1 y 0! = 1. Una forma de codificación sin emplear la recursividad es la siguiente :

```
function Factorial(n : integer) : Integer;
var
  i, temp : integer;
begin
  temp := 1;
  for i := 1 to n do
  begin
    temp := i*temp;
  end;
  Factorial := temp;
end;
```

Si queremos codificar el factorial de forma recursiva hay que tener en cuenta el caso sencillo cuando n = 0, 0! = 1, y que n! = n·(n-1)! El código siguiente es la forma recursiva del factorial :

```
function Factorial(n : integer) : Integer;
begin
  if n = 0 then
```

```
  begin
    Factorial := 1
  end
  else
  begin
    Factorial := n*Factorial(n-1);
  end;
end;
```

Como se ve, siempre llega algún momento en el cual ya no hay más llamadas recursivas puesto que n se vuelve cero en algún momento (siempre la n inicial sea mayor que cero, claro).

Es importante que nuestro código termine en algún momento las llamadas recursivas, pues de no hacerlo la llamada a las funciones sería infinita y gastaríamos toda la memoria dedicada a las variables locales (memoria stack) y se produciría un error runtime llamado Stack Overflow que provocaría que nuestro programa finalizara anormalmente.

10.Punteros y memoria

10.1.Variables estáticas

Hasta ahora todas las variables que hemos empleado se gestionan de forma totalmente automática por el compilador. Éste se encarga de añadir el código necesario para crearlas al iniciar el programa y liberarlas cuando termina. Reciben el nombre de variables estáticas.

En algunos casos necesitaremos acceder directamente a las posiciones de memoria de nuestras variables, en otros necesitaremos zonas de memoria con tamaños especiales, etc. En estos casos tendremos que emplear los punteros.

10.2.Estructura de la memoria

La memoria de un ordenador está formada por un conjunto determinado de celdas de un byte a las cuales podemos acceder mediante su dirección. Esta dirección es única para cada posición de memoria y viene expresada, en ordenadores superiores al 386, por variables enteras de 32-bits. De esta forma, teóricamente, se podrían direccionar 2^{32} posiciones de memoria diferentes lo que equivale 4 Gb de memoria RAM. En la práctica no se llega nunca a esta cifra debido a imposibilidades técnicas evidentes.

10.3.Punteros con tipo

Un puntero es una variable que apunta a una posición de memoria. Los punteros con tipo se declaran añadiendo el símbolo ^ (circunflejo) antes de su tipo.

```
var
  PunteroEntero : ^Integer;
```

Antes de poder emplear un puntero con tipo habrá que reservar memoria. Para esto tenemos la función New que devolverá una dirección a la memoria alojada. New se encarga de reservar suficiente memoria para nuestro puntero con tipo.

```
New(PunteroEntero);
```

Cuando ya no necesitemos el puntero es necesario liberarlo. De esta forma el sistema se da por enterado de que esta posición de memoria está libre y puede ser ocupada por otros datos. Si no lo liberásemos, el sistema entendería que está ocupado con lo que no intentaría alojar más datos en esta

posición y malgastaríamos la memoria. A partir de este momento ya no será válido acceder al contenido del puntero. Para hacerlo emplearemos el procedimiento Dispose :

```
Dispose(PunteroEntero);
```

Ya que PunteroEntero sólo apunta a una dirección de memoria será necesario poder acceder a su contenido. Para esto hay que emplear el operador de desreferenciación ^.

```
PunteroEntero^ := 10;
```

Obsérvese que es radicalmente diferente la asignación `PunteroEntero^ := 10;` que la asignación `PunteroEntero := 10;` En este último caso hacemos que el puntero PunteroEntero apunte a la dirección de memoria 10. Este es un error bastante habitual ya que el compilador autoriza las asignaciones directas al compilador. En este caso es más notable pues el puntero es de tipo entero.

```
program PunterosConTipo;
var
  PunteroEntero : ^Integer;
begin
  New(PunteroEntero);
  PunteroEntero^ := 5; // Asignamos 5 a la posición apuntada por el puntero
  { PunteroEntero := 5; PELIGRO! El puntero apuntaría a la posición nº 5 }
  Dispose(PunteroEntero);
end.
```

Hay que ir con mucho cuidado cuando hacemos asignaciones a punteros sin desreferenciar pues es un error bastante habitual. Además, si el puntero cambia de valor, entonces apunta a otra dirección de la memoria y por tanto el valor de la posición anterior se pierde inevitablemente con lo que no podremos liberar la memoria que habíamos reservado inicialmente pues intentaríamos liberar otra posición y esto es un peligro potencial, pues puede ser memoria que no tenga que ser liberada.

En este ejemplo hemos reservado memoria para un puntero y la hemos liberado. Mediante el operador de desreferenciación hemos modificado el contenido de la memoria direccionada por el puntero.

Si queremos que un puntero no apunte en ningún lugar empleamos la palabra reservada `nil`.

```
PunteroEntero := nil;
```

Puede semblar extraño en un primer momento que un puntero no apunte en ningún lugar. Si intentamos obtener algún valor del puntero mediante PunteroEntero^ obtendremos un error runtime de protección general.

Es importante ver que hasta que no se reserva memoria para un puntero no podemos acceder a su contenido ya que el valor que contiene no apunta en ningún lugar, tiene el valor nil. Podemos comprobar si un puntero no apunta en ningún sitio si su valor es nil.

```
if PunteroEntero = nil then ...
```

Los punteros del mismo tipo son compatibles por asignación. Además las modificaciones que hagamos en un puntero que apunta a una misma posición apuntada por otro puntero se ven ambos punteros.

```
program PunterosConTipo;
var
  PunteroEntero, PunteroModificar : ^Integer;
begin
  New(PunteroEntero);
  PunteroEntero^ := 5;
  PunteroModificar := PunteroEntero; { Asignamos a PunteroModificar la misma
                                       dirección que PunteroEntero }
  Writeln(PunteroEntero^); {5}
  Writeln(PunteroModificar^); {5}

  PunteroModificar^ := 12;

  Writeln(PunteroEntero^); {12}
  Writeln(PunteroModificar^); {12}

  Dispose(PunteroEntero);

  { A partir de aquí PunteroEntero^ i PunteroModificar^ ya no son válidos }
end.
```

Vemos que modificar el contenido de esta posición mediante uno de los dos punteros produce el mismo resultado ya que los dos punteros se refieren a la misma posición de memoria y por tanto apuntan al mismo valor. Consíderese también el hecho e que liberar una posición de memoria invalida todos los punteros que apunta a ella por lo que PunteroModificar y PunteroEntero ya no pueden ser desreferenciados correctamente. Es importante ver que PunteroModificar no se tiene que liberar con Dispose ya que tampoco hemos reservado memoria para él.

10.3.1.El operador de dirección @

Si empleamos el operador @ delante del nombre de una variable estática automáticamente obtendremos su dirección de memoria. Por tanto podemos asignar este resultado a un puntero del mismo tipo que la variable estática y realizar modificaciones sobre la variable estática mediante el puntero.

```
program PunterosConTipo;
var
  PunteroEntero : ^Integer;
  Entero : Integer;
begin
  PunteroEntero := @Entero; { Obtenemos su dirección y la guardamos en
                            PunteroEntero }

  Entero := 10;
  Writeln(Entero); {10}
  Writeln(PunteroEntero^); {10}

  PunterEnter^ := 12;
  Writeln(Entero); {12}
  Writeln(PunterEntero^); {12}
end.
```

Aquí tampoco hay que liberar PunteroEntero ya que no hemos reservado memoria para él. Simplemente lo hemos asignado para que apunte a la dirección de la variable Entero.

10.3.2.Copiar los contenidos de un puntero en otro

Antes hemos visto que al asignar un puntero a otro simplemente hacíamos que apuntase a la dirección asignada pero no se transfería el contenido. Supongamos hemos reservado memoria para los punteros PunteroEntero y PunteroModificar, ambos de tipo Integer.

```
begin
  New(PunteroEntero); // Reservamos memoria para PunteroEntero
  New(PunteroModificar); // Reservamos memoria para PunteroModificar
  PunteroEntero^ := 10;
  PunteroModificar^ := PunteroEntero^; // Copiamos el contenido
  Writeln(PunteroModificar^); {10}
  Dispose(PunteroEntero);
  Dispose(PunteroModificar);
end.
```

Si en vez de hacer una asignación mediante el operador de desreferenciación hubiésemos hecho la asignación siguiente :

```
PunteroModificar := PunteroEntero;
```

No habríamos podido liberar la zona reservada inicialmente para PunteroModificado ya que al hacer Dispose(PunteroModificar) tendríamos un error al intentar liberar una zona de memoria ya liberada. En cualquier otra combinación :

```
PunteroModificar^ := PunteroEntero;
```

```
// o
PunteroModificar := PunteroEntero^;
```

El compilador no nos habría permitido realizar la asignación pues los elementos no son compatibles por asignación.

10.3.3.Aritmética e indexación de punteros

FreePascal permite realizar aritmética de punteros mediante las operaciones suma y resta y los procedimientos Inc y Dec.

Dados dos punteros con tipo P1 y P2 podemos realizar operaciones como las siguientes :

```
P1 := P1 + 1; {Ahora P1 apunta un byte más adelante de la dirección inicial}
P1 := P1 - 1; {Ahora P1 vuelve apuntar a la dirección inicial}
P1 := P2 + 2; {Ahora P1 apunta dos bytes más adelante de P2}
P1 := P1 - 2; {Ahora P1 apunta a la dirección de P2}
```

La función Inc incrementa la dirección del puntero en n bytes donde n bytes es el tamaño del tipo de puntero. Por ejemplo, dado el puntero P de tipo ^Longint entonces :

```
Inc(P); { Equivale a P := P + 4 }
```

ya que un Longint tiene un tamaño de 4 bytes. La función Dec hace lo mismo pero en vez de incrementar n bytes los decrementa.

También podemos indexar un puntero como si de un array se tratara. Obsérvese el ejemplo siguiente :

```
program PunteroArray;
var
  PunteroByte : ^Byte;
  Entero : Word;
begin
  Entero := 18236; { $403C }
  PunteroByte := @Entero;
  Writeln('Byte de menos peso ', PunteroByte[0]); { $3C = 60 }
  Writeln('Byte de mayor peso ', PunteroByte[1]); { $40 = 71 }
end.
```

En este ejemplo hemos empleado un entero de tipo Word que consta de dos bytes, el de mayor peso o superior, y el de menor peso o inferior, y además no tiene signo. Ya que en la plataforma Intel los bytes de menos peso se almacenan primero, leemos PunteroByte[0] y después PunteroByte[1]. En otras plataformas quizá hubiéramos encontrado primero el byte superior y después el byte inferor. En este caso

concreto no es necesario emplear el operador de desreferenciación pues se sobreentiende con la sintaxis de array.

10.4.Punteros sin tipo

Es posible emplear un tipo de puntero que no va asociado a un tipo en concreto. Este tipo que recibe el nombre de Pointer permite apuntar a zonas de memoria genéricamente. Si queremos reservar memoria con una variable de tipo Pointer tendremos que emplear GetMem y FreeMem para liberarla. El uso de punteros son tipo es aún más peligroso que con tipo pues no podemos asignar valores al contenido. Las únicas asignaciones posibles son entre otros punteros, con o sin tipo.

```
program PunteroSinTipo;
var
  Puntero : Pointer;
  PunteroEntero : ^Integer;
begin
  GetMem(Puntero, SizeOf(integer));
  PunteroEntero := Puntero;
  PunteroEntero^ := 10;
  FreeMem(Puntero, SizeOf(integer));
end.
```

Las funciones FreeMem y GetMem exigen que se les explicite el tamaño que hay que reservar y liberar. En este ejemplo hemos reservado espacio para un entero y hemos asignado su dirección a la variable PunteroEntero con el cual podemos trabajar como si de un entero se tratara. Nótese que se ha obtenido el tamaño del entero mediante SizeOf(integer) y no SizeOf(^Integer) o SizeOf(PunteroEntero) ya que cualquier puntero tiene un tamaño de 4 bytes (32-bits) para poder apuntar a una dirección de memoria.

10.5.El tipo AnsiString

El tipo AnsiString es el que se llama cadena larga. No es una cadena propiamente Pascal pero permite superar la limitación de los 255 caracteres impuestos por las cadenas Pascal. Por otra parte tienen un comportamiento especial pues se tratan como punteros por lo que si intentamos escribir en un archivo un AnsiString obtendremos que hemos copiado un entero de 4 bytes y no la cadena en sí.

Cuando una cadena de tipo AnsiString se asigna a un valor vacío ('') entonces esta toma el valor nil. Cuando le asignamos una variable o un literal compatibles, entonces se reserva suficiente memoria para alojar la cadena. Además, las cadenas AnsiString tienen un contador de referencias interno. Si hacemos la asignación siguiente :

```
S1 := S2;
```

Entonces el contador de S2 se reduce a cero para indicar que no tiene un valor propio, y el contador de S1 se incrementa y se copia la dirección de S1 en S2 con lo cual se acelera la gestión de cadenas. Al leer S2 estamos leyendo de hecho S1. En cambio, si acto segudio hacemos :

```
S2 := S2 + '.';
```

Entonces el contador de S1 se reduce a 1, el contador de S2 se incrementa a 1 y se copia el contenido de S1 en la region reservada por S2 y se añade el punto. Todo este proceso es automático y es el compilador quien se encarga de añadir el código necesario para estas operaciones de forma que sea transparente para el programador.

Las cadenas largas son totalmente compatibles con las cadenas de tipo String[n] o ShortString. Si combinamos cadenas largas y cortas en una expresión, el resultado en general es AnsiString pero si lo asignamos a un ShortString se transforma automáticamente.

En el caso de los AnsiString no podemos saber el tamaño máximo de la cadena con High pues son cadenas dinámicas. Podemos especificar su tamaño máximo con SetLength.

```
SetLength(S1, 5); // Ahora S1 ha reservado espacio para 5 caracteres
```

Pero si hacemos una asignación a la cadena que exceda de su tamaño este no se trunca, como pasaría con los string Pascal, sino que crece adecuadamente.

```
program CadenasAnsi;
var
  S1 : AnsiString;
begin
  SetLength(S1, 3);
  Writeln(Length(S1)); {3}
  S1 := 'Hola que tal !';
  Writeln(Length(S1)); {14}
end.
```

Los string Pascal aparte de estar limitados a 255 caracteres no son compatibles con las cadenas finalizadas en nulo que se emplean en C/C++ y por tanto tampoco lo son cuando llamamos funciones escritas en este lenguaje. Las cadenas AnsiString, a la contra, son más fáciles de convertir a cadenas C.

10.5.1.La directiva de compilador $H

La directiva de compilador $H permite determinar si el tipo String (sin indicador de longitud) representa a una cadena larga, AnsiString, o a una cadena corta, ShortString o String[255]. Cuando $H está activada entonces el tipo String representa un AnsiString. En caso contario, representa un ShortString. La forma larga de la directiva $H es $LONGSTRINGS. Por defecto $H está activada. Esta directiva es local, de forma que se puede intercalar dentro del código tal como hacíamos con $I y sólo cambia si aparece una directiva $H posterior.

```
{$H+} // String = AnsiString
{$H-} // String = ShortString
{$LONGSTRINGS ON} // String = AnsiString
{$LONGSTRINGS OFF} // String = ShortString
```

10.6.Cadenas finalizadas en carácter nulo

Algunos lenguajes de programación como C/C++ les falta un tipo especial para las cadenas de caracteres. Como solución se emplean arrays de caracteres sin índice que terminan cuando se encuentra un carácter nulo (de valor ASCII cero). Como hemos comentado esta estructura es incompatible con el tipo string de Pascal. Estas se basan en un byte inicial que indica el nombre de caracteres de la cadena (el valor que devuelve Length) y los bytes restantes son los caracteres de la cadena. El máximo valor que puede expresar el primer byte (de índice cero en la cadena) es 255 y por tanto las cadenas Pascal están limitadas a 255 caracteres como máximo.

Con este motivo existe el tipo PChar. Se puede entender como la equivalencia en Pascal de las cadenas terminadas en nulo. Formalmente el tipo PChar no es nada más que un puntero a un carácter. Esto es así ya que en las funciones y procedimientos en los que necesitemos pasar una cadena terminada en nulo no se pasa nunca la cadena en si, sólo el puntero al primer carácter de ésta. De esta forma la cadena puede ser muy larga pero la información pasada a la función sólo son los 4 bytes del puntero.

El asunto más complicado, por ahora, con las cadenas PChar es trabajar con ellas. Especialmente cuando hay que mezclarlas o convertir en cadenas AnsiString.

Un PChar lo podemos asignar directamente a un literal de cadena. Podemos asignar el contenido de un PChar a otro y a diferencia de los punteros habituales, la modificación de éste último no implica la modificación del primero.

```
program CadenesTerminadasNulo;
var
  a, b : PChar;
begin
  a := 'Hola';
  b := a;
```

```
  b := 'Adios';
  Writeln(a); { Hola }
  Writeln(b); { Adios }
end.
```

Muchas veces nos encontraremos en la tesitura de tener que pasar una cadena PChar a una función pero solo disponemos de un AnsiString o ShortString. En el caso de el AnsiString podemos hacer un typecasting directamente a PChar.

```
program CadenasTerminadasNulo;

procedure Escribir(Cadena : PChar);
begin
  Writeln(Cadena);
end;

var
  a : AnsiString;
begin
  Escribir('Hola'); // Los literales son correctos
  a := 'Hola';
  Escribir(a); // Esto es ilegal
  Escribir(PChar(a)); // Esto es perfectamente legal
end.
```

En el caso de un ShortString hay varias soluciones. La más simple es asignarla a un AnsiString y después aplicar el typecasting. Téngase en cuenta que en el caso de parámetros por referencia los typecastings no están permitidos y los parámetros tienen que ser del mismo tipo idéntico.

Para transformar una cadena PChar a su equivalente AnsiString o ShortString no hay que hacer ninguna operación especial. El compilador se encarga de transformar el PChar a String de forma automática.

10.6.1.El tipo *array[0..X] of Char*

Antes de poder emplear una variable PChar es importante inicializarla. Una forma poco elegante consiste en asignarla a un literal de cadena. Otra forma es emplear las complicadas rutinas de la unit Strings, que sólo operan con cadenas PChar, para obtener un puntero válido del tamaño deseado que posteriormente tendremos que liberar. Otra solución más simple son los arrays de caracteres indexados en cero que son totalmente compatibles con los PChar, se inicializan automáticamente y pueden tener el tamaño que queramos incluso más de 255 caracteres. El único caso en el que no serían compatibles son en los parámetros por referencia.

```
program CadenasTerminadasNulo;

procedure Escribir(Cadena : PChar);
begin
  Writeln(Cadena);
end;

var
  a : array [0..499] of Char; // El tamaño que queramos o necesitemos
begin
  a := '1234'; // Hasta quinientos caracteres
  Escribir(a);
end.
```

El tipo array of Char es compatible en asignación con ShortString y AnsiString.

Es un error bastante habitual declarar un array of Char demasiado corto lo que hace que los caracteres de más se trunquen. Este hecho puede provocar errores difíciles de detectar y por tanto es recomendable declarar arrays suficiente grandes para nuestros datos.

10.7.El tipo función/procedimiento

Es posible definir un tipo de dato que sea evaluable en una expresión, función, o llamado desde una instrucción mediante el que se llama tipo función, y extensivamente, tipo procedural.

El tipo procedural tiene utilidad en algunos casos muy concretos, por ejemplo en programación con DLL u otras librerias de enlace dinámico y por tanto veremos pocos ejemplos. Básicamente permite definir una variable de tipo de procedimiento o función que apuntará a otra función o procedimiento, una función de una DLL en memoria por ejemplo, y que se podrá tratar como si de un procedimiento o función se tratara. Vamos a ver un caso sencillo para una función. Para los procedimientos es muy parecido. Téngase en mente que vamos a emplear una función ya definida. Obsérvese también que es importate que los parámetros sean iguales, o almenos el numero de bytes de los parámetros idéntico, para evitar errores de desbordamiento de pila y de protección general. El ejemplo es bastante inútil pero es suficientemente ilustrativo.

Vamos a resolver una ecuación de segundo grado. Recordemos que las ecuaciones de segundo grado $ax^2 + bx + c = 0$ se resuelven con la fórmula siguiente :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En caso de que el discriminante Δ = b2 - 4ac sea menor que cero no habrá ninguna solución, en caso de que sea igual a cero tendremos una solución x = -b/2a y en caso de que sea positivo tendremos dos soluciones para los dos signos de la raiz cuadrada.

```
const
  NO_SOLUCION = 0;
  UNA_SOLUCION = 1;
  DOS_SOLUCIONES = 2;

function ResolverEq2nGrado(a, b, c : real; var x1, x2 : real) : integer;
{
  Resolución de ecuaciones de 2º grado en los reales

  La función devuelve
  NO_SOLUCION si no hay solución
  UNA_SOLUCION si sólo hay una solución
  DOS_SOLUCIONES si hay dos soluciones
}
var
  discr : real; // El discriminante
begin
  discr := b*b - 4*a*c;
  if discr < 0 then
  begin
   ResolverEq2nGrado := NO_SOLUCION;
  end
  else
  begin
    if discr = 0 then
    begin
      ResolverEq2nGrado := UNA_SOLUCION;
      x1 := -b / (2*a);
      x1 := x2; // Dejamos constancia de q son iguales
    end
    else // Sólo puede ser discr > 0
    begin
      ResolverEq2nGrado := DOS_SOLUCIONES;
      x1 := (-b + Sqrt( discr )) / (2*a);
      x2 := (-b - Sqrt( discr )) / (2*a);
    end;
  end;
end;
```

Una vez ya tenemos definida la función vamos a ver como podemos llamarla mediante un tipo de función. Es importante definir el tipo de función con los mismos parámetros y especificando procedure o function según convenga.

```
type
  TFuncion = function(a, b, c : real; var x1, x2 : real) : integer;
```

En este caso hemos copiado incluso el nombre de las variables, lo único que realmente importa es que tengan el mismo tamaño y mismo tipo de parámetro, por lo que la declaración siguiente es igual a la anterior :

```
type
  TFuncion = function(A1, A2, A3 : real; var A4, A5 : real) : integer;
```

Una vez tenemos declarado el tipo podemos declarar una variable de este tipo.

```
var
  Funcion : TFuncion;
```

Lo más interesante empieza ahora. Es necesario asignar la dirección de memoria de Funcion a la dirección de ResolverEq2nGrado. El código del programa queda de la forma siguiente :

```
program TipoFuncion;

const
  NO_SOLUCION = 0;
  UNA_SOLUCION = 1;
  DOS_SOLUCIONES = 2;

function ResolverEq2nGrado(a, b, c : real; var x1, x2 : real) : integer;
begin
  ... // El código de la función está listado más arriba
end;

type
  TFuncion = function(a, b, c : real; var x1, x2 : real) : integer;

var
  Funcion : TFuncion;
  x1, x2, a, b, c : Real;

begin
  Write('A : '); Readln(a);
  Write('B : '); Readln(b);
  Write('C : '); Readln(c);

  Funcion := @ResolverEq2nGrado; { Asignamos a Funcion la dirección de
                                   ResolverEq2nGrado }


  case Funcion(a, b, c, x1, x2) of
    NO_SOLUCION : begin
                    Writeln('Esta ecuación no tiene solución real');
                  end;
    UNA_SOLUCION : begin
                     Writeln('Una solución : X=', x1:5:2);
```

```
                end;
    DOS_SOLUCIONES : begin
                       Writeln('Dos soluciones');
                       Writeln('X1=', x1:5:2);
                       Writeln('X2=', x2:5:2);
                     end;
  end;
end.
```

Obsérvese que hemos asignado a Funcion la dirección de ResolverEq2nGrado de forma que cuando llamamos a Funcion con los parámetros adecuados en realidad estamos llamando a ResolverEq2nGrado.

Para que los reales se vean con una cantidad adecuada de cifras y decimales emplearemos el añadido :5:2. Esta opción sólo está disponible en Writeln establece 5 cifras en total 2 de las cuales seran decimales del real. En el caso que el real tenga menos de 5 cifras, se escriben espacios en blanco hasta completarlas. Para los enteros tambien se puede emplear pero sólo se pueden especificar las cifras.

```
{ La constante Pi ya está declarada en Pascal }
Writeln(Pi:2:0); {2 cifras, 0 decimales}
Writeln(Pi:0:5); {0 cifras, 5 decimales}
Writeln(Pi:7:3);
Writeln((Pi*10):7:3);
```

Las instrucciones anteriores escribirían en pantalla :

```
␣3
3.14159
␣␣3.142
␣31.416
```

Dónde ␣ representa un espacio en blanco. Obsérvese que el punto no se cuenta como una cifra decimal. Este valor que especifiquemos para cifras y decimales no tiene porque ser un literal, puede ser una variable entera o una constante entera.

10.7.1.Convenciones de llamada

No todos los lenguajes de programación llaman las funciones y procedimientos de la misma forma. Por ejemplo, el compilador Borland Delphi por defecto pasa los parámetros de izquierda a derecha mientras que FreePascal y los compiladores de C los pasan de derecha a izquierda. En función de quien se encarga de liberar la pila de la función y en el orden en el que se pasan los parámetros tendremos una convención de llamada u otra.

Para llamar una función que se encuentre en memoria es conveniente emplear la convención adecuada. Básicamente las convenciones más habituales son cdecl y stdcall, esta última es la convención por defecto de FreePascal.

Convención de llamada	**Paso de parámetros**	**Liberación de memoria**	**Parámetros en los registros ?**
por defecto	Derecha a izquierda	Función	No
cdecl	Derecha a izquierda	Quien hace la llamada	No
export	Derecha a izquierda	Quien hace la llamada	No
stdcall	Derecha a izquierda	Función	No
popstack	Derecha a izquierda	Quien hace la llamada	No
register[4]	Izquierda a derecha	Función	Sí
pascal	Izquierda a derecha	Función	No

Convenciones export, cdecl i popstack

Se emplean en funciones que tendrán que ser llamadas por código en C o que han sido escritas con la convención de llamada de C/C++. Popstack además nombra la rutina con el nombre que el compilador FreePascal le daría si fuera una función normal.

Convenciones pascal i register.

La convención register es ignorada por el compilador que la sustituye por la convención stdcall. La convención pascal tiene una mera finalidad de compatibilidad hacia atrás.

Convención stdcall

La convención stdcall es la convención de llamada por defecto de FreePascal. Además, es la convención de llamada de las funciones de la API de Win32.

Para indicar al compilador qué convención de llamada tiene que emplear sólo hay que especificar la convención de llamada después de la declaración de la función. Téngase en cuenta de que las convenciones de llamada son mutuamente excluyentes entre si, por lo que, no podemos definir más de una. Obsérvese el ejemplo siguiente :

[4]Esta convención está incorporada en el compilador FreePascal pero se ignora y es sustituida por la convención stdcall. Es la convención por defecto que emplea el compilador Borland Delphi de 32-bits.

```
procedure LlamadaStdCall; stdcall; // Convención stdcall
begin
  ...
end;

procedure LlamadaCdeCl; cdecl; // Convención cdecl
begin
  ...
end;
```

Vamos a ver un ejemplo práctico de las diferencias entre convenciones a la hora de pasar parámetros. Emplearemos la convención pascal y la convención stdcall, por defecto, para pasar cuatro enteros de 4 bytes sin signo, Cardinal. Después veremos que es posible acceder a ellos y obtener su dirección en memoria.

```
program ConvencionLlamada;

procedure IzquierdaDerecha(a, b, c, d : Cardinal); pascal;
var
  Puntero : ^Cardinal;
begin
  Puntero := @a;
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')' ); {a}
  Dec(puntero);
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {b}
  Dec(puntero);
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {c}
  Dec(punter);
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {d}
end;

procedure DerechaIzquierda(a, b, c, d : Cardinal); stdcall;
var
  Puntero : ^Cardinal;
begin
  Puntero := @a;
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {a}
  Inc(puntero);
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {b}
  Inc(puntero);
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {c}
  Inc(puntero);
  Writeln(Puntero^, ' (', Cardinal(Puntero), ')'); {d}
end;

begin
  Writeln('Izquierda a derecha :');
  IzquierdaDerecha(1, 2, 3, 4);
  Writeln;
  Writeln('Derecha a izquierda :');
  DretaIzquierda(5, 6, 7, 8);
```

```
end.
```

Cuando ejecute el programa obtendrá un resultado parecido al siguiente :

```
Izquierda a derecha :
1 (39321120)
2 (39321116)
3 (39321112)
4 (39321108)

Derecha a izquierda :
5 (39321108)
6 (39321112)
7 (39321116)
8 (39321120)
```

El programa nos muestra el valor de las variables que hemos pasado a las dos funciones y entre paréntesis nos indica la dirección en memoria del parámetro en cuestión. Es legal un typecasting de puntero a Cardinal pues un puntero no es nada más que un entero de cuatro bytes para acceder a una posición de memoria. En el código obtenemos la dirección del parámetro a y después obtenemos los otros parámetros mediante aritmética de punteros. Recuerde que Inc y Dec incrementan o decrementan el puntero en tantos bytes como el tamaño del tipo de puntero. Un cardinal tiene un tamaño de cuatro bytes por tanto el puntero apuntará cuatro bytes más adelante o atrás después de un Inc o Dec, respectivamente.

Los parámetros se pasan, en el primer caso, de izquierda a derecha, o sea, del último al primero. De forma que el último parámetro d es el que se encuentra en las posiciones más bajas de la memoria y los parámetros van pasando del último al primero, a, el cual tendrá la posición más a la derecha, o sea la posición de memoria más alta.

En el segundo caso los parámetros se pasan de derecha a izquierda, o sea, del primero al último. En este caso, el último parámetro d está en la posición de memoria más alta mientras que el primer parámetro a está en la posición más a la derecha, o sea, en la posición más baja de la memoria.

Izquierda a derecha

Posiciones de memoria	...	39321108	39321112	39321116	39321120
	...	d	c	b	a

Derecha a izquierda

	...	a	b	c	d

Es importante emplear la convención adecuada cada vez que llamamos a una función. Generalmente no tendremos que preocuparnos pues el compilador sabe qué convención se está

empleando en las funciones o procedimientos que declaremos en el código. En el caso de que empleemos funciones definidas en librerías escritas en otros lenguajes de programación habrá que tener mucho cuidado y especificar la convención adecuada, en otro caso emplearía la convención stdcall por defecto. También es posible especificar una convención a un tipo procedural :

```
type
 TIzquierdaDerecha = procedure (a, b, c, d : Cardinal); pascal;
```

11.Units

11.1.La necesidad de las units

Hasta ahora hemos empleado algunas funciones que se encontraban en algunas units estándar de FreePascal. Las units permiten reunir funciones, procedimientos, variables y tipos de datos para disponerlos de forma más ordenada y poderlos reutilizar en otras aplicaciones.

Una primera forma de reutilizar el código que hemos visto ha sido mediante la declaración de funciones. Estas funciones tenían que estar declaradas antes de que las hicieramos servir. Habitualmente, pero, emplearemos funciones y procedimientos en varios programas y en otras units y no será práctico copiar de nuevo la implementación. Además, de esta forma tendríamos que cambiar muchas veces el código si lo optimizaramos o lo corrigieramos, por ejemplo. Empleando units conseguimos reutilizar mejor las funciones, procedimientos pues estas se encontrarán en la unit y cualquier cambio que necesitemos sólo se tendrá que realizar en la unit misma. Cuando necesitemos llamar las funciones y los procedimientos de la unit, o emplear variables de ésta, sólo habrá que indicar al compilador que las emplee.

11.2.Utilizar una unit

Para emplear una unit en nuestro programa ya hemos visto que es tan simple como especificarla en la cláusula uses del programa. Normalmente necesitaremos más de una, pues las separamos entre comas.

```
uses Crt, Dos, SysUtils;
```

Hay que tener en cuenta de que la unit tendrá que ser visible para el compilador. Normalmente el compilador buscará el archivo que contenga la unit en el directorio del programa más algunos directorios del compilador. También podemos especificar la posición en el disco de la unit, en caso de que no se encontrara en el directorio actual, con el parámetro `-Fu`*`directorio`* pasado al compilador, donde directorio es el directorio donde el compilador puede encontrar la unit. Otra forma es modificar el archivo `PPC386.CFG` al directorio `C:\PP\BIN\WIN32` y añadir la línea `-Fu`*`directorio`*. Si tenemos que especificar más de un directorio pues añadiremos más líneas. De esta forma es permanente cada vez que se ejecuta el compilador. Finalmente podemos emplear la directiva de compilador, dentro del código del programa o unit, `$UNITPATH` separando los directorios con puntos y comas. :

```
{$UNITPATH directorio1;..;directorion}
```

Por ejemplo:

```
{$UNITPATH ..\graficos;c:\units}
```

11.3.Crear una unit

Para crear una unit hay que tener en cuenta que el archivo que vamos a crear, con extensión PAS o PP tiene que tener el mismo nombre que le daremos a la unit. Por ejemplo si nuestra unit se llama Utilidades (recuerde que las mayúsculas son poco importantes en Pascal) el archivo tendrá que ser UTILIDADES.PAS o bien UTILIDADES.PP. Es muy recomendable poner el nombre del archivo de la unit en minúsculas sobretodo en el caso de Linux donde los archivos son sensibles a mayúsculas y minúsculas.

El encabezado de una unit tiene que empezar con la palabra reservada unit y el nombre de la unit, que tiene que coincidir con el nombre del archivo. En nuestro caso sería :

```
unit Utilidades;
```

Esta palabra reservada indica al compilador que vamos a hacer una unit y no un programa. Además, a diferencia de program, que era opcional, unit es necesaria para indicarle al compilador que lo que se va a encontrar es una unit y no un programa, como suele ser por defecto.

11.4.Estructura de una unit

Una unit está formada por tres partes bien diferenciadas : una interfaz, una implementación y una sección de inicialización.

En la interfaz se declaran todas las variables, constantes, tipos y funciones o procedimientos que queremos que estén disponibles cuando vayamos a emplear la unit. En lo que los procedimientos y funciones concierne, los declararemos como si se trataran de funciones forward pero sin la palabra reservada forward.

En la implementación implementaremos las funciones de la interfaz y podemos declarar variables, tipos, constantes y otras funciones. A diferencia de los declarados en la interfaz, los elementos declarados en la implementación sólo están disponibles dentro de la misma unit y tienen como finalidad ser elementos auxiliares en la implementación.

Finalmente el código de inicialización se ejecutará al cargase la unit. Este código se ejecuta antes que se inicie el programa y tiene como finalidad realizar tareas de inicialización de la propia unit. Como es de esperar, la unit termina con un end.

11.5.La parte *interface*

La interfaz se implementa primero y se indica su inicio con la palabra reservada `interface`. Justo después de interface podemos especificar las units que empleará la función aunque sólo es recomendable hacerlo si alguno de los tipos de datos que vamos a emplear en interface se encuentra en otra unit. Si sólo la necesitamos para ciertas funciones ya la incluiremos dentro de la implementación. Esto es así para evitar referencias circulares donde A emplea B y B emplea A, ya que el compilador emitirá un error. En la sección implementación este error no se puede dar.

Vamos a implementar diversas funciones y procedimientos simples en nuestra unit que nos faciliten un poco más el trabajo a la hora de pedir datos al usuario. Concretamente vamos a hacer diversas funciones sobrecargadas que emitirán un mensaje y comprobarán que el dato introducido es correcto. La unit empieza de la forma siguiente :

```
unit Utilidades;

interface

procedure PedirDato(Cadena : String; var Dato : String);
procedure PedirDato(Cadena : String; var Dato : Integer);
procedure PedirDato(Cadena : String; var Dato : Real);
```

Sólo lo implementaremos para estos tres tipos pues no se suelen pedir otros tipos de datos ni caracteres sueltos (en este caso pediríamos la pulsación de una tecla). Ahora sólo falta implementar las funciones en la sección de implementación.

11.6.La parte *implementation*

Una vez declarados en interface tipos de datos, constantes y variables no hay que hacer nada más. No es así con las funciones y los procedimientos que hay que implementar. La sección de implementación empieza con la palabra reservada `implementation`. No importa mucho el orden en el que las implementemos sino que realmente las implementemos, de no hacerlo el compilador nos dará error.

```
implementation

procedure PedirDato(Cadena : string; var Dada : string);
begin
  Write(Cadena); Readln(Dato);
end;

procedure PedirDato(Cadena : string; var Dato : Integer);
begin
```

```
  Write(Cadena); Readln(Dato);
end;

procedure PedirDato(Cadena : string; var Dato : Real);
begin
  Write(Cadena); Readln(Dato);
end;
end. // Fin de la unit
```

Curiosamente las implementaciones de las tres funciones son idénticas pero como ya hemos comentado no tiene porque ser así. Por ejemplo podíamos haber pedido un string al usuario y haber añadido código de comprobación para ver si es un entero o no, devolviendo true o false en función de si la función ha tenido éxito. Esto es así ya que Readln falla si al pedir un entero, o un real, el usuario introduce un dato que no es convertible a entero o real.

11.7.Inicialización de una unit

Supongamos que tenemos una unit que dispone de una orden llamada NotificarLog(Cadena : String); que se encarga de escribir en un archivo de logging (donde se anotará lo que va haciendo el programa y así si falla saber dónde ha fallado) la cadena que se especifica como parámetro. En este caso nos interesa abrir un archivo de texto para hacer las anotaciones y cerrarlo si el programa termina ya sea de forma normal o anormal. Es por este motivo que las units incorporan una sección de inicialización y también de finalización. En la especificación habitual de Pascal sólo había sección de inicialización pero recientemente a Delphi se le añadió la posibilidad de disponer también de una sección de finalización y FreePascal también la incorpora.

11.7.Inicialización *"alla antigua"*

Es posible inicializar, sólo inicializar, si antes del end. final añadimos un begin y enmedio las sentencias de inicialización. Por ejemplo podríamos haber puesto unos créditos que se verían al inicializarse la unit.

```
begin
  Writeln('Unit de utilidades 1.0 - Roger Ferrer Ibáñez');
end. // Fin de la unit
```

Este tipo de inicialización tiene la desventaja de que no permite la finalización de la unit de forma sencilla. Sólo nos será útil en algunos casos. Cuando necesitemos inicializar y finalizar tendremos que emplear otra sintaxis.

11.7.Inicialización y finalización

Debido a que la mayoría de inicializaciones requerirán una finalización es posible añadir secciones de inicialización y finalización en la parte final de la unit. Estas secciones irán precedidas por las palabras reservadas `initialization` y `finalization`. En este caso especial no es necesario rodear las sentencias de begin ni de end. Tampoco es obligatoria la presencia de los dos bloques : puede aparecer uno sólo o bien los dos a la vez.

Ahora ya podemos implementar nuestra función de *logging*.

```
unit Logging;

interface

procedure NotificarLog(Cadena : string);

implementation
var // Variable de la unit no acesible fuera de la unit
  Log : Text;

procedure NotificarLog(Cadena : string);
begin
  Writeln(Log, Cadena);
end;

initialization

Assign(Log, 'LOGGING.LOG');
Rewrite(Log);

finalization

Close(Log);

end. // Fin de la unit
```

11.8.Orden de inicialización y finalización

El orden de inicialización y finalización de las units depende exclusivamente de su posición en la cláusula uses. El órden es estrictamente el que aparece en esta cláusula. Supongamos que tenemos la cláusula siguiente en nuestro programa (o unit) :

```
uses Unit1, Unit2, Unit3;
```

Supongamos que todas tienen código de inicio y final. Entonces la Unit1 se inicializaría primera, después Unit2 y después Unit3.

El proceso de finalización es a la inversa. Primero se finalizaría la Unit3, después Unit2 y después Unit1. En general el proceso de inicio es simétrico : las units que se han inicializado primero tambien son las últimas a finalizarse.

En el caso de que una unit que emplemos necesite otra, entonces el criterio del compilador es inicializar todas las units que se encuentre *más afuera* primero y después las de *más adentro*. Por ejemplo, si en uses sólo hubiera Unit1 pero ésta empleara Unit2 y ésta última Unit3 entonces primero se inicializaría Unit3, después Unit2 y finalmente Unit1. El proceso de finalización seria el inverso, tal como se ha comentado.

A diferencia de C y de algunos lenguajes, incluir dos veces una misma unit en el código (por ejemplo que Unit1 y Unit2 empleen Unit3) no comporta ningún problema y la inicialización y finalización sólo se lleva a cabo una vez siguiendo el orden antes indicado.

11.8.Ámbito de una unit

Se llama ámbito a aquel conjunto de partes del código que tienen acceso a un identificador, ya sea una constante, variable, tipo o función/procedimiento.

11.8.1.Interface

Si el identificador se declara en la sección interface entonces el ámbito de este será :

- Toda la propia unit del identificador incluyendo las secciones interface, implementation, initialization y finalization.
- Todas las units y programas que tengan esta unit dentro de la cláusula uses.

11.8.2.Implementation

Si el identificador se declara en la sección implementation entonces su ámbito es :

- Toda la propia unit a excepción de la sección interface. Esto quiere decir, por ejemplo, que no podremos emplear tipos de datos declarados en implementation dentro de los parámetros de una función en interface, puesto que no sería posible para el programador pasar parámetros a esta función fuera de la propia unit pues no conocería el tipo de dato.
- Fuera de la unit estos identificadores no son accesibles.

11.8.3.Incluir units dentro de interface

Es posible especificar una cláusula uses dentro de la sección interface. Esta posibilidad sólo se tendría que emplear en caso de que algun identificador declarado en interface precise de algun otro tipo declarado en otra unit.

Situando la cláusula uses en interface permitimos que toda la unit, incluidas implementation, initialization y finalization, acceda a esta otra unit.

Ahora bien, esta posibilidad tiene el riesgo de que nuestra unit tambien se encuentre en la sección interface de la otra unit lo que resultaría en una referencia circular que es ilegal.

11.8.4.Incluir units dentro de implementation

En este caso sólo las secciones implementation, initialization, finalization tienen acceso a los identificadores delcarados en la unit en uses. Es posible que dos units se incluyan mutuamente siempre que no se incluyan en interface a la vez, ya sea las dos en implementation o una de ellas en interface y la otra en implementation. Siempre que sea posible, es recomendable incluir las units en la cláusula de implementation.

12.Librerías

12.1.Librerías vs units

Hasta ahora hemos visto que todos los elementos que conformaban nuestro programa se resolvían en tiempo de compilación. Es lo que se llama enlace estático. Una vez se ha compilado el programa, el propio EXE dispone de todo lo necesario para ejecutarse.

En el caso de las units escritas compiladas con FreePascal, la compilación nos dará un archivo PPW que contiene la información que en tiempo de compilación se enlazará adecuadamente al EXE final.

Algunas veces, pero, el código no habrá sido generado por nosotros sino que con otros compiladores. Para poder emplear este código dentro de nuestro programa tendremos que enlazarlo. En algunos casos realizaremos enlaces estáticos, resueltos en tiempo de compilación, pero en otros tendremos que enlazar en tiempo de ejecución, especialmente en el caso de las librerías de enlace dinámico (DLL, Dynamic Library Linking) de Windows.

12.2.Archivos objeto

FreePascal permite el enlace estático con archivos llamados archivos objeto (que suelen llevar extensión `O`, o `OW` en Win32) que hayan sido compilados con los compiladores GNU Pascal, GNU C/C++ o algún otro compilador que compile este formato de archivos objeto. Los compiladores Borland, Microsoft e Intel generan un tipo de archivos objeto que en general no son directamente compatibles con los archivos objeto que vamos a emplear.

12.2.1.Importación de rutinas en archivos objeto

Supongamos que queremos enlazar una función muy sencilla escrita en C que tiene el código siguiente :

```
int incrementador(int a)
{
  return a+1;
}
```

Esta función toma como parámetro un entero de 32 bits (el parámetro a) y devuelve otro entero que es el valor del parámetro incrementado en uno. En Pascal esta función sería la siguiente :

```
function incrementador(a : integer) : integer;
begin
  incrementador := a+1;
end;
```

Como se puede ver C y Pascal tienen un cierto parecido sintáctico.

Supongamos que la rutina en C se encuentra almacenada en un archivo llamado incr.c. Ahora hay que compilarlo para obtener un archivo objeto. Como que no es un programa completo, pues no tiene bloque principal, indicaremos al compilador que sólo lo compile y no llame al enlazador, que nos daría un error. El compilador de C que emplearemos en los ejemplos en C es el conocido GNU CC, o `gcc`. Para compilar nuestro código ejecutamos el compilador desde la línea de órdenes habitual :

```
gcc -c -o incr.o incr.c
```

El parámetro `-c` indica que sólo queremos compilación y no enlazado. El parámetro `-o` seguido de `incr.o` sirve para indicar que el archivo objeto que se creará llevará el nombre de `incr.o`. Finalmente `incr.c` es el archivo que estamos compilando.

Una vez compilada la rutina en un archivo objeto ahora hay que enlazarla en nuestro programa Pascal. Para hacerlo tendremos que realizar varios pasos.

Primero hay que indicar al compilador que habrá que enlazarse con el archivo objeto incr.o. Para hacerlo emplearemos la directiva de compilador $LINK seguida del nombre de archivo objeto a incluir. Aunque no es necesario incluir la extensión .o (que es la extensión por defecto) es recomendable.

```
{$LINK incr.o}
```

Una vez hemos incluido esta directiva en el principio del programa ahora habrá que declarar una función externa. Las funciones externas se declaran con la directiva `external` que indica al compilador que esta función se encuentra en un módulo externo al programa y que no la estamos implementando nosotros. En nuestro caso, además, tendremos que añadir la directiva `cdecl` pues esta rutina está escrita en C y conviene que el compilador pase los parámetros correctamente y además encuentre el nombre de la función dentro del archivo objeto. El archivo objeto no contiene los nombres de las funciones directamente sino que éste sufre modificaciones llamadas *mangling*. La directiva cdecl indica a FreePascal que busque el *mangled name* apropiado en C de nuestra función en el archivo objeto.

Tambien hay que tener en cuenta que en este caso la declaración de la función tendrá en cuenta las mayúsculas y las minúsculas. En nuestro caso la función incrementador en C estaba toda en minúsculas y en el código Pascal vamos a tener que hacer lo mismo. Finalmente hay que emplear el mismo tipo de parámetro. En Pascal el tipo `int` de C es equivalente a `Longint`.

```
function incrementador(a : longint) : longint; cdecl; external;
```

Tal como hemos indicado, las siguientes posibles declaraciones serían sintácticamente correctas pero el enlazador no sabría resolver a qué función estamos llamando.

```
function INCREMENTADOR(a : longint) : longint; cdecl; external;
function incremENTADOR(a : longint) : longint; cdecl; external;
```

Una vez la función ha sido declarada ya la podemos emplear dentro de nuestro código ahora ya sin restricción de mayúsculas. Un programa de ejemplo de nuestro código sería el siguiente.

```
program FuncionC;

{$L INCR.O}
function incrementador(a : longint) : longint; cdecl; external;

var
  numero : longint;
begin
  write('Introduzca un número : '); Readln(numero);
  Write('Número + 1 : ', Incrementador(numero));
end.
```

En realidad la mayoría de llamadas a funciones externas suelen ser más complejas pues pueden incorporar variables por referencia. En el ejemplo siguiente emplearemos un procedimiento que incrementa la variable que le pasemos. El código en C es el siguiente :

```
void incrementador(int* a)
{
  *a = *a+1;
}
```

En realidad lo que estamos pasando es un puntero tal como indica el operador de desreferenciación de C *, similar al operador ^ de Pascal. Lo que hacemos es incrementar el contenido del puntero en una unidad. Esto es así porque C no incorpora parámetros por referencia pero permite pasar punteros como parámetros. La directiva void indica que esta función no devuelve nada por lo que es un procedimiento. Una vez compilado y obtenido el archivo objeto nuestro programa puede quedar así :

```
program FuncionC;

{$L INCR.O}

procedure incrementador(var a : Longint); cdecl; external;
```

```
var
  numero : Longint;
begin
  Write('Introduzca un número : '); Readln(numero);
  Incrementador(numero);
  Writeln('Número + 1 : ', numero);
end.
```

En realidad cuando pasamos un parámetro por referencia lo que estamos pasando es su puntero por tanto esta declaración así es del todo correcta. Obsérvese que en este caso hemos declarado un procedimiento y no una función ya que, como sabemos, nuestro código no devuelve nada.

Aunque C no incorpora parámetros por referencia, C++ sí. Vamos a sustituir el código incr.c por un código en C++ que reciba los parámetros por referencia.

```
void incrementador(int &a)
{
  a = a+1;
}
```

El operador & de C++ indica que el parámetro es por referencia. Como podemos ver ahora ya no hay que emplear el operador * ya que el parámetro se trata como una variable normal. Para compilar el código en C++ hay que emplear el compilador g++ en vez de gcc con los mismos parámetros que hemos empleado antes. Una vez obtenido el archivo objeto si intentamos compilar el archivo obtendremos un error. Esto es debido a que el mangling de C++ es muy distinto al de C. Por tanto tendremos que indicarle al compilador de C++ explicitamente que haga mangling al estilo de C. Para hacerlo habrá que modificar ligeramente la función :

```
extern "C" void incrementador(int &a)
{
  a = a+1;
}
```

Volvemos a compilar el código C++ para obtener el archivo objeto. Ahora al compilar el programa en Pascal todo funciona correctamente. En realidad si hubieramos compilado con el mangling de C++ tendríamos que haber llamado la función `__incrementador_FRi`.

Podemos indicar al compilador cual es el nombre real de la función que estamos importando. Para hacerlo sólo hay que añadir la directiva `name` después de external y el nombre, sensible a maýusculas, de la función. Por ejemplo si compilamos el primer ejemplo de código en C++ tendremos

que llamar la función por el nombre `_incrementador__FRi`. La declaración de la función quedaría de la forma siguiente :

```
procedure Incrementador(var a : Longint); cdecl; external name '_incrementador__FRi';
```

En este caso el nombre que demos a la función no se tiene en cuenta ya que se tiene en cuenta el valor que indica `name`. Podíamos haberla declarado como INCREMENTADOR y todo hubiera funcionado igual. Este método lo podemos emplear siempre para ignorar el *mangling* por defecto de la convención de llamada.

12.2.2.Exportación de funciones en archivos objeto

Cuando compilamos un archivo, ya sea una unit o un programa, con FreePascal siempre se obtiene un archivo objeto. Este archivo contiene las declaraciones del programa, variables, etc una vez se han compilado. Para poder exportar funciones en este archivo objeto tendremos que emplear una unit, ya que los programas definen símbolos que nos pueden conllevar problemas a la hora de enlazar con otros programas.

Las funciones que queramos exportar tendrán que llevar la directiva `export` para indicar al compilador que esta función va a ser exportada, en caso contrario no se podría acceder a ella. Para evitar problemas con el mangling de las funciones es recomendable declarar las funciones que exportemos como cdecl o bien definir un nombre o alias para la función :

```
unit PruebaObj;

interface

implementation

procedure incrementar(var a : integer); export; stdcall; [Alias : 'increm' ];
begin
  a := a + 1;
end;

end.
```

Cuando referenciemos a esta función desde codigo C, por ejemplo, podremos emplear el nombre increm, sensible a mayúsculas, que hemos declarado en la estructura del alias.

Hay que tener en cuenta de que las funciones y procedimientos exportados no deberían ser llamados por otras funciones puesto que emplean un mecanismo diferente para el paso de parámetros. Aunque es posible llamar funciones no exportadas dentro de funciones exportadas sin ningún problema.

Para impedir que el programador llame a estas funciones se omite la declaración en interface y sólo se implementa dentro del bloque `implementation`. De esta forma no es posible llamar a la función, aun cuando esta sea incluida en un programa o unit.

Para llamar a una función exportada hay que emplear el mecanismo que hemos visto antes para importar funciones de archivos de objeto. Cada lenguaje de programación tiene sus mecanismos para importar archivos objeto, si es que tiene.

Hay que tener en cuenta que si una función exportada llama funciones que se encuentran en otras units tambien necesitaremos sus archivos objeto. Por ejemplo la función Writeln se encuentra definida en la unit System y necesitaremos el archivo System.o (o bien, System.ow). Esta unit no es necesario añadirla en la cláusula uses ya que se añade siempre automáticamente por el compilador.

Hay otras restricciones por lo que a las funciones exportadas concierne. Por ejemplo los ShortString (ni String[n]) no se exportan correctamente y es recomendable emplear PChar o bien, si es para emplearse en programas y units compilados con FreePascal, el tipo AnsiString que sí que se exporta correctamente. Finalmente hay que tener en cuenta de que los nombres de los tipos de datos, como los enteros que hemos visto antes, pueden variar de un lenguaje a otro.

12.2.3.Exportar e importar variables

De la misma forma que podemos importar y exportar funciones y procedimientos tambien podemos exportar e importar variables. Para exportar una variable emplearemos de nuevo una unit. A diferencia del caso anterior no hay problema en acceder a ella por lo que podremos incluirla en la sección interface de una unit sin ningún problema.

Es muy recomendable que el nombre de la variable se guarde en el archivo objeto con mangling de C. Para hacerlo añadiremos la directiva `cvar` después de la declaración de una única variable. No es posible emplear un mismo `cvar` para varias variables pero sí varios `cvar` para cada variable declarada. En este caso la declaración será sensible a las mayúsculas aunque en el código en Pascal podremos referenciarla como queramos. La unit que emplearemos de ejemplo es la siguiente :

```
unit ExportarVariable;

interface

implementation

var valor : Longint; cvar;

procedure Cuadrado(a : Longint); cdecl; export; // Exportamos la función
begin
  Valor := a*a;
end;
```

```
end.
```

Para importar una variable tenemos dos formas. La primera se basa en aprovecharse del método de mangling de C. Para hacerlo habrá que añadir la directiva `cvar` seguida de la directiva `external`, que indica al compilador que no reserve memoria para esta variable pues es externa. En este caso el nombre de declaración es sensible a las mayúsculas aunque desde el código podremos referirnos a ella sin distinción de mayúsculas.

La otra forma de declarar una variable externa permite emplear el identificador que queramos pero implica conocer el *mangled name* de la variable. En C las variables llevan un guión bajo delante del identificador. De esta forma podemos declarar la variable externa de nuestro archivo objeto.

```
{$LINK EXPORTARVARIABLE.OW}
var
  valor : Longint; cvar; external; // Valor, VALOR u otros no son válidos
  alias : Longint; external name '_valor';
```

Como se observa, las variables valor y alias se referirán a la misma variable de forma que cualquier cambio en esta se apreciará en el valor de las dos variables y viceversa, las asignacions a cualquiera de ellas dos afectarán al valor de la otra. Finalmente vamos a importar la función `Cuadrado` que hemos exportado en la unit.

```
procedure Cuadrado(a : Longint); cdecl; external; // quadrat, o QUADRAT no valen
```

El programa completo es el siguiente :

```
program ImportarFuncionesyVariables;

{$L EXPORTARVARIABLE.OW}

procedure Cuadrado(a : Longint); cdecl; external;

var
  valor : Longint; cvar; external;
  alias : Longint; external name '_valor';

begin
  Cuadrado(6);
  Writeln(valor, ' = ', alias);
end.
```

12.3.Librerías de enlace dinámico

Hasta ahora hemos visto mecanismos de enlace estático con funciones y procedimientos y variables. La existencia de la función y su posterior enlace con el código se ha resuelto todo el rato antes de ejecutar el programa, en tiempo de compilación.

Las librerías de enlace dinámico (con extensión DLL en Windows y so en Linux) permiten realizar el enlace en tiempo de ejecución.

El enlace dinámico tiene varias ventajas. El archivo, al no ir enlazado estáticamente, puede ser sustituido por otro que corrija errores o mejoras en el algoritmo, siempre que se conserven los nombres y los parámetros de las funciones. De esta forma se puede actualizar la librería. En el caso del enlace estático habríamos tenido que recompilar el programa para actualizarlo. De esta forma, el programador la mayoría de veces sólo tiene que preocuparse en cuánto y cómo llamar a la función más que no como ha sido implementada, pues los cambios en la implementación no implican recompilar el programa de nuevo.

12.3.1.Creación de una librería de enlace dinámico

La creación de una librería de enlace dinámico es parecida a un programa. Para empezar es necesario que la librería empiece con la palabra reservada `library` que indica al compilador que lo que se encontrará responde a la forma de una librería. Hay que acompañar a la palabra reservada de un identificador que no es necesario que tenga el mismo nombre que el archivo.

Las funciones y procedimientos se declararán de la forma habitual, sin `export`, y especificando la convención de llamada. Es muy habitual en Windows la convención `stdcall` para funciones en DLL por lo que se recomienda esta convención.

Para exportar las funciones emplearemos la cláusula `exports` (con ese final, es importante) seguida de los identificadores de función o procedimiento que queramos exportar y, opcionalmente pero más que recomendable, el nombre de exportación de la función. Este será el nombre con el que llamaremos después la función o procedimiento de la librería. En caso contrario, si no lo especificamos se empleará el mangling por defecto de FreePascal, que en el caso concreto de las DLL es todo el nombre de la función toda en mayúsculas.

```
library CuadradosyCubos;

function Cuadrado(a : longint) : longint; stdcall;
begin
  Cuadrado := a*a;
end;
```

```
function Cubo(a : longint) : longint; stdcall;
begin
  Cubo := a**3;
end;

exports
  Cuadrado name 'Cuadrado',
  Cubo name 'Cubo';

end.
```

En este ejemplo estamos exportando dos funciones Cuadrado y Cubo. Una vez compilada la librería obtendremos un archivo `.DLL` que ya podrá ser llamado desde nuestro programa.

Las funciones que exporte la librería no tienen porqué estar declaradas forzosamente dentro de la librería. Pueden estar declaradas en la interfaz de una unit que se encuentre dentro de la cláusula `uses` después de la cabecera `library`.

12.3.2.Importación de funciones en librerías de enlace dinámico

La forma de importar las funciones y los procedimientos es parecida. Ahora no es necesario especificar ninguna directiva como $LINK y el compilador tampoco detectará si la función es incorrecta. Por este motivo es importante especificar los parámetros correctamente conjuntamente con la convención de llamada adecuada.

En este caso la directiva `external` tiene que ir seguida del nombre de la librería y puede ir seguida de una directiva `name` que especifique el nombre. En caso contrario se empleará el *mangling* habitual.

```
program FuncionesDLL;

const
 NOMBREDLL = 'CuadradosyCubos';

function Cubo(a : longint) : longint; stdcall; external NOMBREDLL name 'Cubo';
function Cuadrado(a : longint) : longint; stdcall; external NOMBREDLL name 'Cuadrado';

var
  a : integer;
begin
  Write('Introduzca un número : '); Readln(a);
  Writeln('Cuadrado : ', Cuadrado(a), ' Cubo : ', Cubo(a));
end.
```

Como se puede ver, es posible y muy recomendable, emplear constantes cuando nos referimos a una misma librería a fin de evitar errores tipográficos. En este ejemplo hemos supuesto que la librería

era CUADRADOSYCUBOS.DLL. Tambien podemos observar que no importa el orden en el que se importan las funciones. En este caso no hemos especificado la extensión en NOMBREDLL, que por defecto en Win32 es .DLL, pero es posible especificarla sin ningun problema.

Este programa sólo funcionará si es capaz de encontrar la librería CUADRADOSYCUBOS.DLL en alguno de estos directorios :

- El directorio donde se encuentra el archivo ejecutable.
- El directorio actual del sistema de archivos.
- El directorio de Windows. Habitualmente C:\WINDOWS
- El subdirectorio SYSTEM de Windows. Habitualmente C:\WINDOWS\SYSTEM
- Los directorios de la variable PATH.

En caso de no existir o que no se encontrará la librería en alguno de estos directorios, Windows nos mostraría un error y no podríamos ejecutar el programa. Igualmente pasaría si la función que importemos de una DLL no existiera. En este aspecto es importante remarcar el uso de la directiva name para importar correctamente las funciones.

12.3.3.Importación y exportación de variables en DLL

Es posible en FreePascal, y Delphi, exportar variables en librerías DLL aunque sólo el primero permite importarlas mediante la sintaxis. El método para hacerlo es muy similar a exportar e importar funciones. Por ejemplo, la librería siguiente exporta una función que inicializará la variable exportada.

```
library VarExport;
var
  variable_exportada : integer;

procedure Inicializar;
begin
 variable_exportada := 10;
end;

exports
 Inicializar name 'Inicializar',
 variable_exportada name 'variable_exportada';
end.
```

El programa siguiente importa la función y la variable. Supongamos que la DLL recibe el nombre VarExport.DLL.

```
program ImportarVar;
const
```

```
  VarExportDLL = 'VarExport';
var
  variable : integer; external VarExportDLL name 'variable_exportada';

procedure Inicializar; external VarExportDLL name 'Inicializar';

begin
  Inicializar;
  Writeln(variable); {10}
end.
```

12.3.4.Importación dinámica de funciones en librerías DLL (sólo Win32)

Mediante la API de Win32, interfaz de programación de aplicaciones, podemos importar funciones de librerías de forma programática. De esta forma podemos controlar si la función existe o si la librería se encuentra en el sistema y dar la respuesta adecuada ante estas situaciones.

Tendremos que emplear tres funciones y varias variables. Por suerte FreePascal incorpora la unit Windows donde están declaradas la mayor parte de tipos de datos e importadas la mayor parte de funciones de la Win32 API.

La función LoadLibrary, que recibe como parámetro el nombre de la librería en una cadena terminada en nulo, devolverá cero si esta librería no existe. En caso contrario devuelve un valor distinto de cero.

La función GetProcAddress nos devolverá un puntero nil si la función que importamos no existe. En caso contrario nos dará un puntero que podremos enlazar en una variable de tipo función a fin de poderla ejecutar. Finalmente la función FreeLibrary libera la memoria y descarga la librería si es necesario.

Vamos a ver un ejemplo con la función Cuadrado de la primera librería de ejemplo.

```
program FuncionDLLDinamica;
uses Windows;
const
  NOMBREDLL = 'CuadradosYCubos.dll';
  NOMBREFUNCION = 'Cuadrado';

type
  TCuadrado = function (a : longint) : longint; stdcall;

var
  a : integer;
  Cuadrado : TCuadrado;
  Instancia : HMODULE; // Tipo de la unit Windows
begin
  Write('Introduzca un numero : '); Readln(a);
  // Intentaremos importar la función
  Instancia := LoadLibrary(PCHar(NOMBREDLL));
  if Instancia <> 0 then
```

```
  begin
    // Hay que realizar un typecasting correctamente del puntero
    Cuadrado := TCuadrado(GetProcAddress(Instancia, NOMBREFUNCION));
    if @Cuadrado <> nil then
    begin
      Writeln('Cuadrado : ', Cuadrado(a));
    end
    else
    begin
      Writeln('ERROR - No se ha encontrado la función en ', NOMBREDLL);
    end;
    // Liberamos la librería
    FreeLibrary(Instancia);
  end
  else
  begin
    Writeln('ERROR - La librería ', NOMBREDLL,' no se ha encontrado');
  end;
end.
```

Puede parecer un código complejo pero no hay nada que no se haya visto antes. Comentar sólo que hay que hacer un amoldado del puntero que se obtiene con GetProcAddress antes de asignarlo correctamente al tipo función. Como nota curiosa se puede cambiar 'Cuadrado' de la constante NOMBREFUNCION por el valor 'Cubo' y todo funcionaría igual ya que Cubo y Quadrado tienen la misma definición en parámetros y convención de llamada. Sólo que en vez de obtener el cuadrado obtendríamos el cubo del número.

12.3.4.Llamada a funciones del API de Win32

Aunque la mayoría de funciones de la API se encuentran importadas en la unit Windows de vez en cuando tendremos que llamar alguna que no esté importada. El método es idéntico para cualquier DLL y sólo hay que tener en cuenta de que la convención de llamada siempre es `stdcall`.

13.Programación orientada a objetos

13.1.Programación procedimental vs programación orientada a objetos

Hasta ahora hemos visto un tipo de programación que podríamos llamar procedimental y que consiste en reducir los problemas a trozos más pequeños, funciones y procedimientos, y si es necesario agrupar estos trozos más pequeños con elementos en común en módulos, units y librerías.

Este modelo de programación, que parece muy intuitivo y necesario para programadores que no han visto la programación orientada a objetos (POO de ahora en adelante) tiene varios inconvenientes.

Para empezar, se basa en un modelo demasiado distinto a la forma humana de resolver los problemas. El ser humano percibe las cosas como elementos que suelen pertenecer a uno o más conjuntos de otros elementos y aplicalos conociemientos que tiene de estos conjuntos sobre cada elemento. Así, es evidente de que un ratón y un elefante son seres vivos y como seres vivos ambos nacen, crecen, se reproducen y mueren. En otro ejemplo, un alambre no es un ser vivo pero sabemos que es metálico y como elemento metálico sabemos que conduce bien la electricidad y el calor, etc. Esta asociación de ideas a conjuntos no es fácil de implementar en la programación procedimental ya que si bien un ser vivo es algo más amplio que el concepto de elefante no es posible implementar (siempre hablando de implementación de forma eficiente, claro) un sistema que dado un ser vivo cualquiera pueda simular el naciemiento, crecimiento, etc. básicamente porque cada ser vivo lo hace a su manera.

Llegados a aquí, se empieza a entrever más o menos el concepto de objeto. Es algo que pertenecerá a un conjunto y que al pertenecer en él poseerá sus cualidades a la vez que puede tener sus propias cualidades o adaptar las que ya tiene.

En qué se parecen una moto y un camión ? Bien, almenos ambos son vehículos y tienen ruedas. En qué se distinguen ? La moto sólo tiene dos ruedas y sólo puede llevar un tripulante (en el peor de los casos) mientras que el camión tiene cuatro ruedas y además dispone de un compartimiento para llevar materiales. Qué pasa si cogemos una moto y le añadimos una tercera rueda y un pequeño lugar para un segundo tripulante ? Pues que la moto se ha convertido en un *sidecar*. En qué se distinguen la moto y el sidecar ? Pues básicamente en sólo este añadido. En qué se parecen ? En todo lo demás. Por tanto no es exagerado decir que un sidecar *hereda* las propiedades de una moto y además añade nuevas cualidades, como el nuevo compartimiento.

Ya hemos visto la mayor parte de las propiedades de un objeto de forma bastante intuitiva que definen los objetos y que veremos más adelanta. Estas tres propiedades tienen nombre y son : la encapsulación, la herencia y el polimorfismo.

13.2.Encapsulación, herencia y polimorfismo

13.2.1.Encapsulación

Quizás ahora no queda muy claro qué quiere decir encapsulación. Básicamente consisten en que todas las caracteristicas y cualidades de un objeto están siempre definidas dentro de un objeto pero nunca fuera de los objetos. Es el mismo objeto quien se encarga de definir sus propiedades y en su turno las implementa. Siempre dentro del contexto de objeto que veremos.

13.2.2.Herencia

Hablamos de herencia cuando un objeto adquiere todas las características de otro. Esta característica permite construir jerarquias de objetos donde un segundo objeto hereda propiedades de otro y un tercero de este segundo de forma que el tercero tambien tiene propiedades del primero.

En el ejemplo anterior una moto es [*heredera de*] un vehiculo y un sidecar es [*heredera de*] una moto. Las propiedades que definen a un vehículo tambien las encontraremos en un sidecar. En cambio a la inversa no siempre es cierto.

13.2.3.Polimorfismo

Esta palabra sólo signifca que dado un objeto antepasado, o ancestro, si una acción es posible llevarla a cabo en este objeto (que se encontrará en la parte superior de la jerarquia de objetos) tambien se podrá llevar a cabo con sus objetos hijos pues la heredan. Pero cada objeto jerárquicamente inferior puede (que no tiene por qué) implementar esta acción a su manera.

Volviendo al ejemplo de los seres vivos : todos se reproducen. Algunos de forma asexual, dividiendose o por gemación. De otros de forma sexual, con fecundación interna o externa, etc.

13.2.4.Resumiendo

Ahora que hemos visto más o menos cuales son las propiedades de los objetos podemos llegar a la conclusión que una de las ventajas directas de la POO es la reutilización del código y su reusabilidad es mayor.

Supongamos que tenemos un objeto que, por el motivo que sea, está anticuado o le falta algún tipo de opción. Entonces podemos hacer uno nuevo que herede de éste y que modifique lo que sea necesario, dejando intacto lo que no se tenga que retocar. El esfuerzo que habremos tenido que hacer es mucho menor al que hubieramos tenido que hacer para reescribir todo el código.

13.3.Clases y objetos

De las distintas formas de aproximación a la POO que los lenguajes de programación han ido implementado a lo largo del tiempo, en Pascal encontraremos dos formas parecidas pero distintas en concepto : los objetos y las clases.

En este manual emplearemos las clases sin ver los objetos pues es una sintaxis y concepción mucho más parecida a la forma de entender la POO de C++ que se basa en clases.

Un objeto es la unión de un conjunto de métodos, funciones y procedimientos, y campos, las variables, que son capaces de heredar y extender los métodos de forma polimórfica. Para emplear un objeto necesitamos una variable de este objeto. Sólo si tiene métodos virtuales, que ya veremos que son, es necesario inicializar y destruir el objeto.

Las clases son como objetos que, a diferencia, no se pueden emplear directamente en una variable sinó que la variable tiene como finalidad recoger una instancia, una copia usable para entendernos, de esta clase. Esta copia la devuelve el constructor y recibe el nombre de objeto ya que es el elemento real mientras que la clase es un elemento formal del lenguaje. Posteriormente habrá que liberar el objeto. Es necesario siempre obtener una instancia de la clase u objeto, en caso contrario no es posible trabajar con él. De ahora en adelante cuando hablemos de objetos estaremos hablando de instancias de clases y no del otro modelo de aproximación a la POO.

13.4.Las primeras clases

Antes de poder trabajar con la POO mediante clases tenemos que avisar al compilador de que emplaremos clases. Para hacerlo hay que emplear la directiva de compilador `{$MODE OBJFPC}`[5].

Implementaremos una clase muy simple que permita realizar operaciones simples a partir de dos variables de la clase. La clase se llamara TOperaciones. Las clases hay que definirlas como si fueran tipos de datos ya que, habitualmente, con lo que se trabaja es una instancia de una clase, un objeto, no con la clase en sí.

Para declarar una clase hay que emplear la palabra reservada class y una estructura algo parecida a un record. Hay que especificar tambien de quien es heredera esta clase. En Pascal todas las clases tienen que heredar de alguna otra y la clase superior a todas se llama TObject. Esta es la declaración.

```
{$MODE OBJFPC}
type
```

[5]Esto provoca algunos cambios en el comportamiento del compilador. El más importante está en la declaración del tipo Integer. En los modos OBJFPC y DELPHI el tipo Integer es lo mismo que Longint, o sea un entero de 32-bits con signo. En los modos TP y FPC es un entero sin signo de 16-bits.

```
TOperaciones = class ( TObject )
```

Dentro de esta declaración primero declararemos los campos, las variables de la clase. En nuestro caso declararemos los operandos de las operaciones binarias (operaciones de dos operandos) y les daremos los nombres a y b. La declaración se hace de la misma forma que cuando declaramos variables normalmente. Las declararemos de tipo entero. Tambien declararemos una variable, Resultado, en la cual guardaremos el valor de las operaciones.

Ahora hay que declarar algún método (procedimientos y funciones de la clase). En nuestro ejemplo implementaremos la operación suma en un procedimiento llamado Suma que sumará a y b y almacenará el resultado en la variable Resultado. La interfaz de la clase quedará así.

```
type
  TOperaciones = class ( TObject )
    a, b, Resultado : integer;
    procedure Suma;
  end;
```

Obsérvese que Suma no necesita parámetros ya que trabajará con a y b. Una vez tenemos la interfaz habrá que implementar los métodos. Para hacerlo lo haremos como normalmente pero cuidando de que el nombre del método vaya precedido del nombre de la clase. La implementación de Suma sería la siguiente :

```
procedure TOperaciones.Suma;
begin
  Resultado := a + b;
end;
```

Como se ve, podemos trabajar con las variables de la clase dentro de la misma clase sin necesidad de redeclararlas. Esto es posible pues la clases tienen una referencia interna que recibe el nombre de Self. Self se refiere a la misma clase por lo que la siguiente definición es equivalente a la anterior :

```
procedure TOperaciones.Suma;
begin
  Self.Resultado := Self.a + Self.b;
end;
```

El identificador Self lo emplearemos cuando haya alguna ambigüidad respecto a las variables o métodos que estamos empleando. En general no suele ser necesario.

Para poder emplear una clase es necesario que la instanciemos en una variable del tipo de la clase. Para hacerlo necesitamos emplear el constructor de la clase, por defecto Create. Finalmente cuando ya no la necesitemos más hay que liberarla con el método Free. Estos métodos no los implementamos, de momento, pues ya vienen definidos en TObject. El código del bloque principal para emplear la clase es el siguiente :

```
var
  Operaciones : TOperaciones;
begin
  Operaciones := TOperaciones.Create; // Instancia de la clase

  Operaciones.a := 5;
  Operaciones.b := 12;
  Operaciones.Suma;
  Writeln(Operaciones.Resultado);

  Operaciones.Free; // Liberamos la instancia
end.
```

Por defecto el constructor Create no toma parámetros. Su resultado es una instancia de una clase que almacenamos en Operaciones, como si se tratara de una especie de puntero. Podemos acceder a los campos y métodos de la clase como si de un record se tratara. No hay que olvidar de liberar la instancia pues dejaríamos memoria reservada sin liberar lo cual podría afectar al rendimiento posterior del sistema. Téngase en cuenta que una vez liberada la instancia ya no es posible acceder a sus campos ni a sus métodos. Hacerlo resultaría en un error de ejecución de protección general de memoria.

13.5.Hacer clases más robustas

La clase que hemos diseñado anteriormente tiene unos cuantos problemas. Para empezar, el programador puede acceder a la variable Resultado y modificarla a su gusto. Esto podría falsear el resultado de la operación. Por otro lado, en una hipotética operación División el programador podría caer en la tentación de poner un divisor con valor cero lo que provocaría un error de ejecución al realizarse la operación.

Hay varias formas de implementar clases robustas que sean capaces de reaccionar delante de los errores del programador y del usuario. Las clases tienen varios mecanismos para proteger al programador de sus propios errores : los ámbitos de clase y las propiedades.

13.5.1.Ámbitos de la clase o grados de visibilidad

Las clases definen varios grados de visibilidad. Estos grados de visibilidad regulan en cierta forma el acceso a los elementos del objeto, campos y métodos, desde otros ámbitos del código. Los grados de visibilidad fundamentales son `private` y `public`.

Los métodos y campos con visibilidad private sólo son accesibles dentro de la misma clase, o sea, en los métodos que esta clase implementa. Los elementos public son accesibles desde cualquier lugar desde el cual se tenga acceso a la instancia de la clase.

Después hay un grado intermedio que recibe el nombre de `protected`. A diferencia de private, protected, permite el acceso a los miembros de la clase pero sólo en las clases que hereden de la clase, aspecto que en el grado private no es posible. Como veremos más adelante, las clases inferiores pueden modificar los grados de visibilidad de elementos a los cuales tienen acceso en herencia, protected y public.

En nuestra clase protegeremos la variable Resultado y la estableceremos private. La resta de clase será public. Los atributos se fijan directamente en la declaración de la clase :

```
type
  TOperaciones = class ( TObject )
   public
    a, b : integer; // Esto es publico
    procedure Suma; // Esto tambien
   private
    Resultado : integer; // Esto en cambio es privado
  end;
```

Ahora, tenemos un inconveniente, no es posible acceder a la variable Resultado fuera de la clase. Una solución puede pasar por definir una función que devuelva el valor de Resultado. Una solución más elegante es emplear propiedades.

13.5.2.Propiedades de una clase

Las propiedades son una aproximación a la POO muy eficaz que fueron introducidas en Delphi. Son unos miembros especiales de las clases que tienen un comportamiento parecido a un campo, o sea una variable, pero además es posible establecer la posibilidad de lectura y/o escritura. Lo más interesante de las propiedades es la posiblidad de asociar un método con las operaciones de lectura y escritura con lo cual podemos escribir clases eficientes y robustas.

Las propiedades se definen de la forma siguiente :

```
property NombrePropiedad : tipoDato read Variable/Metodo write Variable/Metodo;
```

Téngase en cuenta de que las propiedades sólo se pueden declarar dentro de una clase. Si omitmos la parte write tendremos una propiedad de sólo lectura mientras que si omtimis la parte de read tendremos una propiedad de sólo escritura. Este caso es técnicamente posible pero es poco útil, en general.

Para el ejemplo anterior renombraremos la variable Resultado a FResultado con lo cual tendremos que modificar el método Suma donde habrá que cambiar Resultado por FResultado.

```
type
  TOperaciones = class ( TObject )
   private
    FResultado : integer;
   public
    a, b : integer;
    procedure Suma;
  end;

procedure TOperaciones.Suma;
begin
  FResultado := a + b;
end;
```

Aunque el orden es indiferente, es habitual poner los atributos de visibilidad en el orden : private, protected y public. Esto es así porque en miembros públicos se emplean propiedades privadas per nunca al revés. Declaramos la propiedad pública Resultado de sólo lectura sobre la variable FResultado.

```
property Resultado : Integer read FResultado;
```

Cuando leamos el valor de Resultado lo que estamos leyendo es el valor FResultado. Si intentamos modificar su valor el compilador nos advertirá de que no es posible pues no hemos especificado esta posiblidad. La definición de la clase quedaría tal como sigue :

```
type
  TOperaciones = class ( TObject )
   private
    FResultado : integer;
   public
    a, b : integer;
    property Resultado : integer read FResultado;
    procedure Suma;
  end;
```

Tal como hemos visto antes en la definición de la declaración de propiedades es posible indicar que la lectura y escritura se realice mediante un método. Si el método es de lectura emplearemos una

función del tipo de la propiedad. Si el método es de escritura emplearemos un procedimiento con un único parámetro del tipo de la propiedad.

El programa siguiente es un ejemplo inútil del uso de las propiedades :

```
program EjemploPropiedades;
{$MODE OBJFPC}
type
  TEjemplo = class (TObject)
   private
    function LeerPropiedad : Integer;
    procedure EscribirPropiedad(Valor : Integer);
   public
    property Propiedad : Integer read LeerPropiedad write EscribirPropiedad;
  end;

function TEjemplo.LeerPropiedad : Integer;
var
  i : integer;
begin
  Randomize; // Para que los numeros sean aleatorios
  i := Random(10);
  Writeln('Leyendo la propiedad. Devolviendo un ', i);
  LeerPropiedad := i;
end;

procedure TEjemplo.EscribirPropiedad(Valor : Integer);
begin
  Writeln('Escribiendo la propiedad. Ha asignado el valor ', Valor);
end;

var
  Ejemplo : TEjemplo;

begin
  Ejemplo := TEjemplo.Create;
  Writeln('Devuelto : ', Ejemple.Propiedad); // Leemos -> LeerPropiedad
  Ejemplo.Propiedad := 15; // Escribimos -> EscribirPropiedad
  Ejemplo.Free;
end.
```

Una posible salida del programa es la siguiente :

```
Leyendo la propiedad. Devolviendo un 6
Devuelto : 6
Escribiendo la propiedad. Ha asignado el valor 15
```

Obsérvese que al leer la propiedad hemos llamado al método LeerPropiedad que ha escrito la primera línea. Una vez se ha devuelto un valor aleatorio Writeln escribe el valor de la propiedad. Después cuando asignamos la propiedad se ejecuta el método EscribirPropiedad.

Hay que observar que las referencias a métodos en las propiedades se hacen igual que las referencias en variables. Tambien es posible combinar lectura de métodos con escritura de variables y viceversa sin ningún tipo de problema. De hecho la combinación mas usual suele ser la de una propiedad que lee una variable privada y que activa un método privado cuando se modificia.

Finalmente sólo hay que comentar que la declaración de métodos de escritura permite que se pase el parámetro como un parámetro constante. De forma que la declaración anterior de EscribirPropiedad podría ser así :

```
procedure EscribirPropiedad(const Valor : Integer);
```

La posterior implementación habría necesitado tambien la palabra reservada `const`.

La función Randomize inicializa los números aleatorios y la función Random(n) devuelve un entero aleatorio entre 0 y n-1.

13.5.3.Propiedades indexadas

Las propiedades indexadas son un tipo de propiedad que permite acceder y asignar datos a través de un índice como si de un array se tratara. A diferencia de los arrys usuales, el índice no tiene porque ser un entero, tambien es válido un real, cadenas, caracteres, etc.

Definir una propiedad indexada no es más complicado que definir una propiedad normal. Simplemente los métodos write y read añaden los parámetros del índice. Veamos el ejemplo siguiente:

```
type
 TUnaClasse = class (TObject)
  private
    procedure SetIndex(a, b : integer; c : Extended);
    function GetIndex(a, b : integer) : Extended;
  public
    property Index[a, b : integer] : Extended read GetIndex write SetIndex;
  end;
```

En este caso si accedemos a la propiedad Index[n, m] obtendremos un real de tipo Extended. Igualmente, las asignaciones a Index[n, m] tienen que ser de tipo Extended. También podíamos haber declarado un tipo string como índce de la propiedad.

```
type
 TUnaClasse = class (TObject)
  private
    procedure SetIndex(a : String; b : Integer);
    function GetIndex(a : String) : Integer;
  public
```

```
    property Index[a : String] : integer read SetIndex write SetIndex;
end;
```

Ahora sería legal la sentencia siguiente :

```
UnaClasse.Index['hola'] := 10;
```

13.5.4.Propiedades por defecto

Si al final de una propiedad indexada añadimos la directiva `default` entonces la propiedad se vuelve por defecto. Observese la declaración siguiente :

```
property Index[a : String] : integer read SetIndex write SetIndex; default;
```

Ahora sería legal la asignación siguiente :

```
UnaClasse['hola'] := 10;
```

Como se deduce sólo puede haber una propiedad indexada por defecto en cada clase y las clases descendientes no pueden modificarla.

14.Herencia y polimorfismo

14.1.Ventajas de la herencia en la POO

Los objetos, como hemos comentado antes, son capaces de heredar propiedades de otros objetos. Algunos lenguajes como el C++ permiten que una misma clase herede propiedades de uno o más objetos. La herencia múltiple no deja de tener sus problemas, como por ejemplo la duplicidad de identificadores entre clases. Es por este motivo que muchos lenguajes simplifican la herencia a una sola clase. O sea, las clases pueden heredar como máximo de otra clase. Esto provoca que las jerarquias de objetos sean totalmente jerárquicas sin estructuras interrelacionadas entre ellas. La aproximación a la POO que se hace en FreePascal se basa en la herencia simple por lo que una clase siempre es heredera de una sóla cada vez. Esto no impide que las clases tengan numerosos descendientes.

14.2.La clase TObject

Tal como hemos comentado antes, FreePascal implementa la herencia simple de forma que en algun momento llegaremos a la clase más superior. Esta clase que ya hemos visto se llama TObject. En FreePascal todas las clases descienden directa o indirectamente (ya sea por transitividad de herencia) de TObject.

La clase TObject define bastantes métodos pero los más interesantes son sin duda el constructor Create y el destructor Destroy. El destructor Destroy no se llama nunca directamente. En vez de ello se emplea otro método de TObject que es Free. Free libera la clase incluso si no ha sido asignada por lo que es algo más seguro que Destroy.

14.3.Diseñar clases heredadas

Una de las ventajas de diseñar clases heredadas es la de poder dar soluciones concretas derivadas de soluciones más genéricas o comunas. De esta forma nuestras clases se pueden especializar.

La tercera propiedad de los objetos, el polimorfismo, permite que bajo el nombre de un mismo método se ejecute el código adecuado en cada situación. Los métodos que incorporan esta posibilidad reciben el nombre de métodos dinámicos, en contraposición a los estáticos, o más habitualmente métodos virtuales.

Antes de emplear métodos virtuales habrá que ver ejemplos sencillos, a la par que poco útiles pero ilustrativos, del funcionamiento de la herencia.

14.3.1.Herencia de clases

Para poder heredar las características de una clase, métodos y propiedades, habrá que tener alguna clase de donde heredar. Con esta finalidad declararemos la clase TPoligono que se encargá teóricamente de pintar un polígono. De hecho no pintará nada, sólo escribirá un mensaje en la pantalla. Se supone que el programador no empleará nunca esta clase sino que sus derivados por lo que su único método PintarPoligono será protected. Hay que pasar un parámetro n indicando el número de lados del polígono. La clase es la siguiente :

```
type
  TPoligono = class (TObject)
   protected
     procedure PintarPoligono(n : integer);
  end;

procedure TPoligono.PintarPoligono(n : integer);
begin
  Writeln('Pintar polígono de ', n,' lados');
end;
```

Hemos supuesto que el programador no trabajará nunca con objetos de tipo TPoligono sino que sólo sus derivados. Por eso definimos dos clases nuevas derivadas de TPoligono : TTriangulo y TCuadrado.

```
type
  TTriangulo = class (TPoligon)
    public
      procedure PintarTriangulo;
  end;

  TCuadrado = class (TPoligon)
    public
      procedure PintarCuadrado;
  end;

procedure TTriangle.PintarTriangulo;
begin
  PintarPoligono(3);
end;

procedure TQuadrat.PintarCuadrado;
begin
  PintarPoligono(4);
end;
```

Tal como se ve, los métodos PintarTriangulo y PintarCuadrado no son nada más que casos más concretos de PintarPoligono. Supongamos ahora que definimos una clase llamada TPoligonoLleno que

además de pintar el polígon además lo rellena de color. Será posible reaprovechar el código de PintarPoligono de TPoligono. Lo que haremos es ampliar el método PintarPoligono en la clase TPoligonoLleno que derivará de TPoligono. La declaración e implementación es la que sigue :

```
type
  TPoligonoLleno = class (TPoligono)
    protected
      procedure PintarPoligono(n : integer);
  end;

procedure TPoligonoLleno.PintarPoligono(n : integer);
begin
  inherited PintarPoligono(n);
  Writeln('Llenando el poligono de color');
end;
```

Para llamar al método de la clase superior que hemos redefinido en la clase inferior hay que emplear la palabra reservada inherited. La sentencia **inherited** PintarPoligono(n) llama a TPoligono.PintarPoligono(n). De otra forma se entendería como una llamada recursiva y no es el caso.

Finalmente declararemos dos clases nuevas TTrianguloLleno y TCuadradoLleno que desciendad de TPoligonoLleno.

```
program EjemploHerencia;
{$MODE OBJFPC}
type
  TPoligono = class (TObject)
    protected
      procedure PintarPoligono(n : integer);
  end;

  TTriangulo = class (TPoligono)
    public
      procedure PintarTriangulo;
  end;

  TCuadrado = class (TPoligono)
    public
      procedure PintarCuadrado;
  end;

  TPoligonoLleno = class (TPoligono)
    protected
      procedure PintarPoligono(n : integer);
  end;

  TTrianguloLleno = class (TPoligonoLleno)
    public
```

```
      procedure PintarTriangulo;
  end;

  TCuadradoLleno = class (TPoligonLleno)
    public
      procedure PintarCuadrado;
  end;

// Mètodos de TPoligono
procedure TPoligono.PintarPoligono(n : integer);
begin
  Writeln('Pintar polígono de ', n,' lados');
end;

// Mètodes de TTriangulo
procedure TTriangulo.PintarTriangulo;
begin
  PintarPoligono(3);
end;

// Mètodes de TCuadrado
procedure TCuadrado.PintarCuadrado;
begin
  PintarPoligono(4);
end;

// Mètodes de TPoligonoLleno
procedure TPoligonoLleno.PintarPoligono(n : integer);
begin
  inherited PintarPoligono(n);
  Writeln('Llenando el polígono de color');
end;

// Mètodes de TTrianglePle
procedure TTrianguloLleno.PintarTriangulo;
begin
  PintarPoligono(3);
end;

// Mètodes de TQuadratPle
procedure TCuadradoLleno.PintarCuadrado;
begin
  PintarPoligono(4);
end;

var
  Triangulo : TTriangulo;
  TrianguloLleno : TTrianguloLleno;
  Cuadrado : TCuadrado;
  CuadradoLleno : TCuadradoLleno;

begin
  Triangulo := TTriangulo.Create;
  Triangulo.PintarTriangulo;
  Triangulo.Free;
  Writeln; // Una linea de separación
```

```
  TrianguloLleno := TTrianguloLleno.Create;
  TrianguloLleno.PintarTriangulo;
  TrianguloLleno.Free;
  Writeln;

  Cuadrado := TCuadrado.Create;
  Cuadrado.PintarCuadrado;
  Cuadrado.Free;
  Writeln;

  CuadradoLleno := TCuadradLleno.Create;
  CuadradoLleno.PintarCuadrado;
  CuadradoLleno.Free;
end.
```

Como se ve, la POO exige algo más de código pero no implica que los archivos compilados resultantes sean mayores. Simplemente la sintaxis es algo más compleja y se requieren algo más de código.

El resultado del programa anterior sería :

```
Pintar polígono de 3 lados

Pintar polígono de 3 lados
Llenando el polígono de color

Pintar polígono de 4 lados

Pintar polígono de 4 lados
Llenando el polígono de color
```

14.3.2.Problemas de los métodos estáticos

Vamos a ver el problema de los métodos estáticos en POO. Supongamos que definimos una clase TVehiculo con un método público Ruedas. Este método será una función que devolverá el número de ruedas del vehiculo. Para el caso de TVehiculo al ser genérico devolveremos -1. Declararemos tambien dos clases descendientes de TVehiculo llamadas TMoto y TCoche que tambien implementan la función Ruedas. En este caso el resultado será 2 y 4 respectivamente.

```
type
  TVehiculo = class (TObject)
    public
      function Ruedas : integer;
  end;

  TMoto = class (TVehiculo)
    public
      function Ruedas : integer;
```

```
  end;

  TCoche = class (TVehiculo)
    public
      function Ruedas : integer;
  end;

function TVehiculo.Ruedas : integer;
begin
  Ruedas := -1;
end;

function TMoto.Ruedas : integer;
begin
  Ruedas := 2;
end;

function TCoche.Ruedas : integer;
begin
  Ruedas := 4;
end;
```

Si declaramos una variable del tipo TVehiculo y la instanciamos con los constructores de TMotro o TCoche entonces podremos acceder al método Ruedas ya que TVehiculo lo lleva implementado. Esto es sintácticamente correcto ya que las clases TMoto y TCoche descienden de TVehiculo y por tanto todo lo que esté en TVehiculo también está en TMoto y TCoche. La forma inversa, pero, no es posible. Por este motivo es legal realizar instancias de este tipo.

Supongamos el programa siguiente :

```
var
  UnVehiculo : TVehiculo;
  UnCoche : TCoche;
begin
  UnVehiculo := TVehiculo.Create;
  Writeln(UnVehiculo.Ruedas);
  UnVehiculo.Free;

  UnCoche := TCoche.Create;
  Writeln(UnCoche.Ruedas);
  UnCoche.Free;

  UnVehiculo := TCoche.Create;
  Writeln(UnVehiculo.Ruedas);
  UnVehiculo.Free;
end.
```

El resultado del programa tendría que ser el siguiente :

```
-1
```

```
4
4
```

Pero al ejecutarlo obtenemos :

```
-1
4
-1
```

Qué es lo que ha fallado ? Pues ha fallado el hecho de que el método Ruedas no es virtual, es estático. Este comportamiento es debido a la forma de enlazar los métodos. Los métodos estáticos, o no virtuales, se enlazan en tiempo de compilación. De esta forma las variables de tipo TVehiculo siempre ejecutarán TVehiculo.Ruedas aunque las instanciemos con los constructores de clases descendientes. Esto es así para evitar que una clase inferior cambiara de visibilidad el método. A qué metodo tendría que llamar el compilador ? Por ejemplo, si TCoche.Ruedas fuera privado a quien habría tenido que llamar TVehiculo ?

Para superar este problema los lenguajes orientados a POO nos aportan un nuevo tipo de métodos llamados métodos virtuales.

14.3.3.Los métodos virtuales

Qué es lo que queremos resolver con los métodos virtuales exactamente ? Queremos que dada una instancia de la clase superior a partir de clases inferiors podamos ejecutar el método de la clase superior pero con la implementación de la clase inferior. O sea, dada una variable del tipo TVehiculo creamos la instancia con TMoto, por ejemplo, queremos que la llamada a Ruedas llame a TMoto ya que es la clase con la que la hemos instanciado. Porque es posible ? Básicamente porque los métodos virtuales se heredan como los otros métodos pero su enlace no se resuelve en tiempo de compilación sino que se resuelve en tiempo de ejecución. El programa no sabe a qué metodo hay que llamar, sólo lo sabe cuando se ejecuta. Esta característica de los objetos y las clases recibe el nombre de *vinculación retardada* o *tardía*.

Para indicar al compilador de que un método es virtual lo terminaremos con la palabra reservada virtual. Sólo hay que hacerlo en la declaración de la clase. Modificamos TVehiculo para que Ruedas sea virtual.

```
type
  TVehiculo = class (TObject)
    public
      function Ruedas : integer; virtual;
  end;
```

Si en una clase definimos un método virtual y queremos que en sus clases descendientes tambien lo sean tendremos que emplear la palabra reservada `override`. Los métodos override indican al compilador que son métodos virtuales que heredan de otros métodos virtual.

```
TMoto = class (TVehiculo)
  public
    function Ruedas : integer; override;
end;

TCoche = class (TVehiculo)
   public
    function Ruedas : integer; override;
end;
```

Esto indica al compilador que las funciones Ruedas de TCoche y TVehiculo son virtuales. El resultado del programa ahora será el que esperábamos :

```
-1
4
4
```

Téngase en cuenta de que si en una clase descendiente no hace el método override entonces este método volverá otra vez a ser estático, para esta clase y sus descendientes. Igualmente, si lo establecemos de nuevo `virtual` será virtual para las clases que desciendan. Por ejemplo, suprimiendo la directiva override de la declaración de TCoche el compilador nos advertirá que estamos ocultando una familia de métodos virtuales. El resultado será como si hubieramos empleado métodos estáticos. Igual pasa si lo declaramos `virtual` de nuevo. En cambio, la clase TMoto, que tiene Ruedas virtual, funcionaría correctamente. El resultado del programa siguiente :

```
var
  UnVehiculo : TVehiculo;
begin
  UnVehiculo := TCoche.Create;
  Writeln(UnVehiculo.Ruedas);
  UnVehiculo.Free;

  UnVehiculo := TMoto.Create;
  Writeln(UnVehiculo.Ruedas);
  UnVehiculo.Free;
end.
```

sería :

```
-1
2
```

ya que ahora TCoche le hemos suprimido override y no es nada más que un método estático y el compilador lo enlaza directamente con TVehiculo. Es posible dentro de los métodos `override` llamar al método superior mediante `inherited`.

14.3.4.Los métodos dinámicos

Si en vez de emplear `virtual` empleamos `dynamic` entonces habremos declarado un método dinámico. Los métodos virtual están optimizados en velocidad mientras que los dinámicos están optimizados en tamaño del código (producen un código compilado menor) según las especificaciones de Delphi.

En FreePascal no están implementados y emplear `dynamic` es como escribir `virtual`, pues se admite por compatibilidad con Delphi.

14.3.5.Classes descendientes de classes con métodos override

El funcionamiento es parecido a las clases que derivan de métodos `virtual`. Si no añadimos `override` a la declaración el método será estático y se enlazará con el primer método virtual u override que encuentre. Supongamos que definimos TCochecito, por ejemplo, que deriva de TCoche. TCochecito.Ruedas queremos que devuelva 3.

```
type
  TCochecito = class (TCoche)
    public
      function Ruedas : integer; override;
  end;

function TCochecito.Ruedas : integer;
begin
  Ruedas := 3;
end;
```

Podemos trabajar con esta clase de la forma siguiente :

```
var
  UnVehiculo : TVehiculo;
begin
  UnVehiculo := TCochecito.Create;
  Writeln(UnVehiculo.Ruedas);
  UnVehiculo.Free;
end.
```

Esto es posible ya que si TCoche desciende de TVehiculo y TCochecito desciende de TCoche entonces TCochecito tambien desciende de TVehiculo. Es la transitividad de la herencia.

Al ejecutar este código el resultado es 3. Pero si suprimimos override de TCochecito.Ruedas entoces el resultado es 4 pues se enlaza estáticamente con el primer método virtual que encuentra hacia arriba de la jerarquía de objetos, en este caso de TCoche.Ruedas.

14.4.Las clases abstractas

Una clase abstracta es una clase con uno o más métodos abstractos. Qué son los métodos abstractos ? Los métodos abstractos son métodos que no se implementan en la clase actual, de hecho no es posible hacerlo en la clase que los define, sino que hay que implementarlo en clases inferiores. Es importante tener en cuenta de que las clases que tengan métodos abstractos no se tienen que instancias. Esto es así para evitar que el usuario llame a métodos no implementados. Si intentamos instanciar una clase con métodos abstractos el código se compilará correctamente pero al ejecutarlo obtendremos un error de ejecución.

La utilidad de las clases abstractas se basa en definir métodos que se implementarán en clases inferiores por fuerza, a menos que las queramos inutilizar para no poderlas instanciar. Los métodos abstractos tienen que ser virtuales por definición.

Volviendo a los ejemplos anteriores podíamos haber declarado el método Ruedas de TVehiculo como abstracto y así evitar la implementación estraña que devolvía -1. Para definir un método como abstracto hay que añadir la palabara `abstract` después de `virtual`.

```
type
  TVehiculo = class (TObject)
    public
      function Ruedas : integer; virtual; abstract;
  end;
```

Ahora hay que suprimir la implementación de TVehiculo.Ruedas. Ahora ya no es posible hacer instancias de TVehiculo pero si emplear variables de tipo TVehiculo con instancias de TMoto y TCoche que tienen el método Ruedas de tipo `override`. Hay que ir con cuidado con los métodos abstractos, si una clase no implementa algun método abstracto de la clase superior entonces la clase tambien será abstracta. Si la implementación en clases inferiores se hace estáticamente entonces el enlace se resolverá en tiempo de compilación : se intentará enlazar con el método abstracto y se obtendrá un error de ejecución. Nótese que los métodos descendientes directos de un método abstracto no pueden llamar al método `inherited` pues no está implementado. Sí es posible hacerlo en métodos que no sean descendientes inmediatos de métodos abstractos, incluso si en algun momento de la jerarquía el método

ha sido abstracto. O sea, desde TCochecito podríamos llamar a `inherited` de Ruedas pero desde TCoche no.

14.5.Y el polimorfismo?

Ya lo hemos visto ocultado en los ejemplos anteriores. Cuando llamábamos Ruedas desde las distintas instancias hemos llamado al código adecuado gracias a los métodos virtuales.

La gracia es poder ejecutar código más concreto desde clases superiores. Por ejemplo, supongamos una clase TMusicPlayer. Esta clase implementa 4 métodos abstractos que son Play, Stop, Pause y Restart. El método Play hace sonar la música, el método Stop para la música, el método Pause pone el sistema musical en pausa y Restart lo reinicia.

Entencoes podríamos implementar clases desciendentes de TMusicPlayer como por ejemplo TWavPlayer que haría sonar un archivo de sonido de tipo .WAV. O TMidiPlayer que hace sonar un archivo MIDI. TMP3Player que hace sonar un archivo MP3. Y todo simplemente conociendo 4 métodos que son Play, Stop, Pause y Restart. Cada clase los implementará como precise, todo dependerá de qué constructor haya hecho la instancia. Si lo hemos instanciado con TMP3Player se llamarán los métodos de TMP3Player, si lo hacemos con TMIDIPlayer se llamaran los métodos de TMIDIPlayer, etc.

15.Conceptos avanzados de POO

15.1.Métodos de clase

En C++ se les llama métodos `static` (no tiene nada que ver con los métodos virtual) y en Pascal se les llama métodos de clase. Un método de clase es un método que opera con clases y no con objetos tal como hemos visto con todos los métodos que hemos visto hasta ahora.

Hasta ahora, para poder emplear los métodos de un objeto necesitábamos instanciar primero la clase en una variable del tipo de la clase. Con los métodos de clase no es necesario.ya que trabajan con la clase en sí. Esto quiere decir que los podemos llamar directamente sin haber tenido que instanciar la clase. El único caso que hemos visto hasta ahora era el método Create que se podía llamar directamente sin tener que instanciar la clase.

A diferencia de los métodos normales, los métodos de clase no pueden acceder a los campos de la clase (pues no tienen definida la variable Self). Tampoco pueden llamar a métodos que no sean métodos de clase ni mucho menos acceder a propiedades.

La utilidad de los métodos de clase es bastante restringida al no poder acceder a otros métodos que no sean tambien de clase ni a otras variables del objeto. A diferencia de otros lenguajes orientados a objetos como C++ o el Java, Pascal no incorpora un equivalente de "variables de clase" (o variables estáticas segun la nomenclatura de C++ y Java) y por tanto la utilidad de los métodos de clase queda bastante reducida. Como sustituto de estas variables de clase, que en Pascal no existen, podemos emplear las variables de la sección implementation de una unit, ya que es posible declarar una clase en la sección interface de la unit e implementarla en la sección implementation.

Para declarar un método de clase añadiremos la palabra reservada class antes de la definición del método. Como a ejemplo vamos a ver un método de clase que nos indique cuantas instancias se han hecho de esta clase. Para hacerlo tendremos que sobreesribir el constructor y destructor de la clase. Recordemos que el constructor por defecto es Create y el destructor por defecto es Destroy. Es posible añadir más de un constructor o destructor (este último caso es muy poco habitual). Los constructores y destructores son métodos pro que se declaran con la palabra `constructor` y `destructor` respectivamente.

Los constructores suelen llevar parámetros que determinan algunos aspetos de la funcionalidad de la clase. Tambien inicializan variables y hacen comprovaciones iniciales. Los destructores, a la contra, no suelen llevar parámetros (no tiene mucho sentido) y se encargan de liberar la memoria reservada en el constructor : punteros, otras clases, archivos abiertos, etc.

Es importante que el constructor antes de hacer nada, llame al constructor de la clase superior, que acabará por llamar al constructor de la clase TObject. Esto es así para asegurarse que la reserva de

memoria es correcta. En cambio, el constructor tiene que llamar al destructor de la clase superior una vez ha liberado sus datos, o sea, al final del método. Téngase en cuenta de que en la definición de TObject Destroy es virtual, de forma que tendremos que hacerlo `override` si queremos que todo funcione perfectamente.

Para implementar el ejemplo que hemos propuesto no hay más remedio que emplear una unit. Esto es así porque emplearemos una variable que no queremos que accesible desde fuera de la clase y tampoco desde fuera de la unit.

En la sección interface declaramos la clase :

```
unit EjemploClase;
{$MODE OBJFPC}
interface

type
    TContarInst = class (TObject)
      public
        constructor Create;
        destructor Destroy; override;
        class function ContarInstancias : Integer; // Método de clase
     end;
```

En la sección implementation declararemos la que hará las veces de variable de clase, aunque no lo sea.

```
implementation

var
  NumInstancias : Integer; // Hará las veces de "variable de clase"
```

Ya podemos implementar el constructor y el destructor. Tambien inicializaremos la variable NumInstancias a cero, en la sección inicialización de la unit.

```
constructor TContarInst.Create;
begin
  inherited Create;
  NumInstancias := NumInstancias + 1;
end;

destructor TContarInst.Destroy;
begin
  NumInstancias := NumInstancias - 1;
  inherited Destroy;
end;
```

```
function TContarInst.ContarInstancias : Integer;
begin
  ContarInstancias := NumInstancias; // Devolvemos la "variable de clase"
end;

initialization
  NumInstancias := 0;
end.
```

Llegados aquí hay que hacer un comentario sobre inherited. Es posible que la clase ascendente implemente otros métodos con el mismo nombre y distintos parámetros, métodos sobrecargados. Para llamar al método adecuade es importante pasar bien los parámetros como si de una llamada normal se tratara pero precedida de inherited. En este caso no lo hemos hecho porque no lleva parámetros, incluso podríamos haber escrito simplemente `inherited;` pues el compilador ya entiende que el método que queremos llamar es el mismo que estamos implementando pero de la clase superior (Create y Destroy).

Vamos a ver el funcionamiento de esta clase que cuenta sus instancias.

```
program MetodosDeClase;
{$MODE OBJFPC}
uses EjemploClase;
var
  C1, C2, C3 : TContarInst;
begin
  Writeln('Instancias al empezar el programa : ', TContarInst.ContarInstancias);
  C1 := TContarInst.Create; // Creamos una instancia
  Writeln('Instancias actuales : ', TContarInst.ContarInstancias);
  C2 := TContarInst.Create;
  Writeln('Instancias actuales : ', TContarInst.ContarInstancias);
  C3 := TContarInst.Create;
  Writeln('Instancias actuales : ', TContarInst.ContarInstancias);

  C3.Free;
  Writeln('Instancias actuales : ', TContarInst.ContarInstancias);
  C2.Free;
  Writeln('Instancias actuales : ', TContarInst.ContarInstancias);
  C1.Free;
  Writeln('Instancias al terminar el programa : ', TContarInst.ContarInstancias);
end.
```

Tal como es de esperar el resultado del programa es :

```
Instancias al empezar el programa : 0
Instancias actuales : 1
Instancias actuales : 2
Instancias actuales : 3
Instancias actuales : 2
Instancias actuales : 1
```

```
Instancias al terminar el programa : 0
```

15.1.2.Invocación de métodos de clase

Tal como hemos visto, podemos llamar a un método de clase a través de el nombre de la clase, pero también mediante una instancia. Dentro de los métodos convencionales tambien podemos llamar a los métodos de clase.

15.2.Punteros a métodos

Es posible emplear un tipo especial de tipo procedimiento/función que podríamos llamar tipo de método. La declaración es idéntica a la de un tipo de procedimiento/función pero añadiendo las palabras `of object` al final de la declaración de tipo.

```
type
    TPunteroMetodo = procedure (Num : integer) of object;
```

Este tipo de método se puede emplear dentro de las clases como haríamos con los tipos procedimiento/función normalmente. Es posible, como hemos visto, mediante tipos de procedimiento/función llamar diferentes funciones bajo un mismo nombre ya que las podemos asignar a funciones existentes. Los tipos de método tienen un comportamiento parecido. Declaramos la clase siguiente :

```
TImplementadorA = class (TObject)
  public
    procedure Ejecutar(Num : integer);
  end;

TImplementadorB = class (TObject)
  public
    procedure Ejecutar(Num : integer);
  end;
```

I una tercera clase donde emplearemos el puntero a método.

```
TEjemplo = class (TObject)
  public
    Operacion : TPunteroMetodo;
  end;
```

La implementación de las clases TImplementadorA y TImplementadorB es la siguiente :

```
procedure TImplementadorA.Ejecutar(Num : integer);
begin
  Writeln(Num*7);
end;

procedure TImplementadorB.Ejecutar(Num : integer);
begin
  Writeln(Num*Num);
end;
```

Ahora es posible, dado que las declaraciones de Ejecutar ambas clases TImplementador es compatible con el tipo TPunteroMetodo asignar a Operacion los metodos Ejecutar de las clases TImplementadorA y TImplementadorB. Observese el programa siguiente :

```
var
  ImplA : TImplementadorA;
  ImplB : TImplementadorB;
  Ejemplo : TEjemplo;
begin
  ImplA := TImplementadorA.Create;
  ImplB := TImplementadorB.Create;

  Ejemplo := TEjemplo.Create;

  Ejemplo.Operacion := @ImplA.Ejecutar;
  Ejemplo.Operacion(6); {42}

  Ejemplo.Operacion := @ImplB.Ejecutar;
  Ejemplo.Operacion(6); {36}

  Ejemplo.Free;

  ImplB.Free;
  ImplA.Free;
end.
```

Esta asignación no habría sido válida si TPunteroMetodo no fuera un tipo de método o tambien llamado puntero a método.

La utilidad de los punteros a método sirve cuando queremos modificar el comportamiento de un método. Téngase en cuenta que si el puntero a método no está asignado a algún otro método entonces al ejecutarlo se producirá un error de ejecución.

15.3.Paso de parámetros de tipo objeto

Podemos pasar parámetros de tipo objeto a funciones y procedimientos. En este caso habrá que ir con algo más de cuidado pues una clase no es una variable como las demás.

15.3.1.Las clases son siempre parámetros por referencia

Las clases se pasan siempre por referencia. Esto es así ya que un objeto, o sea, una instancia de clase no es nada más que un puntero y por tanto no tiene sentido pasar una clase por valor. Esto implica que las modificaciones en la clase que hagamos en la función permanecerán después de la llamada.

Si la clase se pasa como un parámetro constante entonces no se podrán modificar variables directamente (mediante una asignación) pero aún es posible llamar métodos que modifiquen los campos del objeto. En caso que el parámetro sea `var` es lo mismo que si hubieramos pasado un parámetro normal.

15.3.2.Las clases conviene que estén instanciadas

Es posible pasar una clase no instanciada e instanciarla dentro de la función pero no es muy recomendable pues nadie nos asegura de que ya ha sido instanciada. A la inversa tampoco, no libere clases dentro de funciones que hayan sido pasadas como parámetros.

15.3.3.Téngase en mente que los objetos son punteros

Por lo que no asigne un objeto a otro porqué no se copiará. Sólo se copiará la referencia de uno a otro y cuando modifique uno se verá tambien en el otro.

15.4.Referencias de clase

Hasta ahora cuando empleábamos el tipo clase era para declarar una variable del tipo de la clase. Ahora vamos a declarar variables que sean auténticas clases y no tipos de la clase. Por ejemplo esto es una variable de tipo TObject :

```
var
  Objeto : TObject
```

Pero tambien podemos tener lo que se llaman referencias de clase. O sea, “sinonimos” de nombres de clase. Para definir que una variable es una referencia de clase empleamos las palabras `class of`.

```
var
  TClase : class of TObject;
```

Qué es en realidad TClase ? Pues TClase es un tipo de clase estrictamente hablando, una referencia de clase. Por lo que es posible asignar a TClase otra clase pero no un tipo de clase (u objeto).

```
TClase := TObject; // Correcto
TClase := TEjemplo;
TClase := Objeto; // Incorrecto! Objeto es un objeto mientras que TClase es una clase
```

En realidad el conjunto de clases que podemos asignar a una referencia de clase es el conjunto de clases que desciendan de la referencia de la clase. En el caso anterior cualquier objeto se puede asignar a TClase ya que en Pascal cualquier clase desciende de TObject. Pero con el ejemplo del capítulo 14 si hacemos :

```
var
  TRefVehiculo : class of TVehiculo;
```

Sólo podremos asignar a TRefVehiculo : TVehiculo, TMoto y TCoche. Las referencias de clase llaman a los métodos que convenga segun la asignación y si el método es virtual o no. No hay ningún problema para que el constructor de una clase tambien pueda ser virtual.

15.5.Los operadores de RTTI *is* y *as*

RTTI, son las siglas de Run-Time Type Information y consiste en obtener información de los objetos en tiempo de ejecución. Los dos operadores que permiten obtener información sobre los objetos son `is` y `as`.

15.5.1.El operador *is*

Algunas veces pasaremos objetos como parámetros especificando una clase superior de forma que todos los objetos inferiores sean compatibles. Esto, pero tiene un inconveniente, no podemos llamar los métodos específicos de la clase inferior ya que sólo podemos llamar los de la clase superior. Para corregir este problema podemos preguntarle a la clase si es de un tipo inferior.

El operador is permite resolver este problema. Este operador devuelve una expresión booleana que es cierta si el objeto es del tipo que hemos preguntado.

En el ejemplo del capítulo 14 sobre TVehiculo, TCoche y TMoto tenemos que :

`TVehiculo is TObject`	es cierto porque todos los objetos son TObject
`TCoche is TVehiculo`	es cierto porque TCoche desciende de TVehiculo
`TMoto is TCoche`	es falso porque TMoto no desciende de TCoche.
`TObject is TVehiculo`	es falso porque TObject no será nunca un TVehiculo.
`TVehiculo is TVehiculo`	es cierto, un objeto siempre es igual a si mismo.

15.5.2.El operador *as*

El operador as hace un amoldado del objeto devolviendo un tipo de objeto al cual hemos amoldado. Es parecido a un typecasting pero resuelve algunos problemas sintácticos. Por ejemplo :

```
procedure Limpiar(V : TVehiculo);
begin
  if V is TCoche then
    TCoche(V).LavaParabrisas;
end.
```

Esto es sintácticamente incorrecto ya que TCoche no es compatible con TVehiculo (aunque TVehiculo lo sea con TCoche). Para resolver este problema tendremos que emplear el operador `as`.

```
if V is TCoche then
  (V as TCoche).LavaParabrisas;
```

Lo que hemos hecho es amoldar V de tipo TVehiculo a TCoche. El resultado de la expresion `V as TCoche` es un objeto del tipo TCoche que llamará a un hipotético método LavaParabrisas.

El operador as a diferencia de is puede fallar si hacemos una conversión incorrecta y lanzará una excepció. El ejemplo siguiente muestra una conversión incorrecta :

```
if V is TCoche then (V as TMoto)...
```

Como sabemos de la jerarquia de TVehiculo, TCoche no podrá ser nunca un tipo TMoto y viceversa.

Debido a que el resultado de as es un objeto del tipo del segundo operando, siempre y cuando la operación sea correcta, podremos asignar este resultado a un objeto y trabajar con este.

```
var
  C : TCoche;
begin
  if V is TCoche then
  begin
    C := V as TCoche;
    ...
  end;
end;
```

16.Gestión de excepciones

16.1.Errores en tiempo de ejecución

A diferencia de los errores de compilación, que suelen ser provocados por errores en la sintaxis del codigo fuente, hay otros errores a tener en cuenta en nuestros programas. Aparte de los errores que hacen que nuestro programa no lleve a cabo su tarea correctamente, causado en general por una implementación erronea, hay otros errores los cuales suelen causar la finalización de un programa. Estos errores reciben el nombre de errores en tiempo de ejecución (errores runtime) y como ya se ha dicho provocan la finalización anormal del programa. Por este motivo es importante que no ocurran.

Una forma eficiente de proteger nuestras aplicaciones pasa por lo que se llaman excepciones. Las excepciones son como alertas que se activan cuando alguna parte del programa falla. Entonces el código del programa, a diferencia de los errores runtime que son dificilmente recuperables, permite gestionar estos errores y dar una salida airosa a este tipo de fallos.

El soporte de excepciones lo aporta la unit SysUtils que transforma los errores runtime estandar en excepciones. No se olvide tampoco de incluir la directiva `{$MODE OBJFPC}` ya que las excepciones son clases.

16.1.1.Protegerse no duele

Aunque no todas las instrucciones lanzan excepciones, es importante proteger el código de nuestras aplicaciones. Una forma muy segura de anticiparse a los fallos es lanzar la excepción cuando se detecta que algo va mal, antes de que sea demasiado tarde. Como veremos, lanzar excepciones pone en funcionamiento el sistema de gestión de errores que FreePascal incorpora en el código.

16.2.Lanzar excepciones

Obsérvese el código siguiente. Es un algoritmo de búsqueda de datos en arrays de enteros. El algoritmo supone que el array está ordenado y entonces la búsqueda es más rápida puesto que si el elemento comprobado es mayor que el que buscamos entonces no es necesario que busquemos en posiciones posteriores, sino en las posiciones anteriores. La función devuelve cierto si se ha encontrado el elemento d en el array t, p contendrá la posición dentro del array. En caso de que no se encuentre la función devuelve falso y p no contiene información relevante.

```
// Devuelve cierto si d está en el array t, p indica la posición de la ocurrencia
// Devuelve falso si d no está en el array t
// El array t TIENE QUE ESTAR ORDENADA sino el algorismo no funciona
function BuscarDato(d : integer; t : array of integer; var p : integer) : boolean;
var
```

```
  k, y : integer;
begin
  p := Low(t); y := High(t);
  while p <> y do
  begin
    k := (p + y) div 2;
    if d <= t[k] then
    begin
      y := k;
    end
    else
    begin
      p := k + 1;
    end;
  end;
  BuscarDato := (t[p] = d);
end;
```

Esto funciona bien si los elementos del array están ordenados en orden creciente, en caso que no esté ordenado no funciona bien. Por este motivo añadiremos algo de código que comprobará que la tabla está ordenada crecientemente. En caso contrario lanzaremos la excepcion de array no ordenado EArrayNoOrdenado. La declaración, que irá antes de la función, de la excepción es la siguiente :

```
type EArrayNoOrdenado = class (Exception);
```

No es necesario nada más. Para lanzar una excepción emplearemos la sintaxis siguiente :

```
raise NombreExcepcion.Create('Mensaje');
```

El código que añadiremos al principio de la función es el siguiente :

```
function CercarDada(d : integer; t : array of integer; var p : integer) : boolean;
const
  EArrNoOrdMsg = 'El array tiene que estar bien ordenado';
var
  k, y : integer;
begin
  // Comprobamos si está bien ordenada
  for k := Low(t) to High(t)-1 do
  begin
    if t[k] > t[k+1] then // Si se cumple quiere decir que no está ordenado
      raise EArrayNoOrdenado.Create(EArrNoOrdMsg); // Lanzamos la excepcion !
  end;
  ... // El resto de la función
end;
```

Si el array no está bien ordenado se lanzará una excepción. Si nadie la gestiona, la excepción provoca el fin del programa. Por lo que vamos a ver como gestionar las excepciones.

16.3.Trabajar con excepciones

16.3.1.Código de gestión de excepciones

Si dentro de un código se lanza una excepción entonces el hilo de ejecución salta hasta el primer gestor de excepciones que se encuentra. Si no hay ninguno, el gestor por defecto es el que tiene la unit SysUtils. Lo único que hace es escribir el mensaje de la excepción y terminar el programa. Como se ve, esto no es práctico, sobretodo si la excepción no es grave (como el caso que hemos visto) y esta puede continuar ejecutándose.

Es posible añadir código de protección que permita gestionar y resolver las excepciones tan bien como se pueda. Para hacerlo emplearemos la estructura `try .. except .. end`.

Entre `try` y `except` pondremos el código a proteger. Si dentro de esta estructura se produce una excepción entonces el hilo de ejecución irá a la estructura `except .. end`. Ahora podemos realizar acciones especiales en función de diferentes excepciones mediante la palabra reservada `on`.

En el caso de que no haya habido ninguna excepción en el bloque `try` entonces el bloque `except` se ignora.

Veamos un ejemplo que casi siempre fallará puesto que inicializaremos la tabla de enteros con valores aleatorios.

```
var
  tabla : array [1..50] of Integer;
  i : integer;
begin
  try
    Randomize;
    for i := Low(tabla) to High(tabla) do
    begin
      tabla[i] := Random(400);
    end;
    BuscarDato(2, tabla, i);
  except
    on EArrayNoOrdenat do
    begin
      Writeln('La tabla no está ordenada!');
    end;
  end;
end.
```

Como se puede apreciar la estructura try y except no precisa de begin y end para incluir más de una instrucción. Cuando se produce una excepción de tipo EArrayNoOrdenado entonces se escribe en pantalla que la tabla no está ordenada. Para gestionar otras excepciones podemos añadir nuevas estructuras del tipo `on .. do`. Tambien es posible añadir una estructura `else` que se activará en el caso de que la excepción no coincida con ninguna de las anteriores.

```
try
  // Instrucciones protegidass
except
  on EExcepcion1 do begin ... end; // Varias sentencias
  on EExcepcion2 do ...; // Una sola sentencia
  else // En caso de que no haya sido ninguna de las anteriores
  begin
       // Sentencias
  end;
end;
```

Tambien es posible especificar un código único de gestión para todas las excepciones que se produzcan. Simplemente no especificamos ninguna excepción y escribimos directamente las sentencias.

```
try
  // Sentencias protegidas
except
  // Sentencias de gestión de las excepciones
end;
```

Dentro de una estructura `on EExcepcion do` podemos especificar un identificador de trabajo de la excepción. Por ejemplo podemos escribir el mensaje propio de la excepción, el que hemos establecido en la orden `raise`, de la forma siguiente :

```
try
  // Sentencias protegidas
except
 on E : Exception do Writeln(E.Message);
end;
```

El identificador E sólo existe dentro del código de gestión de excepciones. Este código se ejecutará siempre puesto que todas las excepciones derivan de Exception.

El tipo Exception tiene la definición siguiente :

```
Exception = class(TObject)
```

```
 private
  fmessage : string;
 public
  constructor Create(const msg : string);
  property Message : string read fmessage write fmessage;
end;
```

De hecho la declaración es algo más extensa para mantener la compatibilidad con Delphi, pero estos son los elementos básico de las excepciones en FreePascal. Como se ve, la propiedad Message es libremente modificable sin ningun problema. Esta propiedad se asigna en la llamada al constructor Create.

16.3.2.Código de finalización

Muchas veces nos interesará que un código de finalización se ejeucte siempre, incluso si ha habido una excepción. Esto es posible gracias a la estructura `try .. finally`. El código siguiente es bastante habitual :

```
try
  MiObjeto := TMiObjeto.Create(...);
  MiObjeto.Metodo1;
  MiObjeto.Metodo2;
  ...
  MiObjeto.MetodoN;
finally
  MiObjeto.Free;
end;
```

La sentencia MiObjeto.Free se ejecutará siempre, haya habido o no alguna excepcion en las sentencias anteriores a `finally`. En el caso de que haya una excepción, el hilo de ejecución pasará directamente al bloque `finally` sin completar las otras sentencias. Este código es sumamente útil para asegurarse de que una excepción no impedirá que se llamen las sentencias de finalización.

17.Sobrecarga de operadores

17.1.Ampliar las capacidades del lenguaje

Uno de los problemas más graves del lenguaje de programación Pascal es su rigidez sintáctica. FreePascal añade una característica interesante, a la vez que potente, llamada sobrecarga de operadores. La sobrecarga de operadores se basa en añadir nuevas funcionalidades a algunos operadores. Como sabemos, un mismo operador no siempre hace la misma función. Sin ir más lejos, el operador + concatena cadenas y hace sumas enteras y reales.

Dónde más utilidad puede tener la sobrecarga de operadores está en añadir nuevas posibilidades al lenguaje de forma que es posible emplear los operadores de una forma más intuitiva. Este hecho abre un abanico enorme de posibilidades desde el punto de vista programático. Ya no será ncesario emplear funciones especiales para poder sumar unos hipoteticos tipos TMatriz o TComplejo sino que podremos emplear los operadores que hemos sobrecargado.

17.1.1.Operadores sobrecargables

No es posible sobrecargar todos los operadores, sólo los más típicos dentro de expresiones. Estos operadores son :

Aritméticos : `+ - * / **`

Relacionales : `= < <= > >=`

Assignación : `:=`

Como se puede ver, no es posible sobrecargar el operador de desigualdad (`<>`) ya que no es nada más que el operador de igualdad pero negado.

El resto de operadores de Pascal (`^`, `@`) o las palabras reservadas que hacen las veces de operadoras tampoco se pueden sobrecargar (`div`, `mod`, `is`, ...).

17.2.Un ejemplo completo : operaciones dentro del cuerpo de matrices

Una matriz es una tabla formada por elementos. Generalmente estos elementos suelen ser reales, pero tambien podría ser una tabla de enteros. Esto es un ejemplo de matriz :

$$A_{2x3}=\begin{pmatrix} -2 & 1 & -5 \\ 10 & -7 & 3 \end{pmatrix}$$

Esta matriz es de 2 filas por 3 columnas. Vamos a definir una clase que represente una matriz. Dado que FreePascal no admite los arrays dinámicos, tendremos que hacer alguna "trampa" para poder disponer de un comportamiento similar a un array dinámico, o sea, de tamaño no prefijado antes de compilar. Nos aprovecharemos de una propiedad de los punteros que permite acceder a los datos con sintaxis de array. La clase, de nombre TMatriz, es la siguiente :

```
{$MODE OBJFPC}
uses SysUtils; // Para las excepciones

// La matriz estará indexada en cero ---> [0..CountI-1, 0..CountJ-1]
type
  TMatriz = class(TObject)
   private
     PTabla : Pointer;
     FTabla : ^Extended;
     FCountI, FCountJ : Integer;
     procedure SetIndice(i, j : integer; v : Extended);
     function GetIndice(i, j : integer) : Extended;
   public
     constructor Create(i, j : integer);
     destructor Destroy; override;
     procedure EscribirMatriz;
     procedure LlenarAleatorio(n : integer);
     property Indice[i,j:integer]:Extended read GetIndice write SetIndice; default;
     property CountI : Integer read FCountI;
     property CountJ : Integer read FCountJ;
  end;

constructor TMatriz.Create(i, j : integer);
begin
  inherited Create;
  FCountI := i;
  FCountJ := j;
  GetMem(PTabla, SizeOf(Extended)*FCountI*FCountJ); // Reservamos memoria suficiente
  FTabla := PTabla;
end;

destructor TMatriu.Destroy;
begin
  FreeMem(PTabla, SizeOf(Extended)*FCountI*FCountJ); // Liberamos la tabla
  inherited Destroy;
end;

procedure TMatriu.SetIndice(i, j : integer; v : Extended);
begin
  if ((0 <= i) and (i < CountI)) and ((0 <= j) and (j < CountJ)) then
  begin
    FTabla[CountJ*i + j] := v;
  end
  else raise ERangeError.Create('Rango no válido para la matriz');
end;
```

```
function TMatriu.GetIndice(i, j : integer) : Extended;
begin
  GetIndice := 0;
  if ((0 <= i) and (i < CountI)) and ((0 <= j) and (j < CountJ)) then
  begin
    GetIndice := FTaula[CountJ*i + j];
  end
  else raise ERangeError.Create('Rango no válido para la matriz');
end;

procedure TMatriu.EscribirMatriz;
var
  i, j : integer;
begin
  for i := 0 to CountI-1 do
  begin
    for j := 0 to CountJ-1 do
    begin
      Write(Self[i, j]:3:1, ' ');
    end;
    Writeln;
  end;
end;

procedure TMatriu.LlenarAleatorio(n : integer);
var
  i, j : integer;
begin
  for i := 0 to CountI-1 do
  begin
    for j := 0 to CountJ-1 do
    begin
      Self[i, j] := Random(n);
    end;
  end;
end;
```

Trabajar con la matriz es sencillo. Simplemente la creamos, indicando el tamaño que queramos que tenga. Después la llenamos con valores aleatorios y siempre que queramos podemos escribir la matriz. Esto lo podemos hacer mediante los procedimientos EscribirMatriz y LlenarAleatorio. Por ejemplo :

```
var
  Matriz : TMatriz;
begin
  try
    Matriz := TMatriz.Create(2, 3);
    Matriz.LlenarAleatorio(20);
    Matriz.EscriureMatriu;
  finally
    Matriz.Free;
  end;
```

```
end.
```

Un posible resultado sería el siguiente :

```
8.0 16.0 18.0
13.0 19.0 3.0
```

17.2.1.La operación suma

Dadas dos matrices A_{nxm} y B_{nxm} podemos hacer la suma y obtener una matriz C_{nxm} donde todos los elementos son $c_{i,j} = a_{i,j} + b_{i,j}$

Per ejemplo :

$$A_{2x3}=\begin{pmatrix} -2 & 1 & -5 \\ 10 & -7 & 3 \end{pmatrix} \quad B_{2x3}=\begin{pmatrix} 7 & 1 & 13 \\ 65 & -9 & 2 \end{pmatrix}$$

$$A+B=C$$

$$C_{2x3}=\begin{pmatrix} 5 & 2 & 8 \\ 75 & -16 & 5 \end{pmatrix}$$

Como es evidente sólo podremos sumar matrices del mismo tamaño y el resultado tambien será una matriz del mismo tamaño. La operación suma, además, es conmutativa por lo que A+B = B+A. Con todos estos datos ya podemos redefinir el operador +.

Para redefinir operadores emplearemos la palabra reservada `operator` que indica al compilador que vamos a sobrecargar un operador. Justo después hay que indiar el operador en cuestión, los dos operados, el identificador de resultado y su tipo. En nuestro caso, el operador + devolverá una instancia a un objeto de tipo matriz con el resultado. Si queremos asignar el resultado a una matriz habrá que tener en cuenta de que no tiene que estar instanciada. Si las matrices no se pueden sumar porque tienen dimensiones diferentes, entonces emitiremos una excepción que será necesario que sea gestionada y el resultado de la operación será un puntero `nil`.

```
operator + (A, B : TMatriz) C : TMatriz;
var
  i, j : integer;
begin
  // C no tiene que estar instanciad porque lo haremos ahora
  C := nil;
  if (A.CountI = B.CountI) and (A.CountJ = B.CountJ) then
  begin
    // Creamos la matriz de tamaño adecuado
```

```
    C := TMatriz.Create(A.CountI, A.CountJ);
    for i := 0 to A.CountI-1 do
    begin
      for j := 0 to A.CountJ-1 do
      begin
        // Sumamos reales
        C[i, j] := A[i, j] + B[i, j];
      end;
    end;
  end
  else
  begin
    // Lanzamos una excepción
    raise EDimError.Create('Tamaño de matrices distinto');
  end;
end;
```

Hemos sobreescrito el operador para que permita realizar operaciones de objetos de tipo TMatriz. Como hemos comentado, la suma de matrices es conmutativa. Ya que los dos operadores son iguales, no es necesario sobrecargar el operador para permitir la operación conmutada. La excepción EDimError la hemos definida para indicar excepciones de dimensiones de matrices.

Este es un ejemplo del uso de la suma sobrecargada :

```
var
  MatrizA, MatrizB, MatrizC : TMatriz;
begin
  Randomize; { Para que salgan números aleatorios }
  try
    MatrizA := TMatriz.Create(2, 3);
    MatrizA.LlenarAleatorio(50);
    Writeln('A = ');
    MatrizA.EscribirMatriu;
    Writeln;

    MatrizB := TMatriz.Create(2, 3);
    MatrizB.LlenarAleatorio(50);
    Writeln('B = ');
    MatrizB.EscribirMatriz;
    Writeln;

    // MatrizC s'inicialitza a la suma !!!
    try
      Writeln('C = ');
      MatrizC := MatrizA + MatrizB;
      MatrizC.EscribirMatriz;
      Writeln;
    except
      on E : EDimError do
      begin
        Writeln(E.message);
      end;
```

```
    end;
  finally
    MatrizA.Free;
    MatrizB.Free;
    MatrizC.Free;
  end;
end.
```

En este caso el código no falla pero si hubieramos creado matrices de tamaños distintos hubieramos obtenido un mensaje de error. La salida del programa es parecida a la siguiente :

```
A =
14.0 10.0 33.0
10.0 23.0 46.0

B =
1.0 48.0 32.0
47.0 44.0 5.0

C =
15.0 58.0 65.0
57.0 67.0 51.0
```

17.2.2.Producto de una matriz por un escalar

Es posible multiplicar una matriz por un numero escalar de forma que todos los elementos de la matriz queden multiplicados por este número. Téngase en cuenta de que esta operación tambien es conmutativa k·A = A·k donde k es un real. En este caso, los dos operandos son diferentes y habrá que redefinir dos veces el producto puesto que sino no podríamos hacer productos a la inversa. En estos casos no lanzaremos ninguna excepción pues el producto siempre se puede llevar a cabo. Ahora tenemos que redefinir el operador *.

```
operator * (A : TMatriz; k : Extended) C : TMatriz;
var
  i, j : integer;
begin
  C := TMatriz.Create(A.CountI, A.CountJ);
  for i := 0 to A.CountI-1 do
  begin
    for j := 0 to A.CountJ-1 do
    begin
      C[i, j] := A[i, j]*k;
    end;
  end;
end;

operator * (k : Extended; A : TMatriz) C : TMatriz;
```

```
begin
  C := A*k; // Esta operación ya ha sido sobrecargada
end;
```

Un ejemplo del funcionamiento de una matriz por un escalar :

```
var
  MatrizA, MatrizC : TMatriu;

begin
  Randomize;
  try
    MatrizA := TMatriz.Create(2, 3);
    MatrizA.LlenarAleatorio(50);
    Writeln('A = ');
    MatrizA.EscribirMatriz;
    Writeln;
    Writeln('A*12 = ');
    MatrizC := MatrizA*12;
    MatrizC.EscribirMatriz;
    MatrizC.Free;
    Writeln;
    Writeln('12*A = ');
    MatrizC := 12*MatrizA;
    MatrizC.EscribirMatriz;
    MatrizC.Free;

  finally
    MatrizA.Free;
  end;
end.
```

La salida del programa es la siguiente :

```
A =
22.0 47.0 37.0
11.0 15.0 28.0

A*12 =
264.0 564.0 444.0
132.0 180.0 336.0

12*A =
264.0 564.0 444.0
132.0 180.0 336.0
```

Como se ve, la sobrecarga de * que hemos implementado es forzosamente conmutativa.

17.2.3.Producto de matrices

Las matrices se pueden multiplicar siempre que multipliquemos una matriz del tipo A_{nxm} con una matriz del tipo B_{mxt}. El resultado es una matriz de tipo C_{nxt}. El cálculo de cada elemento es algo más complejo que en las dos operaciones anteriores :

$$A_{2x3}=\begin{pmatrix} -2 & 1 & -5 \\ 10 & -7 & 3 \end{pmatrix} \quad B_{3x4}=\begin{pmatrix} 4 & -3 & 2 & -1 \\ -2 & 0 & -7 & 6 \\ 0 & -1 & 4 & 7 \end{pmatrix}$$

$$A_{2x3}.B_{3x4}=C_{2x4}$$

$$C=\begin{pmatrix} (-2)\cdot 4+1(-2)+(-5)\cdot 0 & (-2)(-3)+1\cdot 0+(-5)(-1) & (-2)\cdot 2+1\cdot(-7)+(-5)\cdot 4 & (-2)(-1)+1\cdot 6+(-5)\cdot 7 \\ 10\cdot 4+(-7)(-2)+3\cdot(0) & 10\cdot(-3)+(-7)\cdot 0+3\cdot(-1) & 10\cdot 2+(-7)(-7)+3\cdot 4 & 10\cdot(-1)+(-7)\cdot 6+3\cdot 7 \end{pmatrix}$$

$$C_{2x4}=\begin{pmatrix} -10 & 11 & -31 & -27 \\ 54 & -33 & 8\ 1 & -31 \end{pmatrix}$$

El algoritmo de multiplicación es más complicado. El operador sobre cargado es el siguiente :

```
operator * (A, B : TMatriz) C : TMatriz;
var
  i, j, t : integer;
  sum : extended;
begin
  C := nil;
  // Es necesario ver si las matrices son multiplicables
    if A.CountJ <> B.CountI then raise EDimError.Create('Las matrices no son
multiplicables');
  C := TMatriz.Create(A.CountI, B.CountJ);
  for i := 0 to C.CountI-1 do
  begin
    for j := 0 to C.CountJ-1 do
    begin
      sum := 0;
      for t := 0 to A.CountJ-1 do
      begin
        sum := sum + A[i, t]*B[t, j];
      end;
      C[i, j] := sum;
    end;
  end;
end;
```

Veamos un código de ejemplo de esta operación :

```
var
  MatrizA, MatrizB, MatrizC : TMatriz;
begin
  Randomize;
  try
    Writeln('A = ');
    MatrizA := TMatriz.Create(3, 4);
    MatrizA.LlenarAleatorio(10);
    MatrizA.EscribirMatriz;
    Writeln;
    Writeln('B = ');
    MatrizB := TMatriz.Create(4, 2);
    MatrizB.LlenarAleatorio(10);
    MatrizB.EscribirMatriz;
    Writeln;
    try
    MatrizC := (MatrizA*MatrizB);
    Writeln('A x B = ');
    MatrizC.EscribirMatriz;
    except
      on E : EDimError do
      begin
        Writeln(E.message);
      end;
    end;
  finally
    MatrizA.Free;
    MatrizB.Free;
    MatrizC.Free;
  end;
end.
```

Este programa genera una salida parecida a la siguiente :

```
A =
2.0 -8.0 4.0 -2.0
7.0 2.0 0.0 -3.0
0.0 -3.0 -4.0 -4.0

B =
-4.0 0.0
-5.0 5.0
-1.0 -2.0
4.0 3.0

A x B =
20.0 -54.0
-50.0 1.0
3.0 -19.0
```

18.Programación de bajo nivel

18.1.Los operadores bit a bit

Llamamos operadores bit a bit a los operadores que permiten manipular secuencias de bits. Todos los datos en un ordenador se codifican en base de bits que pueden tomar el valor 0 o 1. Los enteros, por ejemplo, son conjuntos de bits de diferentes tamaños (byte, word, double wor, quadruple word...).

Algunas veces, en programación de bajo nivel y ensamblador, nos interesará modificar los valores bit a bit. FreePascal incorpora un conjunto de operadores que hacen este trabajo por nosotros.

Los operadores bit a bit son : `not`, `or`, `and`, `xor`, `shl` y `shr`. Los operadors bit a bit sólo trabajan sobre enteros. En los ejemplos emplearemos los números 71 y 53 que en representanción binaria de 8 bits son :

$71_{(10} = 01000111_{(2}$
$53_{(10} = 00110101_{(2}$

18.1.1.El operador or

El operador or hace un or binario que responde a la tabla siguiente :

		or
0	0	0
0	1	1
1	0	1
1	1	1

En nuestro ejemplo :

```
71 or 53 = 01000111 or 00110101 = 01110111
```

18.1.2.El operador and

El operador and hace un and bit a bit que responde a la tabla siguiente :

		and
0	0	0

0	1	0
1	0	0
1	1	1

Ejemplo :

```
71 and 53 = 01000111 and 00110101 = 00000101
```

18.1.3.El operador xor

El operador xor hace una or exclusiva que responde a la tabla siguiente :

		xor
0	0	0
0	1	1
1	0	1
1	1	0

```
71 xor 53 = 01000111 xor 00110101 = 01110010
```

18.1.4.El operador not

El operador not hace una negacion binaria que responde a la tabla siguiente. Es un operador unario.

	not
0	1
1	0

```
not 71 = not 01000111 = 10111000
```

18.1.5.Los operadors shl y shr

Los operadores shl y shr hacen un desplazamiento de bits a la izquierda o derecha respectivamente de todos los bits (SHift Left, SHift Right). Los bits que hay que añadir de más son siempre cero. El número de desplazamientos viene indicado por el segundo operando.

```
71 shl 5 = 01000111 shl 00000101 = 11100000
71 shr 5 = 01000111 shr 00000101 = 00000010
```

Las operacions shl y shr representan productos i divisiones enteras por potencias de dos.

```
71 shl 5 = 71*(2**5)        (8 bits)
71 shr 5 = 71 div (2**5)
```

18.2.Empleo de código ensamblador

Es posible incluir sentencias de código ensamblador enmedio del código. Para indicar al compilador que lo que sigue es ensamblador emplearemos la palabra reservada `asm`.

```
Sentencias Pascal
...
asm
   Sentencias de ensamblador
end;
...
Sentencias Pascal
```

El código que se encuentra enmedio de asm y end se incluye directamente al archivo de ensamblaje que se le pasa al enlazador (linker).

Es posible espeficicar una función exclusivamente escrita en ensamblador mediante la directiva assembler. Esto produce funciones en ensamblador ligeramente más optimizadas.

```
procedure NombreProcedimiento(parametros); assembler;
asm
end;
```

Es posible hacer lo mismo con funciones.

18.2.1.Tipo de sintaxis del ensamblador

FreePascal soporta dos tipos de sintaxis de ensamblador. La sintaxis Intel, la más habitual y la más conocida cuando se trabaja en esta plataforma, y la sintaxis AT&T que es la sintaxis que se hace servir en el GNU Assembler, el ensamblador que emplea FreePascal. Por defecto la sintaxis es AT&T pero es posible emplear el tipo de sintaxis con la directiva de compilador $ASMMODE.

```
{$ASMMODE INTEL} // Sintaxis INTEL
{$ASMMODE ATT} // Sintaxis AT&T
{$ASMMODE DIRECT} { Los bloques de ensamblador se copian directamente al archivo de
                    ensamblaje }
```

Consulte en la documentación de FreePascal sobre las peculiaridades y diferencias entre 1a sintaxis Intel y la sintaxis AT&T. Consutle tambien como puede trabajar con datos declarados en sintaxis Pascal dentro del código ensamblador.

19.Compilación condicional, mensajes y macros

19.1.Compilación condicional

La compilación condicional permite modificar el comportamiento del compilador en tiempo de compilación a partir de símbolos condicionales. De esta forma definiendo o suprimiendo un símbolo podemos modificar el comportamiento del programa una vez compilado.

19.1.1.Definición de un símbolo

Vamos a ver un ejemplo de compilación condicional. Haremos un un programa que pida un año al usuario. Este año puede ser de dos cifras o bien puede ser de cuatro cifras. En ambos casos necesitaremos un entero : un byte para las dos cifras y un word para las cuatro cifras. Declararemos el símbolo `DIGITOS_ANYOS2` cuando queramos que los años tengan dos cifras, en otro caso tendrán cuatro cifras.

Para declarar un símbolo emplearemos la directiva de compilador `{$DEFINE nombresimbolo}` en nuestro caso cuando queramos dos cifras añadiremos al principio del código :

```
{$DEFINE DIGITOS_ANYOS2}
```

Como en la mayoría de identificadores Pascal, el nombre del símbolo no es sensible a mayúsculas. Por lo que concierne a la validez del nombre del símbolo se sigue el mismo criterio que los nombres de variables y constantes.

Esta directiva es local, de forma que, justo después (pero no antes) de incluirla el símbolo `DIGITOS_ANYOS2` queda definido. Pero tambien podemos desactivar o anular el símbolo más adelante.

19.1.2.Anulación de símbolos condicionales

Si llegados a un punto del código nos interesa anular un símbolo entonces emplearemos la directiva de compilador `{$UNDEF nombresimbolo}`. Por ejemplo, en nuestro caso :

```
{$UNDEF DIGITOS_ANYOS2}
```

anularía el símbolo, en caso de que estuviera definido. Si el símbolo no está definido entonces simplemente no pasa nada. Esta directiva tambien es local, a partir del punto donde la hemos añadido el símbolo queda anulado.

19.1.3.Empleo de los símbolos condicionales

Las directivas condicionales más usuales son {$IFDEF simbolo}, {$ELSE} y {$ENDIF}.

{$IFDEF nombresimbolo} compila todo lo que le sigue sólo si está definido el simbolo nombresimbolo. Además, inicia una estructura de compilación condicional que tenemos que terminar siempre con un {$ENDIF}. Si nombresimbolo no estuviera definido podríamos emplear la directiva {$ELSE} para indicar que sólo se compilará si nombresimbolo no está definido. {$ELSE} tiene que estar entre $IFDEF y $ENDIF.

Obsérvese el código siguiente :

```
program CompilacionCondicional;

{$IFDEF DIGITOS_ANYOS2}
procedure PedirAnyo(var Dato : Byte);
begin
  repeat
    Write('Introduzca el valor del año (aa) : ');
    Readln(Dato);
  until (0 <= Dato) and (Dato <= 99);
end;
{$ELSE}
procedure PedirAnyo(var Dato : Word);
begin
  repeat
    Write('Introduzca el valor del año (aaaa) : ');
    Readln(Dato);
  until (1980 <= Dato) and (Dato <= 2099);
end;
{$ENDIF}

var
 {$IFDEF DIGITOS_ANYOS2}
 anyo : byte;
 {$ELSE}
 anyo : word;
 {$ENDIF}
begin
  PedirAnyo(anyo);
end.
```

Si compilamos ahora este código, el programa nos pedirá cuatro cifras para el año. En cambio si después de la cláusula program añadimos {$DEFINE DIGITOS_ANYOS2}

```
program CompilacionCondicional;

{$DEFINE DIGITOS_ANYOS2} // Definimos el símbolo

{$IFDEF DIGITOS_ANYOS2}
```

```
procedure SolicitarAny(var Dato : Byte);
...
```

entonces el programa una vez compilado sólo pedirá dos cifras. Como se puede deducir, por defecto los símbolos no están activados.

La directiva {$IFNDEF nombresimbolo} actúa al revés que $IFDEF ya que compilará el bloque siguiente si **no** está definido nombresimbolo. En cuanto al uso, es idéntico a $IFDEF.

19.1.4.Empleo de símbolos comunes a todo un proyecto

El compilador no permite definir símbolos comunes a todo un conjunto de archivos sino que tienen que ser definidos para todos y cada uno de los archivos que necesitemos. Para no tener que añadirlos cada vez en cada archivo, sobretodo en proyectos grandes, podemos emplear la directiva de incluir archivo {$I nombrearchivo}.

La directiva {$I nombrearchivo} hace que el compilador sustituya esta directiva por el contenido del archivo que indica. Por ejemplo, podíamos haber creado un archivo llamado CONDICIONALES.INC (suelen llevar esta extensión) que tuviera la línia siguiente :

```
{$DEFINE DIGITOS_ANYOS2}
```

y después haberlo incluido en el programa :

```
program CompilacionCondicional;

{$I CONDICIONALES.INC}

{$IFDEF DIGIT_ANYS2}
procedure PedirAnyo(var Dato : Byte);
...
```

De esta forma no es necesario cambiar varios $DEFINE en cada archivo. Simplemente cambiando la directiva contenida en CONDICIONALES.INC tendremos suficiente.

Es importante que el archivo incluido pueda ser encontrado por el compilador. Lo más fácil es que se encuentre en el mismo directorio que el archivo que lo incluye. Tambien podemos indicarle al compilador que lo busque al directorio que nosotros queramos mediante el parámetros -Fi*directorio* donde directorio es el directorio donde puede encontrar el archivo. Tambien podemos emplear la directiva {$INCLUDEPATH directorio*1*;..;directorio*n*}

Como se ve, los archivos incluidos permiten resolver algunos problemas que no podían solventarse sólo con la sintaxis.

19.1.5.Trabajar con directivas de tipo *switch*

Hasta el momento, hemos visto algunas directivas de tipo switch {$I+}/{$I-}, {$H+}/{$H-} que permiten desactivar algunas opciones del compilador. Para compilar un cierto código en función del estado de una directiva de este tipo podemos emplear {$IFOPT directiva+/-}. Por ejemplo :

```
{$IFOPT H+}
Writeln('String = AnsiString');
{$ELSE}
Writeln('String = String[255]');
{$ENDIF}
```

19.1.6.Símbolos predefinidos del compilador FreePascal.

Los símbolos siguientes están predefinidos a la hora de compilar programas con FreePascal.

Símbolo	**Explicación**
FPC	Identifica al compilador FreePascal. Es útil para distinguirlo de otros compiladores.
VERv	Indica la versión, donde v es la versión mayor. Actualmente es VER1
VERv_r	Indica la versión, donde v es la versión mayor y r la versión menor. Actualmente es VER1_0
VERv_r_p	Indica la versión, donde v es la versión mayor, r la versión menor y p la distribución. La versión 1.0.4 tiene activado el símbolo VER1_0_4
OS	Donde OS puede valer `DOS`, `GO32V2`, `LINUX`, `OS2`, `WIN32`, `MACOS`, `AMIGA` o `ATARI` e indica para qué entorno se está compilando el programa. En nuestro caso será `WIN32`.

19.2.Mensajes, advertencias y errores

Durante la compilación, el compilador puede emitir diversos mensajes. Los más usuales son los de error (precedidos por `Error:`) que se dan cuando el compilador se encuentra con un error que no permite la compilación. Normalmente suelen ser errores sintácticos. Aún así se continúa leyendo el código para ver si se encuentra algun otro error. Los errores fatales (precedidos por `Fatal:`) impiden que el compilador continue compilando. Suele ser habitual el error fatal de no poder compilar un archivo porque se ha encontrado un error sintáctico.

El compilador tambien emite mensajes de advertencia (precedidos de `Warning:`) cuando encuentra que algún punto es susceptible de error (por ejemplo, una variable no inicializada). Tambien emite mensajes de observación (`Note:`) y consejos (`Hint:`) sobre aspectos del programa, como son variables declaradas pero no empleadas, inicializaciones inútiles, etc.

Es posible hacer que el compilador emita mensajes de este tipo y que el compilador se comporte de forma idéntica a que si los hubiera generado él. Para esto emplearemos las directivas `$INFO`, `$MESSAGE` (que emiten un mensaje), `$HINT` (un consejo), `$NOTE` (una indicación), `$WARNING` (una advertencia), `$ERROR` (un error), `$FATAL`, `$STOP` (un error fatal). Todas las directivas toman como parámetro único una cadena de caracteres que no tiene que ir entre comas. Por ejemplo

```
{$IFDEF DIGITOS_ANYOS2}
{NOTE El soporte de cuatro digitos para años está desactivado}
{$ENDIF}
```

Hay que tener en cuenta que dentro de esta cadena de caracteres no podemos incluir el caracter } puesto que el compilador lo entenderá como que la directiva ya se ha terminado.

19.3.Macros

Las macros son una característica bastante habitual en C y C++ puesto que este lenguaje posee un preprocesador. En Pascal no suele haber un preprocesador pero FreePascal permite activar el soporte de macros, que por defecto está desactivado, con la directiva `{$MACRO ON}` y para desactivarlo `{$MACRO OFF}`.

Una macro hace que se sustituyan las expresiones del código por otras. Esta sustitución se hace antes de compilar nada, o sea, desde un punto de vista totalmente no síntáctico. Las macros se definen de la forma siguiente :

```
{$DEFINE macro := expresion}
```

Expresión puede ser cualquier tipo de instrucción, sentencia o elemente del lenguaje. En este ejemplo hemos sustituido algunos elementos sintácticos del Pascal por otros para que parezca pseudocódigo :

```
{$MACRO ON} // Hay que activar las macros
{$DEFINE programa := program}
{$DEFINE fprograma := end.}
{$DEFINE si := if}
{$DEFINE entonces := then begin }
{$DEFINE sino := end else begin}
{$DEFINE fsi := end;}
{$DEFINE fvar := begin}
{$DEFINE entero := integer}
{$DEFINE escribir := Writeln}
{$DEFINE pedir := readln}
```

```
programa Macros;
var
 a : entero;
fvar
 escribir('Introduzca un entero cualquiera : '); pedir(a);
 si a mod 2 = 0 entonces
   escribir('El número es par')
 sino
   escribir('El número es impar');
 fsi
fprograma
```