|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial  (para Delphi)** | **[Actualizar](http://www.marcocantu.com/EPascal/Spanish/default.htm) Introducción** |

1 de octubre de 1999: este libro (versión inglesa) viene incluido en el CD que acompaña a Delphi 5.   
Código fuente disponible aquí. Se añadieron los ejemplos.



Portada del libro. El dios Apolo adorado en Delphi,   
en un fresco italiano del siglo XVII.

Las primeras ediciones de *Mastering Delphi* (en español: Delphi 4), el libro que más se ha vendido de los que he escrito sobre Delphi, proveía una introducción al lenguaje Pascal en Delphi. Debido a restricciones de espacio y porque muchos programadores en Delphi buscan información más avanzada, en la penúltima edición del libro (*Mastering Delphi 4*) se omitió este material completamente. Para superar esta carencia, he comenzado a componer este libro *en línea*, llamado

***Pascal Esencial***.

Este es un libro detallado sobre Pascal, que por el momento estará disponible gratis en mi sitio en la Red (realmente no sé qué pasará después ... puede que incluso encuentre un editor). Este es un trabajo en desarrollo, y cualquier respuesta del lector es bienvenida. La primera versión completa de este libro, de julio de 1999, fue publicada en el CD que acompaña a Delphi 5.

**Copyright (derechos de autor)**

El texto y el código fuente de este libro son propiedad intelectual de Marco Cantù. Por supuesto, puede usted utilizar los programas y adaptarlos a sus propias necesidades, sólo que no se le permite usarlos en libros, material de enseñanza ni otros formatos sujetos a derechos de autor. Puede usted situar un enlace a este sitio desde su página en la Red, pero, por favor, no duplique el material, ya que está sujeto a frecuentes cambios y actualizaciones.

**La estructura del libro**

Sigue la actual estructura del libro:

* [**Capítulo 1: Historia de Pascal**](http://www.marcocantu.com/epascal/Spanish/ch01hist.htm)
* [**Capítulo 2: Codificación en Pascal**](http://www.marcocantu.com/epascal/Spanish/ch02code.htm)
* [**Capítulo 3: Tipos, Variables y Constantes**](http://www.marcocantu.com/epascal/Spanish/ch03data.htm)
* [**Capítulo 4: Tipos de Datos Definidos por el Usuario**](http://www.marcocantu.com/epascal/Spanish/ch04udt.htm)
* [**Capítulo 5: Instrucciones**](http://www.marcocantu.com/epascal/Spanish/ch05stat.htm)
* [**Capítulo 6: Procedimientos y Funciones**](http://www.marcocantu.com/epascal/Spanish/ch06proc.htm)
* [**Capítulo 7: Manejo de Cadenas**](http://www.marcocantu.com/epascal/Spanish/ch07str.htm) (de Caracteres)
* [**Capítulo 8: Memoria**](http://www.marcocantu.com/epascal/Spanish/ch08mem.htm) (actualm. sólo cubre vectores dinámicos)
* [**Capítulo 9: Programación Windows**](http://www.marcocantu.com/epascal/Spanish/ch09win.htm)
* [**Capítulo 10: Variantes**](http://www.marcocantu.com/epascal/Spanish/ch10var.htm)
* [**Capítulo 11: Programas y Unidades**](http://www.marcocantu.com/epascal/Spanish/ch11unit.htm)
* [**Apéndice A: Glosario de Términos**](http://www.marcocantu.com/epascal/Spanish/glossary.htm)
* [**Apéndice B: Traducciones de *Essential Pascal***](http://www.marcocantu.com/epascal/Spanish/transla.htm)
* [**Apéndice C: Ejemplos**](http://www.marcocantu.com/epascal/Spanish/examples.htm)

**C:\Users\juan xcarlos\Desktop\flash cell\INF-120 'SF'\libro de programacion1\default_files\newani.gifCódigo fuente**

Está disponible el código fuente de todos los ejemplos mencionados en el libro. Está sujeto al mismo Copyright que el libro: puede usted usarlo libremente, pero no publicarlo en otros documentos o sitio en la Red. Son bienvenidos enlaces a estas páginas.

Descargue el código fuente en un único archivo zip, [**EPasCode.zip**](http://www.marcocantu.com/epascal/EPasCode.zip) (sólo 26 kB), y compruebe la [**lista de ejemplos**](http://www.marcocantu.com/epascal/Spanish/examples.htm).

**Respuesta de los usuarios**

Por favor, hágamelo saber si encuentra cualquier error, o si hay algún aspecto que no esté lo bastante claro para un principiante. Podré dedicar tiempo al proyecto dependiendo de la respuesta que obtenga. Así que, si el proyecto le parece interesante y quiere verlo terminado, comuníquemelo. Hágame saber también qué otros asuntos (no cubiertos por *Mastering Delphi 4* [N. del T.: publicado en español como *Delphi 4*, por Anaya]) quisiera usted ver en este sitio. Suscríbase al grupo de noticias mencionado en mi sitio en la Red, y busque en él la sección de libros. Si tiene alguna petición concreta, escríbame **en inglés** a [**email**](http://www.marcocantu.com/home/mail.htm), escribiendo ***Essential Pascal*** en la línea de "asunto" y su solicitud o comentario en el "texto".

**Reconocimientos**

Si publico un libro en la Red gratis, es especialmente debido a la experiencia de Bruce Eckel con *Thinking in Java*. Soy amigo de Bruce y creo que hizo un trabajo realmente magnífico con este y otros libros.

Al mencionar el proyecto a la gente de Borland, también recibí una gran respuesta. Y, por supuesto, debo agradecer a la compañía el haber hecho la primera serie de compiladores de Turbo Pascal y ahora la serie Delphi de IDEs (entornos integrados de programación).

Estoy empezando a recibir respuestas de gran valor. Los primeros lectores que me ayudaron a mejorar este material bastante son Charles Wood y Wyatt Wong. Mark Greenhaw me ayudó con la edición del texto. Rafael Barranco-Droege ofreció muchas correcciones técnicas y lingüísticas. Gracias.

**Traducciones**

*Essential Pascal* está siendo traducido a unos cuantos idiomas, entre ellos el japonés, holandés, francés, turco y español. Las versiones traducidas estarán disponibles gratis en la Red y habrá enlaces a ellas desde mi sitio en la Red. Si está usted interesado en traducir *Essential Pascal* o busca una versión traducida, consulte la página de [**Traducciones**](http://www.marcocantu.com/epascal/Spanish/transla.htm).

**El autor**

Marco Cantù vive en Piacenza, Italia. Después de escribir libros y artículos sobre C++ y Object Windows Library, se dedicó a la programación en Delphi. Es el autor de la serie de libros *Mastering Delphi* publicada por Sybex [N. del T. : Cuarta edición publicada en castellano bajo el título *Delphi 4*, por Anaya], como también del *Delphi Developers Handbook*, para programadores avanzados. Escribe artículos para muchas revistas, incluida *The Delphi Magazine*, da conferencias sobre Delphi y Borland en todo el mundo, e imparte clases de Delphi de niveles básico y avanzado.

Puede encontrar más detalles sobre Marco y su trabajo en su sitio en la Red, [**www.marcocantu.com**](http://www.marcocantu.com/).

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 1 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch01hist.htm) Historia de Pascal** |

El lenguaje de programación Object Pascal que usamos en Delphi no fue inventado en 1995 con el entorno visual de desarrollo de Borland. Simplemente, era una extensión de Object Pascal, que ya se estaba usando en los productos Pascal de Borland. Pero Borland no inventó Pascal, sólo ayudó a hacerlo popular y lo extendió un poco ...

Este capítulo contiene alguna información histórica de fondo sobre el lenguaje Pascal y su evolución. Por el momento sólo contiene resúmenes muy cortos.

**El Pascal de Wirth**

El lenguaje Pascal fue diseñado originalmente EN 1971 por Niklaus Wirth, profesor en el Politécnico de Zúrich, Suiza. Fue concebido como versión simplificada con fines educativos el lenguaje Algol, que data de 1960.

Cuando Pascal fue diseñado, ya existían muchos lenguajes de programación, pero pocos eran de uso generalizado : FORTRAN, C, ensamblador, COBOL. La idea clave del nuevo lenguaje fue el orden, administrado mediante un concepto sólido de tipo de dato, y requiriendo declaraciones de tipo y controles de programa estructurados. El lenguaje fue diseñado también para ser un instrumento de enseñanza a estudiantes que aprendían a programar.

**Turbo Pascal**

El compilador de Pascal de Borland, famoso en todo el mundo, fue presentado en 1985. El compilador Turbo Pascal ha sido una de las series de compiladores que mejor se han vendido de todos los tiempos, e hizo de Pascal un lenguaje especialmente importante en la plataforma PC, gracias a su equilibrio entre simplicidad y potencia.

Turbo Pascal introdujo un entorno integrado de programación (IDE) en que se podía editar el código (en un editor compatible con WordStar), ejecutar el compilador, ver los errores, y volver directamente a las líneas que contenían los errores. Ahora suena trivial, pero antes de eso había que salir del editor, volver a MS-DOS, ejecutar el compilador de línea de comandos, anotar las líneas erróneas, abrir de nuevo el editor y buscarlas.

Además, Borland puso a la venta Turbo Pascal por 49 dólares (USA), mientras que el compilador de Pascal de Microsoft estaba a unos cuantos cientos de dólares. Los muchos años de éxito de Turbo Pascal contribuyeron a que Microsoft finalmente retirase su compilador del mercado.

**El Pascal de Delphi**

Tras 9 versiones de compiladores de Turbo Pascal y Borland Pascal, que fueron extendiendo el lenguaje gradualmente, Borland puso a la venta Delphi en 1995, convirtiendo Pascal en un lenguaje de programación visual.

Delphi extiende el lenguaje Pascal de muchas formas, incluyendo muchas extensiones orientadas a objetos que son distintas de otras versiones de Object Pascal, incluidas las del compilador *Borland Pascal with Objects*.

**Capítulo siguiente:** [**Codificación en Pascal**](http://www.marcocantu.com/epascal/Spanish/ch02code.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 2 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch02code.htm) Codificación en Pascal** |

Antes de proseguir con cómo escribir instrucciones en Pascal, es importante resaltar algunos de elementos del estilo de código de Pascal. La cuestión de la que hablo es la siguiente: aparte de las reglas sintácticas, ¿cómo hay que escribir el código? No hay una respuesta única a esta pregunta, ya que el gusto personal puede dictar distintos estilos. De cualquier manera, hay algunos principios que necesitará saber en relación a comentarios, letras mayúsculas, espacios en blanco y la llamada impresión bonita (pretty-printing). En general, la finalidad de cualquier estilo de codificación es la claridad. Las decisiones que usted toma sobre el estilo y el formato son una especie de taquigrafía que indica la finalidad de una porción de código dada. Una herramienta esencial para la claridad es la coherencia -- sea cual sea el estilo que utilice, asegúrese de seguirlo en todo el proyecto.

**Comentarios**

En Pascal, los comentarios se encierran bien entre llaves o paréntesis acompañados por un asterisco. Delphi también entiende los comentarios de estilo C++, que se extienden hasta el final de la línea :

{este es un comentario}

(\* este es otro comentario \*)

// este es un comentario hasta el final de la línea

La primera forma es más breve, y es más común. La segunda forma a menudo se prefiere en Europa, porque muchos teclados europeos carecen del símbolo de llaves. La tercera se tomó prestada de C++, y está disponible sólo en las versiones de Delphi de 32-bits. Los comentarios hasta el final de la línea son muy útiles para comentarios cortos o para evitar que se ejecute una línea (temporalmente).

En los listados de este libro, intentaré marcar los comentarios como cursiva (y palabras clave en negrita), para ser coherente con el resaltado sintáctico por defecto de Delphi.

Que haya tres tipos diferentes de comentarios, puede ser útil para hacer comentarios anidados. Pero si desea evitar que se ejecuten varias líneas de código fuente, y estas líneas contienen algunos comentarios auténticos, no podrá usar el mismo identificador de comentario :

{  ... código

{comentario, que producirá problemas en la ejecución}

... código }

Con un segundo identificador, puede escribir el siguiente código, que es correcto :

{  ... código

//este comentario va bien

... código }

Nótese que si la llave abierta o la cadena paréntesis-asterisco va seguida del signo dólar ($), se convierte en una directiva de compilador, como en {$X+}.

De hecho, directivas de compilador también son comentarios. Por ejemplo, {$X+ Esto es un comentario} es válido. Es a la vez una directiva y un comentario, aunque programadores en su sano juicio probablemente tiendan a separa comentarios y directivas.

**Uso de las mayúsculas**

El compilador de Pascal (contrariamente a los compiladores de otros lenguajes) no distingue entre mayúsculas y minúsculas. Por tanto, interpreta Minombre, MiNombre, minombre, miNombre y MINOMBRE como exactamente equivalentes. En general, esto es bueno, ya que en otros lenguajes se producen muchos errores por grafía de mayúsculas incorrecta.

Nota: sólo hay una excepción a la equivalencia entre mayúsculas y minúsculas en Pascal: el procedimiento *Register* de un paquete de componentes debe comenzar con una *R* mayúscula, por compatibilidad con C++Builder.

Sin embargo, esto conlleva algunas desventajas sutiles. En primer lugar, usted debe ser consciente de que las dos formas de una letra son equivalentes, y no utilizarlas para referirse a elementos distintos. En segundo lugar, debería intentar ser coherente en el uso de mayúsculas y minúsculas, para hacer el código más legible.

Un uso de mayúsculas y minúsculas no es reforzado por el compilador, pero es un buen hábito que se debe adquirir. Una costumbre extendida es poner en mayúsculas sólo la primera letra de los identificadores. Cuando un identificador está formado por varias palabras seguidas (no puede insertar un espacio en un identificador), deberá poner en mayúsculas la primera letra de cada palabra :

MiIdentificadorLargo

MiIdentificadorMuyLargoYCasiEstúpido

Otros elementos que el compilador ignorará completamente son los espacios, las líneas nuevas y los espacios de tabulador. A estos elementos se les llama "espacio en blanco" (*white space*). Un espacio en blanco sólo sirve para hacer el código más legible. No afecta a la compilación.

Al contrario que BASIC, Pascal le deja escribir una instrucción en varias líneas de código, extendiendo una instrucción larga en dos o más líneas. La desventaja (al menos para muchos programadores en BASIC) de permitir esto es que debe acordarse de añadir un punto y coma al final de cada instrucción, o, más exactamente, acordarse de separar las instrucciones entre sí. Dése cuenta de que la única restricción a romper una instrucción es que una cadena de texto debe permanecer en una sola línea.

Tampoco aquí hay reglas fijas para el uso de espacios e instrucciones de varias líneas; sólo algunas reglas básicas :

* El editor Delphi tiene una línea vertical que usted puede situar tras 60 ó 70 caracteres. Si usa esta línea e intenta evitar superar su límite, su código fuente tendrá mejor aspecto cuando sea impreso en papel. Si no, las líneas largas se romperán en cualquier punto, includo a mitad de una palabra, al imprimirlas.
* Cuando una función o procedimiento tiene varios parámetros, es costumbre escribir los parámetros en diferentes líneas.
* Puede dejar una línea vacía antes de un comentario o dividir una porción de código en partes más pequeñas con líneas en blanco. Incluso esta simple idea puede mejorar la legibilidad del código, tanto en la pantalla como cuando lo imprima.
* Use espacios para separar los parámetros de una llamada a una función, y aun un espacio antes del paréntesis inicial. Separe también operandos en una expresión. Sé que algunos programadores estarán en desacuerdo con estas ideas, pero yo insisto: los espacios son libres; no se paga por ellos. (De acuerdo, sé que ocupan espacio en el disco y tiempo de conexión por módem cuando eleva o descarga un archivo a/de la Red, pero esto es cada día menos importante, hoy en día.)

**"Impresión bonita"**

La última sugerencia sobre el uso de espacios en blanco se refiere al típico estilo de formato en Pascal, llamado "impresión bonita" (pretty-printing). Esta regla es simple: cada vez que necesite escribir una instrucción compuesta, haga una sangría de dos espacios a la derecha del resto de dicha instrucción. Una instrucción compuesta dentro de otra instrucción compuesta llevará una sangría de cuatro espacios, y así sucesivamente :

**if** ... **then**

  instrucción;

**if** ... **then**

**begin**

  instrucción1;

  instrucción2;

**end**;

**if** ... **then**

**begin**

**if** ... **then**

    instrucción1;

  instrucción2;

**end**;

El formato que se muestra arriba está basado en "impresión bonita", pero los programadores hacen distintas interpretaciones de esta regla general. Algunos sangran las instrucciones begin y end hasta el nivel del código interno; otros sangran estas instrucciones respecto al if ... then y sangran aún más el código interno; también hay programadores que sitúan la instrucción begin en la misma línea de la instrucción if ... then. Esto suele ser cuestión de gusto.

Un formato de sangría similar se usa a menudo para listas de variables o tipos de datos, y para continuar una instrucción que viene de la línea anterior :

**type**

  Letters = **set of** Char;

**var**

  Name: string;

**begin**

*{ comentario largo e instrucción larga, que sigue en la*

*línea siguiente y está sangrada dos espacios }*

   MessageDlg (*'Esto es un mensaje'*,

     mtInformation, [mbOk], 0);

Por supuesto, cualquier convención de este tipo es sólo una sugerencia para hacer el código más legible a otros programadores, y es totalmente ignorado por el compilador. He intentado utilizar esta regla de forma coherente en todas las muestras y fragmentos de código en este libro. El código fuente Delphi, los manuales y los ejemplos de ayuda (Help) usar un estilo de formato similar.

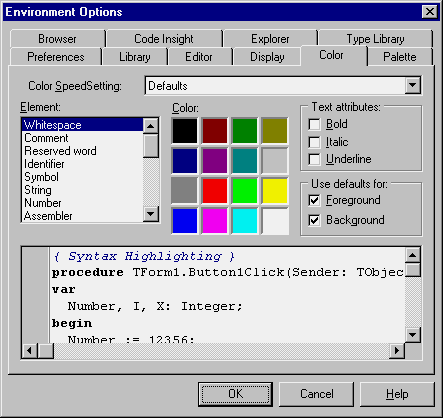
**Resaltado sintáctico**

Para hacer más fácil el leer y escribir código Pascal, el editor Delphi tiene una cualidad llamada *resaltado sintáctico*. Dependiendo de la función en Pascal de las palabras que usted introduce a través del editor, son representadas con distintos colores. Por defecto, las palabras clave están en letra negrita; cadenas de texto y comentarios están en color (y a menudo en cursiva), etcétera.

Las palabras reservadas, los comentarios y las cadenas de texto son, probablemente, los elementos que más se benefician de esta cualidad. De un vistazo puede usted localizar un error ortográfico, una cadena no terminada correctamente y determinar la longitud de un comentario de varias líneas.

Puede personalizar el resaltado sintáctico utilizando la página Editor Colors del cuadro de diálogo Environment Options (opciones del entorno) (v. figura 2.1). Si trabaja solo, elija los colores que prefiera. Si colabora estrechamente con otros programadores, deberían ponerse de acuerdo en un esquema de color común. Encuentro realmente difícil trabajar en una computadora con un esquema sintáctico de colores distinto al que estoy acostumbrado.

***Figura 2.1: El cuadro de diálogo desde el que se establece el resaltado sintáctico de color.***



Nota: En este libro he intentado aplicar un tipo de resaltado a los listados de código fuente. Espero que esto los haga más legibles.

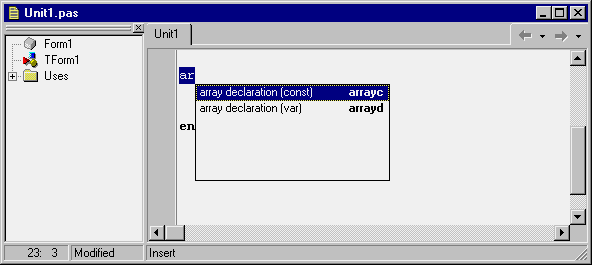
**Uso de Plantillas de Código**

Delphi 3 introdujo una nueva cualidad, relacionada con la edición de código fuente. Como al escribir instrucciones en Pascal a menudo se repite la misma secuencia de palabras clave, Borland ha aportado una nueva cualidad llamada Plantillas de Código (Code Templates). Una Plantilla de Código es simplemente una porción de código relacionada con una abreviatura. Usted escribe la abreviatura, pulsa Ctrl+J, y la porción de código correspondiente aparece completa. Por ejemplo, si usted escribe arrayd y pulsa Ctrl+J, el editor de Delphi expandirá esta abreviatura a :

**array** [0..] **of** ;

Como las plantillas predefinidas suelen incluir varias versiones de la misma construcción, la abreviatura suele terminar en una letra que indica de cuál de las versiones se trata. En cualquier caso, también puede escribir sólo las primeras letras de la abreviatura. Por ejemplo, si escribe ar y pulsa Ctrl+J, el editor mostrará un menú local con una lista de las opciones disponibles con una descripción breve, como se ve en la figura 2.2.

***Figure 2.2: Selección de Plantillas de Código (Code Templates)***



Puede usted personalizar totalmente las plantillas, modificando las existentes, o añadiendo las porciones de código que más use. Si hace esto, dése cuenta de que el texto de una plantilla de código normalmente include el carácter '|' , para indicar a dónde debe saltar el cursor tras la operación, esto es, dónde comenzará usted a escribir para completar el código deseado.

**Instrucciones de lenguaje**

Una vez que haya definido algunos identificadores, puede utilizarlos en instrucciones y en las expresiones que son parte de algunas instrucciones. Pascal ofrece varias instrucciones y expresiones. Ante todo, echemos un vistazo a las palabras clave, las expresiones y los operadores.

**Palabras clave**

Se llama palabras clave (keywords) a todos los identificadores reservados por Object Pascal que tienen un papel en el lenguaje. La ayuda de Delphi distingue entre palabras reservadas y directivas : las palabras reservadas no pueden ser utilizadas como identificadores, mientras que las directivas no deberían ser utilizadas como tales, aunque el compilador las aceptará si se usan así. En la práctica, no debería utilizar ningún tipo de palabras clave como identificadores.

En la Tabla 2.1 puede ver una lista completa de los identificadores que tienen un papel específico en Object Pascal (en Delphi 4), incluyendo palabras clave y otras palabras reservadas.

***Tabla 2.1: Palabras clave y otras palabras reservadas en Object Pascal.***

|  |  |
| --- | --- |
| **Palabra clave** | **Papel** |
| absolute | directiva (variables) |
| abstract | directiva (método) |
| and | operador (booleano) |
| array | tipo |
| as | operador (RTTI) |
| asm | instrucción |
| assembler | retrocompatibilidad (asm) |
| at | instrucción (excepciones) |
| automated | especificador de acceso (clase) |
| begin | marcador de bloque |
| case | instrucción |
| cdecl | convención de llamada a una función |
| class | tipo |
| const | declaración o directiva (parámetros) |
| constructor | método especial |
| contains | operador (conjunto) |
| default | directiva (propiedad) |
| destructor | método especial |
| dispid | especificador dispinterfaz |
| dispinterface | tipo |
| div | operador |
| do | instrucción |
| downto | instrucción (for ... then) |
| dynamic | directiva (método) |
| else | instrucción (if or case) |
| end | marcador de bloque |
| except | instrucción (excepciones) |
| export | retrocompatibilidad (clase) |
| exports | declaración |
| external | directiva (funciones) |
| far | retrocompatibilidad (clase) |
| file | tipo |
| finalization | estructura unitaria |
| finally | instrucción (excepciones) |
| for | instrucción (v. to) |
| forward | directiva de función |
| function | declaración |
| goto | instrucción |
| if | instrucción (v. then) |
| implementation | estructura unitaria |
| implements | directiva (propiedad) |
| in | operador (conjunto) - estructura de proyecto |
| index | directiva (dipinterfaz) |
| inherited | instrucción |
| initialization | estructura unitaria |
| inline | retrocompatibilidad (véase asm) |
| interface | tipo |
| is | operador (RTTI) |
| label | declaración |
| library | estructura de programa |
| message | directiva (método) |
| mod | operador (matemático) |
| name | directiva (función) |
| near | retrocompatibilidad (clase) |
| nil | valor |
| nodefault | directiva (propiedad) |
| not | operador (booleano) |
| object | retrocompatibilidad (clase) |
| of | instrucción (case) |
| on | instrucción (excepciones) |
| or | operador (booleano) |
| out | directiva (parámetros) |
| overload | directiva de función |
| override | directiva de función |
| package | estructura de programa (paquete) |
| packed | directiva (registro) |
| pascal | convención de llamada a una función |
| private | especificador de acceso (clase) |
| procedure | declaración |
| program | estructura de programa |
| property | declaración |
| protected | especificador de acceso (clase) |
| public | especificador de acceso (clase) |
| published | especificador de acceso (clase) |
| raise | instrucción (excepciones) |
| read | especificador de propiedad |
| readonly | especificador de interfaz de envío |
| record | tipo |
| register | convención de llamada a una función |
| reintroduce | directiva de función |
| repeat | instrucción |
| requires | estructura de programa (paquete) |
| resident | directiva (funciones) |
| resourcestring | tipo |
| safecall | convención de llamada a una función |
| set | tipo |
| shl | operador (matemático) |
| shr | operador (matemático) |
| stdcall | convención de llamada a una función |
| stored | directiva (propiedad) |
| string | tipo |
| then | instrucción (v. if) |
| threadvar | declaración |
| to | instrucción (v. for) |
| try | instrucción (excepciones) |
| type | declaración |
| unit | estructura unitaria |
| until | instrucción |
| uses | estructura unitaria |
| var | declaración |
| virtual | directiva (método) |
| while | instrucción |
| with | instrucción |
| write | especificador de propiedad |
| writeonly | especificador de interfaz de envío |
| xor | operador (booleano) |

**Expresiones y Operadores**

No hay una regla general para construir expresiones, ya que dependen esencialmente de qué operadores se usan, y Pascal tiene muchos operadores. Hay operadores lógicos, aritméticos, relacionales, de conjuntos y otros. Se pueden utilizar expresiones para determinar el valor que se asigna a una variable para calcular el parámetro de una función o procedimiento, o para probar una condición. Las expresiones pueden incluir también llamadas a funciones. Cada vez que ejecuta una operación sobre el valor de un identificador, más que utilizar el identificador como tal, eso es una expresión.

Las expresiones son comunes a la mayoría de lenguajes de programación. Una expresión es una combinación válida de constantes, variables, valores *literales* (de caracteres), operadores y resultados de funciones. Las expresiones también pueden ser introducidas en los parámetros de valor de procedimientos y funciones, pero no siempre en los parámetros de referencia (que requieren que se les asigne un valor).

**Operadores y precedencia**

Si alguna vez en su vida ha escrito un programa, ya sabe qué es una expresión. Aquí resaltaré elementos específicos de los operadores de Pascal. Puede ver una lista de los operadores del lenguaje, agrupador por precedencia, en la Tabla 2.1.

Al contrario que la mayoría de los demás lenguajes de programación, los operadores and y or tienen prioridad sobre el operador de relación. Así, si usted escribe a < b and c < d, el compilador intentará hacer la operación and primero, lo que resultará en un error de compilación. Por esta razón, debería encerrar cada una de las expresiones relacionales entre paréntesis : (a < b) and (c < d).

Algunos de los operadores comunes tienen distintos significados junto a diferentes tipos de datos. Por ejemplo, el operador + puede usarse para sumar dos números, concatenar dos cadenas de caracteres, unir dos conjuntos e incluso añadir un *offset* a un puntero (*pointer*) PChar. De cualquier manera, no se puede sumar dos caracteres, como sí puede hacerse en C.

Otro operador extraño es div. En Pascal se puede dividir cualesquiera números (reales o enteros) con el operador / , y siempre se obtiene un número real. Si necesita obtener un entero como resultado de la división de dos enteros (ignorando el resto aunque sea distinto de cero), use el operador div.

***Tabla 2.2: Operadores en Pascal, agrupados por precedencia***

|  |  |
| --- | --- |
| **Operadores unarios (precedencia máxima)** | |
| @ | Dirección de la variable o función (devuelve un puntero) |
| not | not (negación) booleana o por bits |
| **Operadores multiplicativos o por bits** | |
| \* | Multiplicación aritmética o intersección de conjuntos |
| / | División de coma flotante |
| div | División entera |
| mod | Módulo (el resto de la división entera) |
| as | *Typecast* (moldeador) a prueba de tipos (RTTI) (Permite una conversión de tipo durante la ejecución, con comprobación de tipo previa.) |
| and | and (conjunción) booleana o por bits |
| shl | Bitwise left shift (Permite rotar una cadena de bits hacia la izquierda.) |
| shr | Bitwise right shift (Ídem, hacia la derecha.) |
| **Operadores aditivos** | |
| + | Suma aritmética, unión de conjuntos, concatenación de cadenas, suma de punteros *offset* |
| - | Resta aritmética, diferencia de conjuntos, resta de punteros *offset* |
| or | or (disyunción) booleana o por bits |
| xor | xor (disyunción exclusiva) booleana o por bits |
| **Operadores relacionales y de comparación (precedencia mínima)** | |
| = | Comprueba si son iguales |
| <> | Comprueba si no son iguales |
| < | Comprueba si es menor que |
| > | Comprueba si es mayor que |
| <= | Comprueba si es menor o igual que, o si es subconjunto de un conjunto |
| >= | Comprueba si es mayor o igual que, o si es "superconjunto" de un conjunto |
| in | Comprueba si pertenece al conjunto |
| is | Comprueba si es compatible respecto al tipo (otro operador RTTI) |

**Operadores de conjuntos**

Los operadores de conjuntos incluyen la unión, (+), la diferencia (-), la intersección (\*), comprobación de pertenencia (in), y algunos operadores relacionales. Para añadir un elemento a un conjunto, puede unir el conjunto a otro que ya tenga el elemento que necesita. Aquí hay un ejemplo de Delphi, relacionado con los estilos de fuente :

Style := Style + [fsBold];

Style := Style + [fsBold, fsItalic] - [fsUnderline];

Como alternativa, puede usar los procedimientos normalizados Include y Exclude, que son mucho más eficientes (pero no se pueden utilizar con propiedades de componente del tipo de conjunto, que requieren un parámetro de un único valor):

Include (Style, fsBold);

**Conclusión**

Ahora que conocemos la estructura básica de un programa en Pascal, estamos preparados para comenzar a comprender su significado en detalle. Comenzaremos explorando la definición de tipos de datos predefinidos y definidos por el usuario. Después, pasaremos a estudiar el uso de palabras clave para formar instrucciones de programación.

**Capítulo siguiente:** [**Tipos, Variables y Constantes**](http://www.marcocantu.com/epascal/Spanish/ch03data.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 3 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch03data.htm) Tipos, variables y constantes** |

El lenguaje Pascal original estaba basado en algunas nociones simples, que ahora se han hecho bastante comunes entre los lenguajes de programación. La primera noción es la de *tipo de datos*. El tipo determina los valores que puede tomar una variable, y las operaciones que se pueden realizar sobre ella. El concepto de tipo es más fuerte en Pascal que en C, donde los tipos de datos aritméticos son casi intercambiables, y mucho más fuertes que en las versiones originales de BASIC, donde no existía concepto similar alguno.

**Variables**

Pascal requiere que todas las variables sean declaradas antes de ser utilizadas. Cada vez que declare una variable, deberá especificar un tipo de dato. Siguen algunas declaraciones :

**var**

  Valor: Integer;

  EsCorrecto: Boolean;

  A, B: Char;

La palabra clave var puede utilizarse en varios lugares del código, como el principio del código de una función o procedimiento, para declarar variables que son locales en la rutina, o dentro de una unidad, para declarar variables globales. Después de la palabra var viene una lista de nombres de nuevas variables, seguidas de dos puntos y el nombre del tipo de dato. Puede escribir más de una variable en una misma línea, como en la última instrucción del ejemplo anterior.

Una vez que haya definido la variable de un tipo determinado, puede realizar sobre ella las operaciones permitidas para aquel tipo de dato. Por ejemplo, se puede utilizar valores booleanos para comprobaciones (sí/no) y valores enteros en una expresión numérica. Sin embargo, no se puede mezclar booleanos con enteros (como sí se puede en el lenguaje C).

Con asignaciones simples, podemos escribir código como el siguiente :

Valor := 10;

EsCorrecto := True;

Pero la siguiente instrucción no es válida, porque las variables implicadas son de tipos de dato distintos :

Valor := EsCorrecto; *// error*

Si intenta compilar este código, Delphi produce un error con la descripción : *Incompatible types: 'Integer' and 'Boolean'*. Normalmente, errores como estos son errores de programación, porque no tiene sentido asignar un valor True (verdadero) o False (falso) a una variable de tipo entero. No le eche la culpa a Delphi de estos errores. Sólo le está advirtiendo de que hay algo equivocado en el código.

Por supuesto, a menudo es posible convertir el valor de una variable de un tipo a otro. En algunos casos, esta conversión es automática, pero en general tendrá que hacer una llamada a una función específica del sistema que cambia la representación interna de los datos.

En Delphi puede asignar un valor inicial a una variable global cuando la declara. Por ejemplo :

**var**

  Valor: Integer = 10;

  Correcto: Boolean = True;

Esta técnica de inicialización funciona sólo en variables globales, no en variables definidas en el interior de un procedimiento o método.

**Constantes**

Pascal también permite la declaración de constantes, para nombrar valores que no cambian durante la ejecución del programa. Para declarar una constante, no necesita especificar un tipo de dato, sino sólo asignar un valor inicial. El compilador examinará el valor y usará el tipo de dato adecuado, automáticamente. Siguen algunos ejemplos :

**const**

  Millar = 1000;

  Pi = 3.14;

  NombreAutor = *'Marco Cantù'*;

Delphi determina el tipo de dato de la constante de acuerdo con su valor. En el ejemplo anterior, la constante Millar se supone que es del tipo SmallInt, el tipo de entero más pequeño en que cabe el valor 1000. Sin embargo, si quiere usted decirle a Delphi que use un tipo específico, puede incluir el nombre del tipo en la declaración :

**const**

  Millar: Integer = 1000;

Cuando usted declara una constante, el compilador puede elegir entre asignarle una posición en la memoria y guardar su valor allí, o duplicar el valor actual cada vez que se usa la constante. Este segundo modo de funcionamiento tiene sentido especialmente en el uso de constantes simples :

Nota: La versión en 16 bits de Delphi permite modificar el valor de una constante durante la ejecución, como si fuera una variable. La versión en 32 bits aún permite este comportamiento para garantizar la retrocompatibilidad, cuando se habilita la directiva *$J* del compilador o se activa la casilla *Assignable typed constants* de la página Compiler del cuadro de diálogo Project Options. Aunque este es el valor por defecto, se aconseja enérgicamente no utilizar este truco como técnica de programación habitual. Asignar un nuevo valor a una constante desactiva todas las optimizaciones previstas en el compilador para el manejo de constantes. En tales casos, decántese por una variable.

**Constantes de cadena de recursos (*Resource String Constants*)**

Cuando defina una constante de cadena, en vez de escribir :

**const**

  AuthorName = *'Marco Cantù'*;

desde Delphi 3 puede escribir :

**resourcestring**

  AuthorName = *'Marco Cantù'*;

En ambos casos se está definiendo una constante, esto es, un valor que no va a cambiar durante la ejecución del programa. La diferencia consiste en la implementación. Una constante de cadena definida con la directiva *resourcestring* se almacena en los recursos del programa, en una tabla de cadenas de caracteres.

Para ver cómo funciona esto, puede examinar el ejemplo ResStr example, que tiene un botón con el siguiente código :

**resourcestring**

  AuthorName = *'Marco Cantù'*;

  BookName = *'Essential Pascal'*;

**procedure** TForm1.Button1Click(Sender: TObject);

**begin**

  ShowMessage (BookName + #13 + AuthorName);

**end**;

Las salidas de las dos cadenas aparecen en líneas separadas, porque las cadenas van separadas por el carácter de nueva línea (indicado por su valor numérico *#13* como constante de tipo carácter).

El aspecto interesante de este programa es que si lo examina con un explorador de recursos (hay uno disponible entre los ejemplos que se suministran con Delphi), verá las nuevas cadenas entre los recursos. Esto significa que las cadenas no son parte del código compilado, sino que son almacenadas en una área separada del ejecutable (el archivo exe).

Nota: En resumen, la ventaja de los recursos es un manejo eficiente de la memoria, realizado por Windows, y también la posibilidad de localizar un programa (traduciendo las cadenas a otro lenguaje) sin tener que modificar su código fuente.

**Tipos de datos**

En Pascal hay varios tipos de datos predefinidos, que pueden ser divididos en tres grupos: *tipos ordinales*, *tipos reales* y *cadenas*. Trataremos sobre los tipos ordinal y real en las próximas secciones, mientras que las cadenas se mencionarán más adelante en este mismo capítulo. En esta sección también introduciré algunos tipos definidos por las bibliotecas de Delphi (no predefinidas por el compilador), que pueden considerarse tipos predefinidos.

Delphi también incluye un tipo de datos sin tipo, llamado *variante*, que discutiremos el [**Capítulo 10**](http://www.marcocantu.com/epascal/Spanish/ch10var.htm) de este libro). Resulta extraño que exista el variante, que no pasa por una comprobación de tipo lo bastante fuerte. Fue introducido en Delphi 2 para poder manejar la automatización OLE.

**Tipos ordinales**

Los tipos ordinales están basados en el concepto de orden o secuencia. Usted no sólo puede comparar dos valores para ver cuál es más alto, sino que también puede pedir el valor que sigue o precede a un valor dado, o calcular el valor más bajo o alto de los posibles.

Los tipos ordinales predefinidos más importantes son entero (Integer), booleano (Boolean) y carácter (Char[acter]). De cualquier manera, hay muchos tipos relacionados que tienen el mismo significado, pero una representación interna y un rango de valores distintos. La tabla 3.1 que sigue muestra los tipos de datos ordinales utilizados para representar números.

***Tabla 3.1: Tipos de datos ordinales para números***

|  |  |  |
| --- | --- | --- |
| **Tamaño** | **Rango con signo (positivo / negativo)** | **Rango sin signo** |
| 8 bits | ShortInt (entero corto)  de -128 a 127 | Byte  de 0 a 255 |
| 16 bits | SmallInt (entero pequeño)  de -32.768 a 32.767 | Word (palabra)  de 0 a 65.535 |
| 32 bits | LongInt (entero largo)  de -2.147.483,648 a 2.147.483.647 | LongWord (palabra larga) (desde Delphi 4)  de 0 a 4.294,967,295 |
| 64 bits | Int64 (entero de 64 bits) |  |
| 16/32 bits | Integer (entero) | Cardinal |

Como puede ver, estos tipos corresponden a diferentes representaciones de números, dependiendo del número de bits usado para expresar el valor, y la presencia o ausencia de un bit de signo. Los valores con signo pueden ser positivos o negativos, pero tienen un rango más estrecho de valores, porque hay un bit menos disponible para el valor en sí. Puede remitirse al ejemplo sobre el Rango, discutido en la sección siguiente, para averiguar el rango de valores que corresponden a cada tipo.

El último grupo (marcado como 16/32) indica valores que tienen una representación distinta en las versiones de 16 bits y 32 bits de Delphi. Los tipos Integer y Cardinal son usados frecuentemente, porque corresponden a la representación nativa de números en la CPU.

**Tipos enteros en Delphi 4**

En Delphi 3, los números sin signo de 32 bits indicados por el tipo Cardinal eran, en realidad, valores de 31 bits, con un rango de hasta 2 megabytes. Delphi 4 introdujo un nuevo tipo numérico sin signo, LongWord, que usa un valor de 32 bits auténtico, de hasta 4 megabytes. El tipo Cardinal es ahora un alias del nuevo tipo LongWord. LongWord permite acceder a 2 MB de datos más a través de un número sin signo, como se menciona arriba. Además, corresponde a la representación nativa de números en la CPU.

Otro tipo introducido por primera vez en Delphi 4 es el tipo Int64, que representa números enteros de hasta 18 dígitos. Este nuevo tipo es totalmente compatible con algunas de las rutinas de tipo ordinal (como High y Low), rutinas numéricas (como Inc y Dec), y rutinas de conversión de cadena (como IntToStr). Para la conversión contraria, de cadena a número, hay dos funciones específicas : StrToInt64 y StrToInt64Def.

**Booleanos**

Valores booleanos distintos del tipo Boolean se utilizan muy poco. Las funciones API de Windows requieren algunos valores booleanos con representaciones específicas. Los tipos son ByteBool, WordBool y LongBool.

En Delphi 3, para conseguir compatibilidad con Visual Basic y automatización OLE, los tipos de datos *ByteBool*, *WordBool* y *LongBool* fueron modificados para representar el valor *True* con -1, mientras que el valor *False* es aún 0. El tipo de datos *Boolean* se mantiene invariable (*True* es 1, *False* es 0). Si usted ha utilizado *typecasts* (moldeadores de tipo) específicos en su código Delphi 2, portar el código a versiones posteriores de Delphi puede causarle errores.

**Caracteres**

Finalmente, hay dos representaciones de caracteres : *ANSIChar* y *WideChar*. El primer tipo representa caracteres de 8-bits, correspondientes al conjunto de caracteres ANSI usado tradicionalmente por Windows; el segundo representa caracteres de 16 bits, que corresponden a los nuevos caracteres Unicode accesibles en Windows NT, y sólo parcialmente por Windows 95 y 98. La mayor parte del tiempo, usted sólo utilizará el tipo *Char*, que en Delphi 3 corresponde a *ANSIChar*. Tenga presente, en cualquier caso, que los 256 primeros caracteres Unicode corresponden exactamente a los caracteres ANSI.

Los caracteres constantes pueden ser representados con su notación simbólica, p. ej. *'k'*, o con una notación numérica, como en *#78*. Este último también puede ser representado utilizando la función *Chr*, como en *Chr (78)*. La conversión recíproca puede hacerse mediante la función *Ord*.

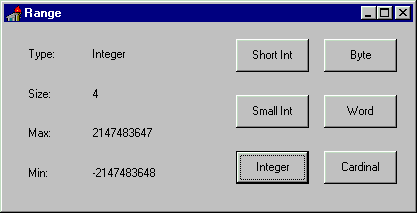
En general es mejor utilizar la notación simbólica cuando se indican letras, dígitos o símbolos. Para referirse a caracteres especiales, sin embargo, normalmente usará la notación numérica. La siguiente lista incluye algunos de los caracteres especiales más usados :

* *#9* tabulador
* *#10* nueva línea
* *#13* retorno de carro (tecla "*enter*" o "intro")

**El ejemplo sobre el Rango**

Para darle una idea de los distintos rangos de algunos de los tipos ordinales, he escrito un programa Delphi sencillo llamado Range. Algunos resultados se muestran en la figura 3.1.

Figura 3.1: El ejemplo *Range* muestra alguna información sobre tipos de datos ordinales (en este caso, del tipo *Integer*).



El programa *Range* está basado en un formulario sencillo, que tiene seis botones (cada uno lleva el nombre de un tipo de datos ordinal), y algunas etiquetas sobre categorías de información, como se ve en la figura. Algunas de las etiquetas se usan para incluir texto estático, y otras para mostrar la información sobre el tipo cada vez que se pulsa uno de los botones.

Cada vez que el usuario pulsa uno de los botones de la derecha, el programa actualiza las etiquetas como salida. Distintas etiquetas muestran el tipo de dato, el número de bytes utilizado y los valores máximo y mínimo que puede almacenar ese tipo de datos. Cada botón tiene su propio método de respuesta a eventos tipo *OnClick*, porque el código usado para calcular los tres valores varía ligeramente de un botón a otro. Por ejemplo, aquí está el código fuente del evento *OnClick* para el botón Integer (*BtnInteger*):

**procedure** TFormRange.BtnIntegerClick(Sender: TObject);

**begin**

  LabelType.Caption := *'Integer'*;

  LabelSize.Caption := IntToStr (SizeOf (Integer));

  LabelMax.Caption := IntToStr (High (Integer));

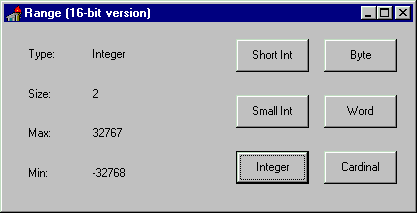
  LabelMin.Caption := IntToStr (Low (Integer));

**end**;

Si tiene algo de experiencia en la programación en Delphi, puede examinar el código fuente del programa para entender cómo funciona. Para principiantes, es suficiente con darse cuenta del uso de tres funciones : *SizeOf* (tamaño de), *High* (alto) y *Low* (bajo). Los resultados de las dos últimas funciones son ordinales del mismo tipo (en este caso, enteros), y el resultado de la función *SizeOf* es siempre un entero. El valor de salida de cada una de estas funciones se traduce primero en cadenas utilizando la función *IntToStr*, y luego se copia en los títulos de las tres etiquetas.

Los métodos asociados con los otros botones son muy similares al mencionado arriba. La única diferencia radica en el tipo de dato que se pasa como parámetro a cada una de las funciones. La figura 3.2 muestra el resultado de ejecutar este mismo programa bajo Windows 95 después de ser recompilado con la versión de 16 bits de Delphi. Comparando la figura 3.1 con la figura 3.2, puede ver la diferencia entre tipos de entero de 16 bits y 32 bits.

***Figure 3.2: La salida de la versión de 16 bits del programa Range, mostrando información sobre enteros, nuevamente.***



El tamaño del tipo *Integer* varía dependiendo de la CPU y el sistema operativo utilizados. En versiones Windows de 16 bits, una variable entera ocupa dos bytes. En versiones de Windows de 32 bits, un Integer ocupa 4 bytes. Por esta razón, cuando usted recompila el programa *Range*, obtiene una salida distinta en cada caso.

Las dos representaciones distintas del tipo *Integer* no son un problema, siempre que su programa no asuma nada sobre el tamaño de los enteros. Si alguna vez guarda un Integer en un archivo utilizando una versión y lo carga con otra, tendrá algunos problemas. En esta situación, debería elegir un tipo de datos independiente de la plataforma (como *LongInt* o *SmallInt*). Para cálculos matemáticos o código genérico, la mejor elección es seguir la representación de enteros de la plataforma en que se quiera trabajar -- esto es, utilice el tipo Integer -- porque esto es lo que más le gusta a la CPU. El tipo *Integer* debería ser su primera elección cuando maneje números enteros. Use una representación distinta sólo si hay una razón imperiosa para hacerlo.

**Rutinas de tipo ordinal**

Hay algunas rutinas del sistema (rutinas definidas en el lenguaje Pascal y la unidad de sistema de Delphi) que funcionan en tipos ordinales. Se muestran en la tabla 3.2. Los programadores en C++ no deben olvidar de que las dos versiones del procedimiento *Inc*, con uno o dos parámetros, corresponden a los operadores ++ y += (y lo mismo para el procedimiento *Dec*).

***Table 3.2: Rutinas del sistema para tipos ordinales***

|  |  |
| --- | --- |
| **Rutina** | **Finalidad** |
| Dec | Decrementa la variable pasada como parámetro, en una unidad o por el valor del parámetro segundo, opcional. |
| Inc | Incrementa la variable pasada como parámetro, en una unidad o por el valor especificado. |
| Odd | Devuelve *True* si el argumento es un número par. |
| Pred | Devuelve el valor precedente al argumento según el orden determinado por el tipo de datos. |
| Succ | Devuelve el valor siguiente (*successor*) al argumento según el orden determinado. |
| Ord | Devuelve un número, indicando el orden del argumento dentro del conjunto de valores del tipo de datos. |
| Low | Devuelve el menor valor en el rango del tipo ordinal pasado como parámetro. |
| High | Devuelve el menor valor en el rango del tipo ordinal. |

Observe que algunas de estas rutinas, cuando se aplican a constantes, son evaluadas automáticamente por el compilados y reemplazadas por su valor. Por ejemplo, si hace una llamada a la función *High(X)*, donde *X* se define como Integer, el compilador puede reemplazar la expresión con el valor más alto posible en el tipo de datos Integer.

**Tipos reales**

Los tipos reales representan números de coma flotante en varios formatos. El menor tamaño de almacenamiento corresponde a los números *Single*, implementados con un valor de 4 bytes. Aparte hay números de coma flotante *Double*, implemententados con 8 bytes, y números *Extended*, implementados con 10 bytes. Todos estos son tipos de datos de coma flotante de distinta precisión, que corresponden a las representaciones normalizadas IEEE, y son compatibles directamente por el coprocesador numérico de la CPU, lo que permite una velocidad máxima.

En Delphi 2 y Delphi 3 el tipo Real tenía la misma definición que en la versión de 16 bits; era un tipo de 48 bits. Pero su uso fue reprobado por Borland, que sugirió que se usasen los tipos Single, Double y Extended, en vez de aquel. Las razón era que el viejo formato de 6 bits no era compatible ni con las CPUs de Intel ni estaba entre los tipos reales oficiales IEEE. Para superar el problema totalmente, Delphi 4 modifica la definición del tipo Real para representar un número de coma flotante normalizado de 8 bytes (64 bits).

Además de la ventaja de usar una definición normalizada, este cambio permite a los componentes publicar propiedades basadas e el tipo Real, algo que Delphi 3 no permitía. Entre las desventajas podría haber problemas de compatibilidad. De ser necesario, puede evitar la eventual incompatibilidad utilizando la definición del tipo de Delphi 2 o 3; haga esto usando la siguiente opción de compilación :

{$REALCOMPATIBILITY ON}

Hay también dos tipos de datos extraños: *Comp* describe enteros muy grandes que usan 8 bytes (que pueden contener números de hasta 18 dígitos decimales); y *Currency* (no disponible en el Delphi de 16 bits) indica un valor decimal de coma fija con cuatro dígitos decimales, y la misma representación en 64 bits que el tipo *Comp*. Como implica el nombre, el tipo de datos *Currency* ha sido añadido para manejar valores monetarios muy precisos, con cuatro cifras decimales.

No podemos crear un programa similar el ejemplo Range de arriba con tipos de datos reales, porque no podemos usar las funciones *High* y *Low* o la función *Ord* en variables reales. Los tipos reales representan (en teoría) un conjunto de números infinito; los tipos ordinales representan un conjunto fijo de valores.

Nota: Déjeme explicar esto mejor. Cuando tiene el entero 23, puede determinar cuál es el valor siguiente. Los enteros son finitos (tienen un rango determinado y tienen un orden). Los números de coma flotante son infinitos incluso dentro de un pequeño, y no tienen orden : de hecho, ¿cuántos valores hay entre 23 y 24? Y ¿qué número sigue al 23.46? Es 23.47, 23.461 o 23.4601 ? ¡Es difícil decidirlo!

Por esta razón, tiene sentido pedir la posición ordinal del carácter '*w*' en el rango del tipo de datos Char, pero no tiene ningún sentido preguntar por lo mismo acerca de 7143,1562 [aquí la coma es decimal] en el rango de un tipo de datos de coma flotante. Aunque, de hecho, puede usted saber si un número real tiene un valor superior a otro, no tiene sentido preguntar cuántos números reales existen antes de uno dado (este es el significado de la función *Ord*).

Los tipos reales juegan un papel pequeño en la parte del código dedicada a la interfaz de usuario (parte dedicada a Windows), pero son perfectamente implementables en Delphi, incluida la parte que toca a las bases de datos. La compatibilidad con tipos de coma flotante de la norma IEEE hace al lenguaje de programación Object Pascal muy apropiado para el amplio abanico de programas que requieren cálculos numéricos. Si está usted interesado en este aspecto, puede buscar las funciones aritméticas incorporadas a Delphi en la unidad de sistema (véase la ayuda de Delphi para más detalles).

**Nota:** Delphi también tiene una unidad *Math* que define rutinas matemáticas avanzadas, que cubren las funciones trigonométricas (como la función *ArcCosh*), finanzas (como la función *InterestPayment*) y estadística (como el procedimiento *MeanAndStdDev*). Hay muchas rutinas así, algunas de las cuales suenan bastante raras, como el procedimiento *MomentSkewKurtosis* (le dejo como ejercicio averiguar de qué se trata ...).

**Fecha y Hora**

Delphi usa tipos reales también para manejar información sobre la fecha y la hora. Para ser más precisos, Delphi define un tipo de datos específico, *TDateTime*. Este es un tipo de coma flotante, porque el tipo tiene que ser lo bastante amplio para almacenar ños, meses, días, horas, minutos y segundos, hasta resolución de milisegundos, en una sola variable. Las fechas se almacenan como número de días desde el 30 de diciembre de 1899 (12/30/1899), donde los valores negativos indican fechas anteriores al año 1899, en la parte entera del valor de *TDateTime*. La hora se almacena como fracciones de un días en la parte decimal del valor.

*TDateTime* no es un tipo predefinido que el compilador comprende, sino que es definido en la unidad de sistema como :

**type**

  TDateTime = **type** Double;

Usar el tipo *TDateTime* es bastante fácil, porque Delphi incluye varias funciones que operan sobre este tipo. Puede encontrar una lista de estas funciones en la tabla 3.3.

***Tabla 3.3: Rutinas del sistema para el tipo TDateTime***

|  |  |
| --- | --- |
| **Rutina** | **Descripción** |
| Now | Devuelve la fecha y la hora actual en un solo valor TDateTime. |
| Date | Devuelve sólo la fecha actual. |
| Time | Devuelve sólo la hora actual. |
| DateTimeToStr | Convierte un valor de fecha y hora en una cadena, utilizando formato por defecto; para tener mayor control sobre la conversión, utilice la función FormatDateTime. |
| DateTimeToString | Copia el valor de fecha y hora en un *buffer* (tampón) de cadena, en formato por defecto. |
| DateToStr | Convierte la porción de fecha de un valor TDateTime en una cadena. |
| TimeToStr | Convierte la porción de hora de un valor TDateTime en una cadena. |
| FormatDateTime | Da un formato especificado a fecha y hora; puede especificar qué valores quiere ver y qué formato utilizar, proveyendo una cadena de formato complejo. |
| StrToDateTime | Convierte una cadena con información de fecha y hora en un valor TDateTime, provocando una excepción en caso de error en el formato de la cadena. |
| StrToDate | Convierte una cadena con información de fecha en un valor TDateTime. |
| StrToTime | Convierte una cadena con información de hora en un valor TDateTime |
| DayOfWeek | Extrae el número correspondiente al día de la semana del valor TDateTime que se pasa como parámetro. |
| DecodeDate | Estrae los valores de año, mes y día de un valor de fecha. |
| DecodeTime | Extrae los valores de un valor de hora. |
| EncodeDate | Convierte valores de año, mes y día en un valor TDateTime. |
| EncodeTime | Convierte valores de hora, minuto, segundo y milisegundo en un valor TDateTime. |

Para mostrarle cómo usar este tipo de datos y algunas de las rutinas relacionadas con él, he construido un ejemplo sencillo, llamado TimeNow. El formulario principal de este ejemplo tiene un componente de botón (Button) y de lista (ListBox). Cuando el programa se inicia, se calcula y muestra la fecha y hora actual, automáticamente. Cada vez que se pulsa el botón, el programa muestra el tiempo pasado desde que el programa comenzó.

Este es el código relacionado con el evento *OnCreate* del formulario :

**procedure** TFormTimeNow.FormCreate(Sender: TObject);

**begin**

  StartTime := Now;

  ListBox1.Items.Add (TimeToStr (StartTime));

  ListBox1.Items.Add (DateToStr (StartTime));

  ListBox1.Items.Add (*'Pulse botón para ver el tiempo transcurrido'*);

**end**;

La primera instrucción es una llamada a la función *Now*, que devuelve la fecha y hora actuales. Este valor se almacena en la variable *StartTime*, declarada como variable global, como sigue :

**var**

  FormTimeNow: TFormTimeNow;

  StartTime: TDateTime;

He añadido sólo la segunda declaración, ya que la primera viene dada por Delphi. Por defecto, es la siguiente :

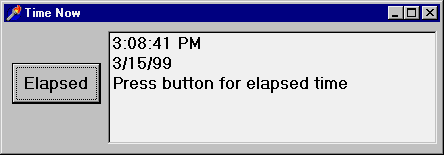
**var**

  Form1: TForm1;

Al cambiar el nombre del formulario, esta declaración se actualiza automáticamente. Utilizando variables globales no es realmente el mejor enfoque : sería mejor utilizar un campo privado de la clase del formulario, un asunto relacionado con la programación orientada a objetos y discutido en *Mastering Delphi 4* [publ. en español como *Delphi 4*].

Las siguientes tres instrucciones añaden tres entradas al componente ListBox a la izquierda del formulario, con el resultado que se ve en la figura 3.3. La primera línea contiene la porción de tiempo del valor *TDateTime* convertido en una cadena, el segundo la porción de la fecha, del mismo valor. Al final, el código añade un simple mensaje recordatorio.

***Figura 3.3: La salida del programa de ejemplo TimeNow al iniciarse.***



Esta tercera cadena es reemplazada por el programa cuando el usuario pincha en el botón Elapsed :

**procedure** TFormTimeNow.ButtonElapsedClick(Sender: TObject);

**var**

  StopTime: TDateTime;

**begin**

  StopTime := Now;

  ListBox1.Items [2] :=  FormatDateTime (*'hh:nn:ss'*,

    StopTime - StartTime);

**end**;

Este código extrae la hora nueva y calcula la diferencia con el valor de hora almacenado cuando el programa se inició. Ya que necesitamos utilizar un valor que calculamos en un gestor de eventos distinto, tuvimos que almacenarlo en una variable global. De hecho, hay mejores alternativas, basadas en clases.

**Nota:** El código que reemplaza el valor acutal de la tercera cadena, utiliza el índice 2. La razón es que las entradas de la lista ListBox se cuentan desde cero : la primera es la número 0, la segunda la número 1, y la tercera la número 2. Comentaremos más sobre esto cuando hablemos de matrices.

Aparte de hacer una llamada a *TimeToStr* y *DateToStr*, puede utilizar la función *FormatDateTime*, más potente, como lo he hecho en el último método, arriba (véase la ayuda de Delphi para saber cómo se da formato a los parámetros). Tenga también en cuenta que los valores de hora y fecha se transforman en cadenas, dependiendo de la configuración regional de su sistema Windows. Delphi lee estos valores del sistema, y los copias a una serie de constantes globales declaradas en la unidad SysUtils. Algunas de ellas son :

DateSeparator: Char;

ShortDateFormat: string;

LongDateFormat: string;

TimeSeparator: Char;

TimeAMString: string;

TimePMString: string;

ShortTimeFormat: string;

LongTimeFormat: string;

ShortMonthNames: array [1..12] **of** string;

LongMonthNames: array [1..12] **of** string;

ShortDayNames: array [1..7] **of** string;

LongDayNames: array [1..7] **of** string;

Otras constantes globales están relacionadas al formato de moneda y de números de coma flotante (coma decimal española contra punto decimal inglés). Puede encontrar la lista completa en el archivo de ayuda de Delphi bajo el título *Currency and date/time formatting variables*.

**Nota:** Delphi incluye un componente DateTimePicker, que prové un modo sofisticado de entrar una fecha, seleccionándola de un calendario.

**Tipos específicos de Windows**

Los tipos de datos predefinidos que hemos visto hasta ahora son parte del lenguaje Pascal. Delphi también incluye otros tipos de datos definidos por Windows. Estos tipos de datos no son parte integrante del lenguaje, pero son parte de las bibliotecas de Windows. Entre los tipos de Windows hay nuevos tipos por defecto (como *DWord* o *UInt*), muchos registros (o estructuras), varios tipos de puntero (*pointer*), etcétera.

De los tipos de datos de Windows, el más importante se representa con *handles*, que se comentan en el capítulo 9.

**Typecasting y conversión entre tipos**

Como hemos visto, no puede asignar una variable a otra de un tipo diferente. En caso de que necesite hacerlo, hay dos opciones. La primera es el *typecasting* (moldeado de tipos), que usa una notación funcional simple, con el nombre del tipo de dato de destino :

**var**

  N: Integer;

  C: Char;

  B: Boolean;

**begin**

  N := Integer (*'X'*);

  C := Char (N);

  B := Boolean (0);

Puede usted moldear de un tipo de datos a otro del mismo tamaño. Normalmente es seguro moldear entre tipos ordinales, pero también puede moldear entre tipos de puntero (y también objetos), siempre que sepa qué está haciendo.

De cualquier modo, moldear suele ser una práctica de programación peligrosa, ya que le permite acceder a un valor como si representase otra cosa. Como las representaciones internas de tipos de datos no suelen encajar, se expone a errores difíciles de detectar. Por esta razón, lo mejor es evitar el moldeado de tipos.

La segunda opción es usar una rutina de conversión entre tipos. Las rutinas para los distintos tipos de conversiones vienen resumidas en la tabla 3.4. Algunas de estas rutinas funcionan sobre los tipos de datos que discutiremos en las siguientes secciones. Tenga en cuenta que la tabla no incluye rutinas para tipos especiales (como *TDateTime* o una variante) o rutinas destinadas especificamente al formateo, como las potentes rutinas *Format* y *FormatFloat*.

***Tabla 3.4: Rutinas del sistema para la conversión entre tipos***

|  |  |
| --- | --- |
| **Rutina** | **Descripción** |
| Chr | Convierte un número ordinal en un carácter ANSI. |
| Ord | Convierte un valor de tipo ordinal en el número que indica su orden. |
| Round | Convierte un valor de tipo real en un número de tipo entero, redondeando su valor. |
| Trunc | Convierte un valor de tipo real en un número de tipo entero, truncando su valor. |
| Int | Extrae la parte entera del argumento de valor en coma flotante. |
| IntToStr | Convierte un número en una cadena. |
| IntToHex | Convierte un número en una cadena con su representación hexadecimal. |
| StrToInt | Convierte una cadena en un número, provocando una exception si la cadena no representa un entero válido. |
| StrToIntDef | Convierte una cadena en un número, usando un valor por defecto si la cadena no es correcta. |
| Val | Convierte una cadena en un número (rutina tomada de Turbo Pascal, disponible por compatibilidad). |
| Str | Convierte un número en una cadena, usando parámetros de formateo (rutina tomada de Turbo Pascal, disponible por compatibilidad). |
| StrPas | Convierte una cadena de terminación nula (*null-terminated*)en una cadena de tipo Pascal. Esta conversión se realiza automáticamente sobre cadenas AnsiStrings en Delphi de 32 bits. (Véase la sección dedicada a las cadenas, más adelante en este mismo capítulo.) |
| StrPCopy | Copia una cadena tipo Pascal en una cadena de terminación nula. Esta conversión se hace con un simple simple moldeo (*typecast*) sobre0 PChar, en Delphi de 32 bits. (Véase la sección dedicada a las cadenas, más adelante en este mismo capítulo.) |
| StrPLCopy | Copia una porción de una cadena tipo Pascal en una cadena de terminación nula. |
| FloatToDecimal | Convierte un valor de coma flotante a un registro, incluyendo su representación decimal (exponentes, dígitos, signo). |
| FloatToStr | Convierte un valor de coma flotante a su representación como cadena, con formato por defecto. |
| FloatToStrF | Convierte un valor de coma flotante a su representación como cadena, con el formato especificado. |
| FloatToText | Copia un valor de coma flotante a un *buffer* (tampón) de cadena, con el formato especificado. |
| FloatToTextFmt | Como la rutina anterior, copia un un valor de coma flotante a un *buffer* (tampón) de cadena, con el formato especificado. |
| StrToFloat | Convierte una cadena Pascal a un valor de coma flotante. |
| TextToFloat | Convierte una cadena de terminación nula a un valor de coma flotante. |

**Conclusión**

En este capítulo hemos explorado la noción básica de tipo en Pascal. Pero el lenguaje tiene otra cualidad muy importante : permite a los programadores definir nuevos tipos de datos personalizados, llamados tipos de datos definidos por el usuario. Este es el asunto de que trata el siguiente capítulo.

**Capítulo siguiente:** [**Tipos de datos definidos por el usuario**](http://www.marcocantu.com/epascal/Spanish/ch04udt.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 4 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch04udt.htm) Tipos de datos definidos  por el usuario** |

Junto con la noción de tipo, una de las grandes ideas introducidas por el lenguaje Pascal es la habilidad de definir nuevos tipos de datos en un programa. Los programadores pueden definir sus propios tipos de datos mediante *constructores de tipos*, como tipos de subrango, tipos de matrices, tipos de *record* [registro], tipos enumerados, tipos de puntero [*pointer*]. El tipo de datos definido por el usuario, más importante, es la clase, que es parte de las extensiones orientadas a objeto de Object Pascal, no cubiertas en este libro.

Si cree usted que los constructores de tipos son comunes en varios lenguajes de programación, tiene usted razón; pero Pascal fue el primer lenguaje que introdujo la idea de una manera formal y precisa. Aún hay pocos lenguajes que tengan tantos mecanismos para definir tipos nuevos.

**Tipos con nombre y tipos sin nombre**

A estos tipos se les puede dar un nombre para usarlos más tarde o se pueden aplicar directamente a una variable. Cuando usted le da un nombre a un tipo, debe dedicarle una sección del código específica, como la siguiente :

**type**

*// definición del subrango*

Uppercase = 'A'..'Z';

*// definición de la matriz*

Temperatures = **array** [1..24] **of** Integer;

*// definición del 'record'*

Date = **record**

Month: Byte;

Day: Byte;

Year: Integer;

**end**;

*// definición de tipo enumerado*

Colors = (Red, Yellow, Green, Cyan, Blue, Violet);

*// establecer la definición*

Letters = **set of** Char;

Construcciones de definición de tipo, similares, pueden ser utilizadas directamente para definir una variable sin un nombre de tipo específico, como en el siguiente código :

**var**

DecemberTemperature: **array** [1..31] **of** Byte;

ColorCode: **array** [Red..Violet] **of** Word;

Palette: **set of** Colors;

Nota: En general, debería usted evitar usar tipos *sin nombre*, como en el código que aparece arriba, porque no puede usarlos como parámetros para rutinas o declarar otras variables del mismo tipo. La compatibilidad de tipos de Pascal, de hecho, está basada en nombres de tipo, no en la definición efectiva de los tipos. Dos variables de dos tipos idénticos aún no son compatibles, a no ser que sus tipos lleven exactamente el mismo nombre, y a los tipos sin nombre el compilador les da nombres internos. Acostúmbrese a definir un tipo de datos cada vez que necesite una variable compleja, y no se arrepentirá del tiempo que invirtió en hacerlo.

Pero ¿qué significan estas definiciones de tipo? Daré algunas descripciones para aquellos que no estén familiarizados con las construcciones de tipo de Pascal. También intentaré destacar las diferencias con las construcciones en otros lenguajes de programación, así que quizá le interese leer las secciones siguientes, incluso si conoce bien las definiciones de tipos como las de arriba. Finalmente, le mostraré algunos ejemplos en Delphi, e introduciré algunas herramientas que le permitirán acceder dinámicamente a la información de tipos.

**Tipos de subrango**

Un tipo de subrango define un rango de valores dentro del rango de otro tipo (de ahí el nombre *subrango*). Puede usted definir un subrango del tipo Integer, de 1 a 10 o de 100 a 1000, o puede usted definir un subrango del tipo Char[acter], como en :

**type**

Ten = 1..10;

OverHundred = 100..1000;

Uppercase = *'A'*..*'Z'*;

En la definición de un subrango, no necesita usted especificar el nombre del tipo base. Sólo necesita proporcionar dos constantes de ese tipo. El tipo original debe ser ordinal, y el tipo resultante será otro tipo ordinal.

Cuando haya definido un subrango, puede asignarle un valor dentro de ese rango. El siguiente código es válido :

**var**

UppLetter: UpperCase;

**begin**

UppLetter := *'F'*;

But this one is not:

**var**

UppLetter: UpperCase;

**begin**

UppLetter := *'e'*; *// error durante la compilación*

Un código como el de arriba resulta en un error durante la compilación : *"Constant expression violates subrange bounds"* [La expresión constante viola los límites del subrango]. Si, en vez de aquel, escribe el siguiente código ...

**var**

UppLetter: Uppercase;

Letter: Char;

**begin**

Letter :=*'e'*;

UppLetter := Letter;

... Delphi lo compilará. Durante la ejecución, si había usted habilitado la opción del compilador Range Checking [comprobación de rango] (en la ficha Compiler del cuadro de diálogo Project Options), obtendrá un mensaje de error *Range check error*.

**Nota:** Le sugiero activar esta opción del compilador mientras esté desarrollando un programa, con lo que sería más robusto y sencillo de depurar, ya que en caso de errores obtendrá un mensaje explícito, y no un comportamiento difícil de comprender. Al final, puede usted deshabilitar aquella opción para la versión definitiva del programa, para hacerlo un poco más rápido. De cualquier manera, la diferencia es realmente pequeña, y por esta razón le sugiero que deje activadas todas estas comprobaciones de tiempo de ejecución, incluso en un programa que vaya a distribuir. Lo mismo es válido para otras opciones de tiempo de ejecución, como las comprobaciones de desbordamiento [*overflow*].

**Tipos enumerados**

Los tipos enumerados constituyen otro tipo ordinal definido por el usuario. En vez de indicar un rango para un tipo existente, en una enumeración usted hace una lista de todos los valores posibles del tipo. En otras palabras, una enumeración es una lista de valores. Aquí hay algunos ejemplos :

**type**

Colors = (Red, Yellow, Green, Cyan, Blue, Violet);

Suit = (Club, Diamond, Heart, Spade);

Cada valor en la lista tiene una *ordinalidad* [número de orden] asociada, comenzando desde cero. Cuando aplica usted la función *Ord* a un valor de tipo enumerado, obtiene este valor. Por ejemplo, *Ord (Diamond)* devuelve 1.

Nota: Los tipos enumerados pueden tener distintas representaciones internas. Por defecto, Delphi usa una representación interna de 8 bits, a no ser que haya más de 256 valores diferentes, en cuyo caso usa la representación de 16 bits. También hay una representación de 32 bits, que podría ser útil para mantener la compatibilidad con bibliotecas de C o C++. De hecho, puede usted cambiar el comportamiento por defecto, pidiendo una representación de mayor tamaño, usando la directiva del compilador *$Z*.

La VCL (*Visual Component Library*, biblioteca de componentes visuales) de Delphi, usa tipos enumerados en muchas ocasiones. Por ejemplo, es estilo del borde de un formulario se define como sigue :

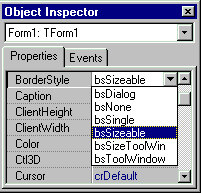
**type**

TFormBorderStyle = (bsNone, bsSingle, bsSizeable,

bsDialog, bsSizeToolWin, bsToolWindow);

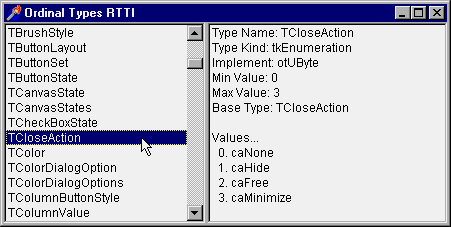
Cuando el valor de una propiedad es una enumeración, normalmente podrá elegir de la lista de valores que se muestra en el Object Inspector, como se muestra en la figura 4.1.

***Figura 4.1: Una propiedad de tipo enumerado en el Object Inspector***



En la ayuda de Delphi se mencionan los posibles valores de una enumeración. Como alternativa, puede usted utilizar el programa *OrdType*, disponible en www.marcocantu.com, para ver la lista de valores en Delphi de cada enumeración, conjunto, subrango y cualquier otro tipo ordinal. Puede ver un ejemplo de la salida de este programa en la figura 4.2.

***Figura 4.2: Información detallada sobre un tipo enumerado, como la muestra el programa OrdType (disponible en mi sitio en la Red).***



**Tipos de conjunto [*set*]**

Tipos de conjunto indican un grupo de valores, donde la lista de valores disponibles se indica mediante el tipo ordinal en que se basa el conjunto. Los tipos suelen ser limitados, y a menudo se representan con una enumeración o un subrango. Si tomamos el subrango 1..3, los valores posible del conjunto basado en él incluyen sólo 1, sólo 2, sólo 3, tanto 1 como 2, tanto 1 como 3, 2 y 3, todos los tres valores, o ninguno de ellos.

Una variable normalmente contiene exactamente uno de los valores posibles para el rango de su tipo. Una variable de tipo conjunto, sin embargo, puede contener uno, dos o más valores del rango. Incluso puede incluirlos todos. He aquí un ejemplo de un Set :

**type**

Letters = **set of** Uppercase;

Ahora podemos definir una variable de este tipo y asignarle algunos valores del tipo original. Para indicar algunos valores en un conjunto, se escribe una lista separada por comas, encerrada en corchetes. El siguiente código muestra la asignación de varios valores a una variable, de uno sólo, y del valor 'vacío' :

**var**

Letters1, Letters2, Letters3: Letters;

**begin**

Letters1 := ['A', 'B', 'C'];

Letters2 := ['K'];

Letters3 := [];

En Delphi, un conjunto se suele utilizar para indicar *flags* no exclusivos. Por ejemplo, las dos siguientes líneas de código (que son parte de la biblioteca Delphi) declaran una enumeración de posibles iconos para el borde de una ventana y el correspondiente tipo de conjunto.

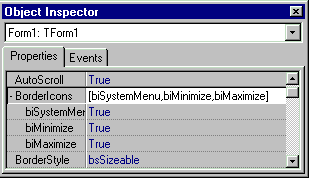
**type**

TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);

TBorderIcons = **set of** TBorderIcon;

De hecho, una ventana determinada podría no tener ninguno de estos iconos, uno de ellos, o más. Cuando trabaje con el Object Inspector (observe la figura 4.3), podrá proporcionarle los valores a un conjunto expandiendo la selección (haga doble clic en el nombre de la propiedad o clic en el signo '+' a su izquierda) activando y desactivando la presencia de cada valor.

***Figura 4.3: Una propiedad de tipo de conjunto en el Object Inspector***



Otra propiedad basada en un tipo de conjunto es el estilo de una fuente. Los valores posibles indican fuentes en negrita, cursiva, subrayadas o tachadas. Por supuesto, una misma fuente puede ser a la vez cursiva y negrita, no tener propiedades, o tenerlas todas. Por esta razón se declara como un conjunto. Puede usted asignar valores a este conjunto en el código de un programa, como sigue :

Font.Style := []; *// no style*

Font.Style := [fsBold]; *// bold style only*

Font.Style := [fsBold, fsItalic]; *// two styles*

También puede operar sobre un conjunto de muchas maneras distintas, incluyendo el añadir dos variables del mismo tipo de conjunto (o, para ser más preciso, calcular la unión de las dos variables de conjunto) :

Font.Style := OldStyle + [fsUnderline]; *// two sets*

Una vez más, puede utilizar los ejemplos OrdType incluidos en el directorio TOOLS del código fuente del libro para ver la lista de valores posibles de muchos conjuntos definidos por la biblioteca de componentes de Delphi.

**Tipos de vector [*array*]**

Los tipos de vector definen listas de un número fijo de elementos de un tipos específico. Normalmente se utiliza un *índice* entre corchetes para acceder a uno de los elementos del vector. Los corchetes también se utilizan para especificar los valores posibles del índice cuando el vector ha sido definido. Por ejemplo, puede definir un grupo de 24 enteros con este código :

**type**

DayTemperatures = **array** [1..24] **of** Integer;

En la definición del vector, tiene usted que proporcionar un tipo de subrango entre corchetes, o definir un sugrango específico nuevo, utilizando dos constantes de un tipo ordinal. Este subrango especifica los índices válidos del vector. Como se especifica tanto el índice superior como el inferior del vector, los índices no tienen por qué comenzar por cero, como sí es necesario en C, C++, Java y otros lenguajes de programación.

Como los índices del vector están basado en subrangos, Delphi puede comprobar su rango, como ya hemos visto. Un subrango de constante no válido resulta en un error de compilación; y un índice fuera de rango utilizado durante la ejecución resulta en un error de ejecución si la correspondiente opción de compilador está activada.

Usando la definición de vector de arriba, puede usted establecer el valor de una variable *DayTemp1* del tipo *DayTemperatures*, como sigue :

**type**

DayTemperatures = **array** [1..24] **of** Integer;

**var**

DayTemp1: DayTemperatures;

**procedure** AssignTemp;

**begin**

DayTemp1 [1] := 54;

DayTemp1 [2] := 52;

...

DayTemp1 [24] := 66;

DayTemp1 [25] := 67; *// compile-time error*

Una matriz puede tener más de una dimensión, como en los siguientes ejemplos :

**type**

MonthTemps = **array** [1..24, 1..31] **of** Integer;

YearTemps = **array** [1..24, 1..31, Jan..Dec] **of** Integer;

Estos dos tipos de vector se construyen esencialmente sobre los mismos tipos. Así que puede declararlos utilizando los tipos de datos precedentes, como en el siguiente código :

**type**

MonthTemps = **array** [1..31] **of** DayTemperatures;

YearTemps = **array** [Jan..Dec] **of** MonthTemps;

Esta instrucción invierte el orden de los índices como se muestra arriba, pero también permite adjudicar bloques enteros entre las variables. Por ejemplo, la siguiente instrucción copia las temperaturas de enero a febrero :

**var**

ThisYear: YearTemps;

**begin**

...

ThisYear[Feb] := ThisYear[Jan];

También puede definir un vector que *comience en el cero*, donde el límite ordinal inferior es cero. Generalmente, el uso de límites más lógicos es una ventaja, ya que no tiene que usar el índice 2 para acceder al tercer elemento, y así sucesivamente. Sin embargo, Windows utiliza invariablemente vectores que comienzan en el cero (porque está basado en el lenguaje C), y la biblioteca de componentes de Delphi tiene a hacer lo mismo.

Si necesita trabajar en un vector, puede siempre comprobar cuáles son sus límites, utilizando las funciones normalizadas *Low* y *High*, que devuelven los límites inferior y superior. Le recomiendo encarecidamente utilizar *Low* y *High* al operar sobre un vector, especialmente en bucles, ya que hace al código independiente del rango de la matriz. Más tarde, podrá cambiar el rango declarado de los índices del vector, y el código que use *Low* y *High* seguirá funcionando. Si escribe usted un bucle fijando el rango de un vector, tendrá que actualizar el código del bucle cuando cambie el tamaño del vector. *Low* y *High* hacen su código más fácil de mantener y más fiable.

Nota: Por cierto, no se produce un aumento en el gasto de recursos durante la ejecución por usar *Low* y *High* con vectores. Son convertidos durante la compilación en expresiones constantes, no en llamadas a funciones. Esta conversión de expresiones y llamadas a funciones durante la compilación también ocurre con muchas otras funciones simples de sistema.

Delphi usa vectores principalmente en el formulario de propiedades de vectores. Ya hemos visto un ejemplo de tal propiedad en el ejemplo TimeNow, para acceder la propiedad *Items* de un componente ListBox. Le mostraré algunos ejemplos más de propiedades de vectores en el [**siguiente capítulo**](http://www.marcocantu.com/epascal/Spanish/ch05stat.htm), cuando discuta los bucles Delphi.

Nota: Delphi 4 introdujo los vectores dinámicos en Object Pascal, esto es, vectores que se pueden cambiar de tamaño durante la ejecución, adjudicándoles la cantidad adecuada de memoria. Es fácil usar vectores dinámicos, pero para esta discusión de Pascal consideré que no era adecuado tratarlos. Puede encontrar una descripción de los vectores dinámicos de Delphi en el [**Capítulo 8**](http://www.marcocantu.com/epascal/Spanish/ch08mem.htm).

**Tipos de registro (*record*)**

Los tipos *record* definen colecciones fijas de elementos de distintos tipos. Cada elemento, o *campo*, tiene su propio tipo. La definición de un tipo *record* incluye todos estos campos, dándole un nombre a cada uno, que se utiliza para acceder a él posteriormente.

Aquí aparece un pequeño listado con la definición de un tipo *record*, la instrucción de una variable de tal tipo, y algunas instrucciones en que se utiliza esta variable :

**type**

Date = **record**

Year: Integer;

Month: Byte;

Day: Byte;

**end**;

**var**

BirthDay: Date;

**begin**

BirthDay.Year := 1997;

BirthDay.Month := 2;

BirthDay.Day := 14;

Las clases y los objetos se pueden considerar una extensión del tipo *record*. Las bibliotecas de Delphi tienden a usar tipos de clase en vez de de *record*, pero hay muchos tipos *record* definidos por el API de Windows.

Los tipos *record* también pueden tener una parte variante, esto es, campos múltiples pueden ser relacionados con la misma área de memoria, incluso si son de un tipo de datos distinto. (Esto equivale a una unión en el lenguaje C.) También puede utilizar estos campos variantes o grupos de campos para acceder al mismo lugar de la memoria dentro de un *record*, pero considerando esos valores desde perspectivas distintas. Los principales usos de este tipo eran almacenar datos similares, pero distintos, y obtener un efecto similar al de *typecasting* (algo menos útil ahora que éste también ha sido introducido en Pascal). El uso de tipos de *record* variantes ha sido reemplazado en gran parte por técnicas orientadas a objetos, y otras técnicas modernas, aunque Delphi los usa en algunos casos peculiares.

El uso de un tipo *record* variante no es a prueba de tipos , y no representa una práctica de programación recomendable, especialmente para principiantes. Los programadores expertos pueden, de hecho, utilizar tipos *record* variantes, y las bibliotecas básicas de Delphi los usan. En cualquier caso, no necesitará ocuparse en ellas hasta que sea realmente un programador experto.

**Punteros [*pointers*]**

Un tipo puntero define una variable que contiene la dirección de memoria de otra variable de un tipo de datos dado (o indefinido). Así, una variable de puntero apunta indirectamente a un valor. La definición de un tipo puntero no está basada en una palabra clave (*keyword*) específica, sino que usa un carácter especial, en vez de ello. Dicho símbolo es el acento circunflejo (^) :

**type**

PointerToInt = ^Integer;

Una vez que haya definido una variable puntero, puede asignarla a la dirección de otra variable del mismo tipo, usando el operador @ :

**var**

P: ^Integer;

X: Integer;

**begin**

P := @X;

*// change the value in two different ways*

X := 10;

P^ := 20;

Cuando tenga un puntero *P*, con la expresión *P* se referirá a la dirección de la posición de memoria a que apunta el puntero, y con la expresión *P^* al contenido real de aquella dirección. Por esta razón, en el fragmento de código de arriba, *^P* corresponde a *X*.

En vez de referirse a una posición de memoria existente, un puntero se puede referir a un nuevo bloque de memoria asignado dinámicamente (en el área de memoria *heap*) con el procedimiento *New*. En este caso, cuando deje de necesitar el puntero, deberá deshacerse de la memoria que adjudicó dinámicamente, haciendo una llamada al procedimiento *Dispose*.

**var**

P: ^Integer;

**begin**

*// initialization*

New (P);

*// operations*

P^ := 20;

ShowMessage (IntToStr (P^));

*// termination*

Dispose (P);

**end**;

Si un puntero no tiene valor, puede asignarle el valor *nil* (vacío). Entonces, puede comprobar si un puntero es *nil* para ver si actualmente apunta a algún valor. Esto se hace a menudo, porque desreferenciar un puntero inválido causa una infracción de acceso (también llamada fallo de protección general, GPF) :

procedure TFormGPF.BtnGpfClick(Sender: TObject);

var

P: ^Integer;

begin

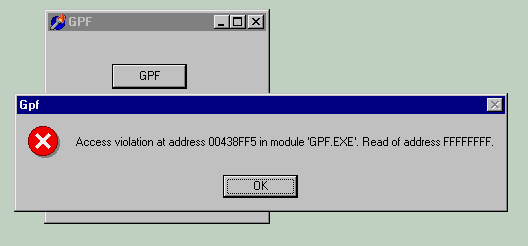
P := nil;

ShowMessage (IntToStr (P^));

end;

Puede usted ver un ejemplo del efecto de esta porción de código ejecutando el ejemplo GPF (u observando la figura correspondiente, 4.4). El ejemplo contiene también los fragmentos de código que aparecen arriba.

***Figura 4.4: El error de sistema que resulta de acceder a un puntero nil , del ejemplo GPF.***



En el mismo programa encontrará un ejemplo de acceso seguro a datos. En este segundo caso, el puntero es asignado a una variable local existente, y puede ser utilizado con seguridad, pero a pesar de ello he añadido una comprobación de seguridad :

**procedure** TFormGPF.BtnSafeClick(Sender: TObject);

**var**

P: ^Integer;

X: Integer;

**begin**

P := @X;

X := 100;

**if** P **nil then**

ShowMessage (IntToStr (P^));

**end**;

Delphi también define un tipo de datos llamado *Pointer*, que indica punteros sin tipo (como el *void\** en el lenguaje C). Si necesita un puntero sin tipo, debería usar *GetMem* en vez de *New*. Se necesita el procedimiento *GetMem* cuando el tamaño de la variable de memoria a que se desea apuntar no esté definido.

El hecho de que los punteros son raramente necesarios en Delphi es una ventaja interesante de este entorno. Sin embargo, entender a los punteros es importante para programación avanzada y para una comprensión completa del modelo de objetos de Delphi, que usa punteros "tras el telón".

Nota: Aunque no se use punteros en Delphi muy a menudo, sí que se usa un tipo de construcción muy similar : las referencias. Cada ejemplo de objeto es, en realidad, un puntero implícito a los datos que contiene. De cualquier forma, esto es absolutamente transparente al programados, que usa variables de objeto como cualquier otro tipo de datos.

**Tipos de archivo [*file*]**

Otro constructor de tipos específico de Pascal es el tipo *file*. Los tipos de archivo representan archivos de disco físicos, lo cual es, ciertamente, una peculiaridad del lenguaje Pascal. Puede definir un nuevo tipo de archivo como sigue :

**type**

IntFile = **file of** Integer;

Entonces puede abrir un archivo físico asociado a esta estructura y escribir valores enteros en él, o leer los valores del mismo.

**Nota del autor** : Ejemplos basados en archivos fueron parte de versiones anteriores a *Mastering Delphi 5* [en español, *Delphi 4*]. Tengo previsto incluirlos también aquí.

El uso de los archivos en Pascal es bastante intuitivo, pero en Delphi hay también algunos componentes que son capaces de almacenar o cargar su contenido de o a un archivo. Hay alguna ayuda para hacer esto en serie, en forma de flujos (*streams*), y también se apoya el uso de bases de datos.

**Conclusión**

Este capítulo, que discute tipos de datos definidos por el usuario completan la información que damos sobre el sistema de tipos de Pascal. Ahora estamos preparados para investigar las instrucciones que proporciona este lenguaje para operar sobre las variables que hemos definido.

**Capítulo siguiente:** [**Instrucciones**](http://www.marcocantu.com/epascal/Spanish/ch05stat.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 5 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch05stat.htm) Instrucciones** |

Si los tipos de datos forman una de las bases de la programación en Pascal, la otra la forman las instrucciones. Las instrucciones de este lenguaje de programación están basadas en palabras clave (*keywords*) y otros elementos que permiten indicarle a un programa una sucesión de operaciones a realizar. Las instrucciones a menudo se encierran en procedimientos o funciones, como veremos en el [**capítulo siguiente**](http://www.marcocantu.com/epascal/Spanish/ch06proc.htm). Ahora, nos ceñiremos al estudio de tipos básicos de comandos que pueden usarse para crear un programa.

**Instrucciones simples y compuestas**

Una instrucción en Pascal es simple si no contiene otras instrucciones. Ejemplos de instrucciones sencillas son las de asignación y las de llamada a procedimientos. Las instrucciones simples se separan con punto-y-comas :

X := Y + Z; *// asignación*

Randomize; *// llamada a un procedimiento*

Normalmente, las instrucciones se agrupan en instrucciones compuestas, delimitadas por los comandos *begin* y *end*. Una instrucción compuesta puede aparecer en lugar de una instrucción en Pascal, genérica. Sigue un ejemplo :

**begin**

A := B;

C := A \* 2;

**end**;

El punto y coma tras la última instrucción previa al *end* no es imprescindible, como se ve aquí :

**begin**

A := B;

C := A \* 2

**end**;

Ambas versiones son correctas. La primera incluye un punto y coma innecesario (pero inofensivo). El punto y coma es, de hecho, una instrucción nula, una instrucción sin código. Dése cuenta de que, a veces, las instrucciones nulas pueden ser utilizadas dentro de bucles, o en otros casos particulares.

Nota: Aunque estos punto-y-comas finales no tienen finalidad, tiendo a usarlos, y le sugiero que haga lo mismo. A veces, tras escribir un cierto número de líneas, podría necesitar añadir otra. Si el último punto y coma fue omitido en primera instancia, tendrá que añadirlo después, y esto resulta algo más engorroso que ponerlo desde un principio.

**Instrucciones de asignación**

Las asignaciones en Pascal usan el operador formado por dos puntos seguidos por el signo 'igual', una notación extraña para programadores acostumbrados a otros lenguajes. El operador =, utilizado para asignar valores en otros lenguajes de programación, se usa en Pascal para comprobar la igualdad de dos valores.

Nota: Al usar distintos símbolos para asignación y para prueba de igualdad, el compilador de Pascal (como el de C) puede traducir código fuente más rápidamente, porque no necesita examinar el contexto en que el operador se utiliza, para determinar su significado. El uso de dos operadores diferentes también hace que el código sea más fácil de leer al ser humano.

**Instrucciones condicionales**

Una instrucción condicional se usa para ejecutar bien una de las instrucciones que incluye o ninguna, dependiendo de cierta comprobación. Hay dos tipos básicos de instrucciones condicionales: instrucciones *if* e instrucciones *case*.

**Instrucciones *If***

La instrucción *if* puede usarse para ejecutar una instrucción sólo si se cumple una cierta condición (*if-then*; si...entonces), o para elegir entre dos alternativas (*if-then-else*; si...entonces..., y, si no, ...) La condición viene expresada mediante una expresión Booleana. Un ejemplo sencillo en Delphi puede mostrar cómo escribir instrucciones condicionales. Primero, cree una aplicación nueva, y coloque dos casillas de selección y cuatro botones en el formulario. No cambie los nombres de los botones o las casillas de selección, sino haga doble clic en cada botón para añadir un gestor de eventos para el evento *OnClick* correspondiente. He aquí una instrucción *if* sencilla para el primer botón :

**procedure** TForm1.Button1Click(Sender: TObject);

**begin**

*// simple if statement*

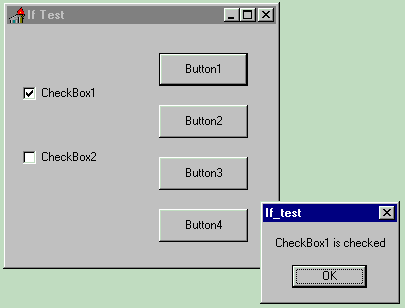
**if** CheckBox1.Checked **then**

ShowMessage (*'CheckBox1 is checked'*)

**end**;

Cuando pulse botón, si la primera casilla de selección lleva un signo de activación, el programa mostrará un breve mensaje (vea la Figura 5.1). He usado la función ShowMessage porque es la función Delphi más sencilla que puede usar para mostrar un mensaje breve al usuario.

***Figura 5.1: El mensaje mostrado por el ejemplo IfTest cuando se pulsa el primer botón y la primera casilla de selección está activada.***



Si pulsa el botón y no pasa nada, significa que la casilla no había sido activada. En un caso así, probablemente sería mejor hacer esto más explícito, como con el código para el segundo botón, que usa una instrucción if-then-else :

b>procedure TForm1.Button2Click(Sender: TObject);

**begin**

*// instrucción if-then-else*

**if** CheckBox2.Checked **then**

ShowMessage (*'La casilla CheckBox2 está activada'*)

**else**

ShowMessage (*'CheckBox2 NO está activada'*);

**end**;

Dése cuenta de que no puede colocar un punto y coma entre la primera instrucción y la palabra clave *else*, o el compilador mostrará un error de sintaxis. De hecho, la instrucción *if-then-else*, de hecho, es una sola instrucción, así que no puede colocar un punto y coma en medio de ella.

Una instrucción *if* puede ser bastante compleja. La condición puede ser convertida en una serie de condiciones (usando los operadores Booleanos *and*, *or* y *not*), o bien la instrucción *if* puede incluir otra instrucción *if*. Los dos últimos botones del ejemplo IfTest muestran estos casos :

**procedure** TForm1.Button3Click(Sender: TObject);

**begin**

*// instrucción de doble condición*

**if** CheckBox1.Checked **and** CheckBox2.Checked **then**

ShowMessage (*'Ambas casillas están activadas'*)

**end**;

**procedure** TForm1.Button4Click(Sender: TObject);

**begin**

*// instrucción if compuesta*

**if** CheckBox1.Checked **then**

**if** CheckBox2.Checked **then**

ShowMessage (*'Las casillas 1 y 2 están activadas'*)

**else**

ShowMessage (*'Sólo la casilla 1 está activada'*)

**else**

ShowMessage (

*'La casilla 1 no está activada. ¿A quién le importa la 2?'*)

**end**;

Observe detenidamente el código y ejecute el programa, para ver si lo entiende todo. Cuando tenga dudas sobre la conformación de un programa, escribir un programa muy simple como este, puede ayudarle a aprender mucho. Puede añadir más casillas de selección y aumentar la complejidad de este breve ejemplo, haciendo todas las pruebas que necesite.

**Instrucciones *Case***

Si sus instrucciones *if* se hacen muy complejas, a veces puede reemplazarlas con instrucciones *case*. Una instrucción *case* consiste en una expresión usada para seleccionar un valor, una lista de posibles valores, o una gama de valores. Estos valores son constantes, y deben ser únicos y de tipo ordinal. Finalmente, puede haber una instrucción *else* que sea ejecutada si ninguna de las etiquetas se corresponde con el valor del selector. Siguen dos ejemplos sencillos :

**case** Number **of**

1: Text := *'Uno'*;

2: Text := *'Dos'*;

3: Text := *'Tres'*;

**end**;

**case** MyChar **of**

'+' : Text := 'Signo más';

'-' : Text := 'Signo menos';

'\*', '/': Text := 'Multiplicación o división';

'0'..'9': Text := 'Número';

'a'..'z': Text := 'Minúscula';

'A'..'Z': Text := 'Mayúscula';

**else**

Text := 'Carácter desconocido';

**end**;

**Bucles en Pascal**

El lenguaje Pascal tiene las instrucciones recursivas típicas de la mayoría de lenguajes de programación, incluyendo las instrucciones *for*, *while* y *repeat*. La mayoría de lo que hacen estos bucles le resultará familiar si conoce otros lenguajes de programación, por lo que los comentaré muy brevemente.

**El bucle *For***

Este tipo de bucle, en Pascal, está basado estrictamente en un contador, que puede ser incrementado o decrementado, cada vez que se ejecuta el bucle.  
He aquí un ejemplo sencillo de un bucle *for* que se usa para añadir los primeros diez números.

**var**

K, I: Integer;

**begin**

K := 0;

**for** I := 1 **to** 10 **do**

K := K + I;

Esta misma instrucción *for* podría haber sido escrita usando un contador inverso :

**var**

K, I: Integer;

**begin**

K := 0;

**for** I := 10 **downto** 1 **do**

K := K + I;

El bucle *for* es menos flexible en Pascal que en otros lenguajes (no es posible especificar un incremento distinto de uno), pero es sencillo y fácil de entender. Si quiere incorporar una comprobación más compleja, o proporcionar un contador personalizado, necesitará usar una instrucción *while* o *repeat*, en vez de un bucle *for*.

Nota: El contador de un bucle for no tiene por qué ser un número. Puede ser un valor de cualquier tipo ordinario, como un carácter o un tipo enumerado.

**Instrucciones *While* y *Repeat***

La diferencia entre el bucle *while-do* (mientras ... haz) y el bucle *repeat-until* (repite ... hasta que) consiste en que el código de la instrucción *repeat* se ejecuta siempre al menos una vez. Puede comprender fácilmente por qué, observando un ejemplo sencillo :

**while** (I <= 100) **and** (J <= 100) **do**

**begin**

*// usamos I y J para calcular algo ...*

I := I + 1;

J := J + 1;

**end**;

**repeat**

*// usamos I y J para calcular algo ...*

I := I + 1;

J := J + 1;

**until** (I > 100) **or** (J > 100);

Si el valor inicial de *I* o *J* es mayor que 100, las instrucciones en el interior del bucle repeat-until son ejecutadas una vez, a pesar de ello.

La otra diferencia clave entre estos dos bucles es que el bucle *repeat-until* incluye una condición *inversa*. El bucle se ejecuta mientras la condición *no* se cumpla. Cuando sí lo hace, el bucle finaliza. Esto es lo contrario de lo que pasa con un bucle *while-do*, que se ejecuta mientras la condición sea cierta. Por esta razón tuve que invertir la condición en el código que aparece arriba, para obtener una instrucción equivalente.

**Un ejemplo de bucles**

Para explorar los detalles de los bucles, echemos un vistazo a un pequeño ejemplo en Delphi. El programa Loops destaca la diferencia entre un bucle con contador fijo y otro con un contador casi aleatorio. Comience con un proyecto nuevo, coloque una casilla de lista (*listbox*) y dos botones en el formulario principal, y déle a los botones un nombre propio (BtnFor y BtnWhile), estableciendo su propiedad Name en el Object Inspector. También puede quitar la palabra Btn de la propiedad Caption (e, incluso, finalmente, añadirle el carácter & para activar la siguiente letra como tecla de acceso directo). Aquí aparece lo más importante de la descripción en código de tal formulario :

**object** Form1: TForm1

Caption = 'Loops'

**object** ListBox1: TListBox ...

**object** BtnFor: TButton

Caption = '&For'

OnClick = BtnForClick

**end**

**object** BtnWhile: TButton

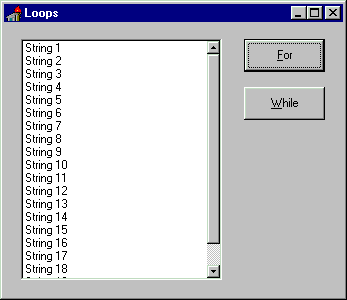
Caption = '&While'

OnClick = BtnWhileClick

**end**

**end**

***Figura 5.2: Cada vez que pulse el botón For del ejemplo Loops, la casilla de lista se llena con números correlativos.***



Ahora podemos añadir algo de código a los eventos *OnClick* de sendos botones. El primero de ellos incluye un sencillo bucle *for* para representar una lista de números, como se ve en la Figura 5.2. Antes de ejecutar este bucle, que añade un cierto número de cadenas a la propiedad Items de la casilla de lista, es necesario eliminar el contenido de la casilla misma :

**procedure** TForm1.BtnForClick(Sender: TObject);

**var**

I: Integer;

**begin**

ListBox1.Items.Clear;

**for** I := 1 **to** 20 **do**

Listbox1.Items.Add (*'String '* + IntToStr (I));

**end**;

El código asociado con el segundo botón es un poco más complejo. En este caso, hay un bucle while basado en un contador, que se va incrementando de forma aleatoria. Para conseguir esto, he efectuado una llamada al procedimiento *Randomize*, que inicializa el generador de números aleatorios, y la función Random, con una gama de 100 valores. El resultado de esta función es un número entre 0 y 99, elegido al azar. La sucesión de números aleatorios controla cuántas veces se ejecuta el bucle *while*.

**procedure** TForm1.BtnWhileClick(Sender: TObject);

**var**

I: Integer;

**begin**

ListBox1.Items.Clear;

Randomize;

I := 0;

**while** I < 1000 **do**

**begin**

I := I + Random (100);

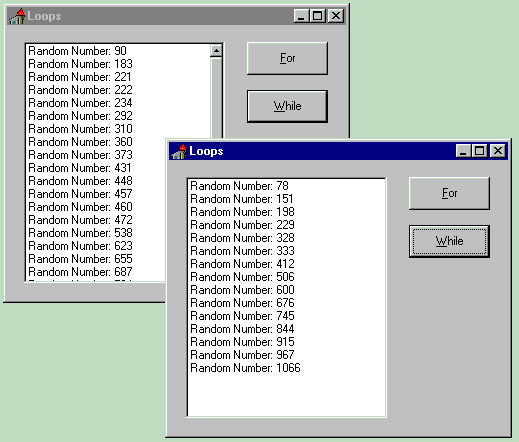
Listbox1.Items.Add (*'Random Number: '* + IntToStr (I));

**end**;

**end**;

Cada vez que pulse el botón While, los números son distintos, porque dependen del generador de números aleatorios. La Figura 5.3 muestra los resultados de dos clics de botón distintos. Observe que no sólo cambian los números generados de una vez a otra, sino también el número de elementos. Esto es, este bucle *while* se ejecuta un número aleatorio de veces. Si pulsa el botón While varias veces seguidas, verá que la casilla de lista tiene un número distinto de líneas.

***Figura 5.3: El contenido de la casilla de lista del ejemplo Loops cambia cada vez que se pulsa el botón While. Como el contador de bucle se incrementa con un valor aleatorio, cada vez que pulse el botón, el bucle se ejecuta un número distinto de veces.mes.***



Nota: Se puede alterar el flujo normal de la ejecución de un bucle, usando los procedimientos de sistema *Break* y *Continue*.   
El primero interrumpe el bucle; el segundo se usa para saltar directamente a la comprobación del bucle o al incremento del contador, siguiendo con la siguiente iteración del bucle (a no ser que la condición sea cero, o que el contador haya llegado a su máximo valor). Otros dos procedimientos de sistema, *Exit* y *Halt*, le permiten volver inmediatamente de la función o el procedimiento actual o finalizar el programa.

**La instrucción *With***

El último tipo de instrucción Pascal que comentaré es la instrucción *with*, que es específica de este lenguaje de programación (introducida recientemente en Visual Basic), y muy útil en la programación Delphi.

La instrucción *with* no es otra cosa que una abreviatura. Cuando necesite referirse a una variable de tipo registro (*record*) o a un objeto, en vez de repetir su nombre cada vez, puede usar una instrucción *with*. Por ejemplo, al presentar el tipo *record*, escribí este código :

**type**

Date = **record**

Year: Integer;

Month: Byte;

Day: Byte;

**end**;

**var**

BirthDay: Date;

**begin**

BirthDay.Year := 1997;

BirthDay.Month := 2;

BirthDay.Day := 14;

Mediante la instrucción *with* puedo mejorar la parte final de este código, como sigue :

**begin**

**with** BirthDay **do**

**begin**

Year := 1995;

Month := 2;

Day := 14;

**end**;

Este enfoque puede ser usado en progamas Delphi para referirse a componentes y otros tipos de clases. Por ejemplo, podemos reescribir la última parte del ejemplo anterior a este, usando una instrucción with para acceder a los elementos de la casilla de lista :

**procedure** TForm1.WhileButtonClick(Sender: TObject);

**var**

I: Integer;

**begin**

**with** ListBox1.Items **do**

**begin**

Clear; *// acceso directo*

Randomize;

I := 0;

**while** I < 1000 **do**

**begin**

I := I + Random (100);

*// shortcut:*

Add (*'Random Number: '* + IntToStr (I));

**end**;

**end**;

**end**;

Cuando trabaje con componentes o clases en general, la instrucción *with* le permitirá ahorrarse algo de código, en especial para campos anidados. Por ejemplo, suponga que necesita cambiar la anchura (Width) y el Color del lápiz de dibujo para un formulario. Puede aplicar el siguiente código :

Form1.Canvas.Pen.Width := 2;

Form1.Canvas.Pen.Color := clRed;

Pero, ciertamente, es más fácil escribir este otro :

**with** Form1.Canvas.Pen **do**

**begin**

Width := 2;

Color := clRed;

**end**;

Cuando se escribe código complejo, la instrucción *with* puede ser efectiva, y le ahorra la declaración de algunas variables temporales, pero tiene una desventaja. Puede hacer que el código sea más ilegible, en especial si está trabajando con distintos objectos que tienen propiedades similares o correspondientes entre sí.

Otra desventaja es que usar la instrucción *with* puede permitir la aparición de sutiles errores lógicos en el código, que el compilador no detectará. Por ejemplo :

**with** Button1 **do**

**begin**

Width := 200;

Caption := *'New Caption'*;

Color := clRed;

**end**;

Este código cambia el *Caption* y el *Width* del botón, pero afecta a la propiedad *Color* del formulario, ¡no del botón! La razón es que los componentes TButton no tienen propiedad Color, y, como el código se ejecuta para un objeto que forma parte de un formulario (estamos escribiendo un método para éste), se accede a este objeto por defecto. Si hubiéramos escrito, en su lugar ...

Button1.Width := 200;

Button1.Caption := *'New Caption'*;

Button1.Color := clRed; *// error!*

... el compilador habría mostrado un mensaje de error. En general, se puede decir que, como la instrucción *with* introduce nuevos identificadores en el ámbito actual, podríamos esconder identificadores presentes, o acceder por error a identificadores que se encuentran en el mismo ámbito (como en la primera versión de este fragmento de código). Incluso considerando este tipo de desventaja, le sugiero que se acostumbre a las instrucciones *with*, porque pueden ser realmente manejables, y a veces aun hacen que el código sea más legible.

De cualquier manera, debería evitar usar instrucciones múltiples, como :

**with** ListBox1, Button1 **do**...

El código que siguiese a esto sería, probablemente, muy ilegible, porque para cada propiedad definida en este bloque, tendría que plantearse a qué componente afecta, dependiendo de las propiedades respectivas y del orden de los componentes en la instrucción *with*.

Nota: Hablando de legibilidad, Pascal no tiene instrucción alguna de tipo *endif* ó *endcase*. Si una instrucción *if* incluye un bloque *begin-end*, el final de tal bloque marca el fin de esa instrucción. Por su parte, la instrucción *case* siempre acaba en *end*. Todas estas instrucciones *end*, que a menudo se encuentran una tras otra, pueden hacer que el código sea difícil de seguir. Sólo siguiendo las sangrías en el código se puede ver a qué instrucción afecta un *end* en particular. Una forma habitual de resolver este problema y hacer el código más legible es añadir un comentario tras la instrucción *end*, indicando su papel, como en :

**if** ... **then**

...

**end**; *// if*

**Conclusión**

En este capítulo se describe cómo se codifican instrucciones condicionales y bucles. En vez de escribir listas largas de tales instrucciones, los programas se desmembran normalmente en rutinas, procedimientos o funciones. Este es el asunto del siguiente capítulo, que introduce también algunos elementos avanzados de Pascal.

**Capítulo siguiente:** [**Procedimientos**](http://www.marcocantu.com/epascal/Spanish/ch06proc.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 6 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch06proc.htm) Procedimientos y funciones** |

Otra idea importante enfatizada por Pascal es el concepto de rutina, básicamente una serie de instrucciones con un único nombre, que pueden ser activadas muchas veces utilizando el mismo. De esta manera, se evita repetir las mismas instrucciones una y otra vez, y teniendo una única versión del código le permite modificarlo con efecto sobre todo el programa. Desde este punto de vista, puede pensar en las rutinas como el mecanismo básico de encapsulamiento de código. Volveré a tratar de este asunto con un ejemplo después de introducir la sintaxis de las rutinas en Pascal.

**Procedimientos y funciones en Pascal**

En Pascal, una rutina puede asumir dos formas: un procedimiento y una función. En teoría, un procedimiento es una operación que se pide a la computadora que realice, y una función es un cálculo que devuelve un valor. Esta diferencia se enfatiza por el hecho de que una función tiene un resultado, un valor de salida, mientras que un procedimiento no. Ambos tipos de rutinas pueden tener múltiples parámetros, de tipos de datos dados.

En la práctica, sin embargo, la diferencia entre funciones y procedimientos es muy limitada: puede hacer una llamada a una función para realizar cierta tarea y luego saltarse el resultado (que podría ser un código de error opcional o algo similar) o puede hacer una llamada a un procedimiento que transmite un resultado dentro de sus parámetros (más sobre parámetros de referencia se comentará más tarde en este capítulo).

Aquí están las definiciones de un procedimiento y dos versiones de la misma función, usando una sintaxis ligeramente distinta :

**procedure** Hello;

**begin**

ShowMessage ('Hello world!');

**end**;

**function** Double (Value: Integer) : Integer;

**begin**

Double := Value \* 2;

**end**;

*// ó, como alternativa*

**function** Double2 (Value: Integer) : Integer;

**begin**

Result := Value \* 2;

**end**;

El uso de Result en vez del nombre de la función para asignar el valor que devuelve una función se está haciendo bastante popular, y, en mi opinión, tiene a hacer el código más legible.

Una vez que estas rutinas han sido definidas, podemos hacer llamadas a ellas cuantas veces queramos. Al procedimiento se le hace una llamada para conseguir que realice su tarea, y se hace una llamada a la función para calcular un valor :

**procedure** TForm1.Button1Click (Sender: TObject);

**begin**

Hello;

**end**;

**procedure** TForm1.Button2Click (Sender: TObject);

**var**

X, Y: Integer;

**begin**

X := Double (StrToInt (Edit1.Text));

Y := Double (X);

ShowMessage (IntToStr (Y));

**end**;

**Nota**: Por el momento, no se preocupe de la sintaxis de los dos procedimientos que aparecen arriba, que son, de hecho, métodos. Sencillamente, coloque dos botones en un formulario Delphi, haga clic en ellos durante el diseño, y el IDE de Delphi generará el código de apoyo adecuado: ahora sólo tiene que cumplimentar las líneas entre *begin* y *end*. Para compilar el código de arriba necesitará añadir un control de edición al formulario.

Ahora podemos volver al concepto de encapsulación de código que introduje antes. Cuando haga una llamada a la función Double, no necesita saber cuál es el algoritmo usado para implementarlo. Si más tarde encuentra un método mejor para doblar el valor de un número, puede cambiar el código asociado a la función, con facilidad, sin que se vea afectado el código de llamada a la función (¡aunque la ejecución será más rápida!). El mismo principio puede ser aplicado al procedimiento Hello: podemos modificar la salida del programa cambiando el código de este procedimiento, y el método Button2Click cambiará su efecto automáticamente. He aquí cómo podemos cambiar el código :

**procedure** Hello;

**begin**

MessageDlg (*'Hello world!'*, mtInformation, [mbOK]);

**end**;

**Consejo:** Cuando efectúa una llamada a una función o procedimiento Delphi, o a cualquier método VCL, debería recordar el número y tipo de los parámetros. El editor de Delphi le ayuda sugiriendo la lista de parámetros de una función o procedimiento con un *hint* emergente en cuanto usted introduce el nombre de la función o procedimiento, y abre paréntesis. Esta característica se llama Code Parameters (parámetros de código) y es parte de la tecnología Code Insight.

**Parámetros de referencia**

Las rutinas en Pascal permiten la transmisión de parámetros mediante valores y por referencias. Transmitir parámetros mediante valores es lo normal: el valor se copia a la pila (*stack*) y la rutina usa y maneja la copia, no el valor original.

Transmitir un parámetro por referencia significa que su valor no se copia en la pila del parámetro formal de la rutina (evitar una copia a menudo significa que el programa se ejecuta más rápido). En su lugar, el programa hace uso del valor original, también en el código de la rutina. Esto permite al procedimiento o función cambiar el valor del parámetro. La transmisión de parámetros por referencia se expresa con la palabra clave *var*.

Esta técnica está disponible en la mayoría de lenguajes de programación. No está presente en C, pero ha sido introducida en C++, donde se usa el símbolo & (transmisión por referencia). En Visual Basic, cada parámetro no especificado mediante ByVal es transmitido por referencia.

He aquí un ejemplo de transmisión de un parámetro por referencia mediante el uso de la palabra clave *var*:

Here is an example of passing a parameter by referene using the *var* keyword:

**procedure** DoubleTheValue (**var** Value: Integer);

**begin**

Value := Value \* 2;

**end**;

En este caso, el parámetro se usa tanto para transmitir un valor al procedimiento como para devolver un nuevo valor al código de llamada. Cuando se escribe ...

**var**

X: Integer;

**begin**

X := 10;

DoubleTheValue (X);

... el valor de la variable X se convierte en 20, porque la función usa una referencia a la posición de memoria original de X, afectando a su valor inicial.

La transmisión de parámetros por referencia tiene sentido para tipos ordinales, para cadenas tradicionales y para registros (*records*) de gran tamaño. Los objetos Delphi, de hecho, son siempre transmitidos mediante valores, porque ellos mismos son referencias. Por esta razón, transmitir un objeto por referencia tendría poco sentido (aparted de casos muy especiales), porque consistiría en transmitir una referencia a otra.

Las cadenas largas de Delphi tienen un comportamiento ligeramente distinto: se comportan como referencias, pero si se cambia una de las variables de cadena que apuntan a la misma cadena en memoria, esto se copia antes de actualizarla. Una cadena larga transmitida como parámetro de valor se comporta como una referencia sólo en términos de uso de la memoria y velocidad de la operación. Pero si modifica el valor de la cadena, el valor original no se ve afectado. Por el contrario, si transmite la cadena larga por referencia, puede alterar el valor original.

Delphi 3 introdujo un nuevo tipo de parámetro, *out*. Un parámetro *out* no tiene valor inicial, y se usa sólo para devolver un valor. Estos parámetros deberían usarse sólo para procedimientos y funciones COM; en general, es mejor ceñirse a los parámetros *var*, más eficientes. Excepto por no tener valor inicial, los parámetros out se comportan como parámetros *var*.

**Parámetros constantes**

Como alternative a los parámetros de referencia, puede usar un parámetro *const*. Ya que no puede asignar un nuevo valor a un parámetro constante dentro de una rutina, el compilador puede optimizar la transmisión de parámetros. El compilador puede elegir un enfoque similar para parámetros de referencia (o una referencia *const*, términos de C++), pero el comportamiento seguirá siendo similar a los parámetros de valor, porque el valor original no se verá afectado por la rutina.  
  
De hecho, si intenta compilar el siguiente código (trivial), Delphi producirá un error :

**function** DoubleTheValue (**const** Value: Integer): Integer;

**begin**

Value := Value \* 2; *// error de compilación*

Result := Value;

**end**;

**Parámetros de vector abierto (Open Array)**

Contrariamente a lo que sucede en C, una función o procedimiento Pascal siempre tiene un número fijo de parámetros. De cualquier forma, hay una manera de transmitir un número variable de parámetros a una rutina, utilizando un vector abierto.  
  
La definición básica de un parámetro de vector abierto es el de un vector abierto con tipo. Esto significa que se indica el tipo de parámetro, pero no se sabe cuántos elementos de tal tipo va a tener el vector. Aquí aparece un ejemplo de una definición así:

**function** Sum (const A: array of Integer): Integer;

**var**

I: Integer;

**begin**

Result := 0;

**for** I := Low(A) **to** High(A) **do**

Result := Result + A[I];

**end**;

Usando High(A) podemos averiguar el tamaño del vector. Tenga en cuenta también el uso del valor de salida de la función, Result, que consiste en almacenar valores temporales. Puede hacer una llamada a esta función transmitiéndole un valor de expresiones enteras (Integer) :

X := Sum ([10, Y, 27\*I]);

Dado un vector de enteros, de cualquier tamaño, puede transmitirlo directamente a una rutina que requiera un parámetro de vector abierto o, en su lugar, hacer una llamada a la función Slice para transmitir sólo una porción del vector (como lo indique su segundo parámetro). He aquí un ejemplo, donde el vector completo se transmite como parámetro:

**var**

List: **array** [1..10] **of** Integer;

X, I: Integer;

**begin**

*// initialize the array*

**for** I := Low (List) **to** High (List) **do**

List [I] := I \* 2;

*// call*

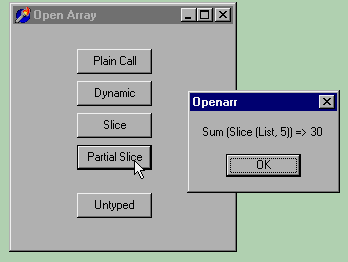
X := Sum (List);

Si desea transmitir sólo una porción del vector a la función Slice, simplemente efectúe la llamada de esta manera :

X := Sum (Slice (List, 5));

Podrá encontrar todos los fragmentos de código que aparecen en esta sección, dentro del ejemplo OpenArr (vea la Figura 6.1, más abajo, para el formulario)

***Figura 6.1: El ejemplo OpenArr, cuando se pulsa el botón Partial Slice***



Los vectores abiertos con tipo de Delphi 4 son totalmente compatibles con los vectores dinámicos (introducidos en Delphi 4 y discutidos en el Capíitulo 8). Los vectores dinámicos usan la misma sintaxis que los vectores dinámicos, con la diferencia de que puede usar una notación como *array of Integer* para declarar una variable, no sólo para transmitir un parámetro.

**Parámetros de vector abierto, de tipo variable**

Aparte de estos vectores abiertos con tipo, Delphi le permite definir vectores de tipo variable o sin tipo. Este tipo especial de vector tiene un número de valores indefinido, lo que puede ser más fácil de manejar a la hora de transmitir parámetros.  
  
Técnicamente, la forma vectorial *const* le permite transmitir un vector con un número indefinido de elementos de distintos tipos a una rutina, de una sola vez. Por ejemplo, he aquí la definición de la función Format (veremos cómo usarla en el [**Chapter 7**](http://www.marcocantu.com/epascal/Spanish/ch07str.htm), que trata de las cadenas):

function Format (const Format: string;

const Args: array of const): string;

El segundo parámetro es un vector abierto, que obtiene un número indefinido de valores. De hecho, puede hacer una llamada a tal función de las siguientes formas:

N := 20;

S := *'Total:'*;

Label1.Caption := Format (*'Total: %d'*, [N]);

Label2.Caption := Format (*'Int: %d, Float: %f'*, [N, 12.4]);

Label3.Caption := Format (*'%s %d'*, [S, N \* 2]);

Dése cuenta de que puede transmitir un parámetro bien como valor constante, el valor de una variable, o una expresión. Declarar una función de este tipo es sencillo, pero ¿cómo se codifica? ¿Cómo se sabe los tipos de los parámetros? Los valores de un parámetro de vector abierto de tipo variable son compatibles con los elementos de tipo *TVarRec*.

**Nota:** No confunda el registro *TVarRec* con el registro *TVarData* usado por el tipo Variant propiamente dicho. Estas dos estructuras tienen una finalidad distinta, y no son compatibles. Incluso la lista de tipos posibles es diferente, porque *TVarRec* puede contener tipos de datos de Delphi, mientras que *TVarData* puede contener tipos de datos OLE.

El registro *TVarRec* tiene la siguiente estructura :

**type**

TVarRec = **record**

**case** Byte **of**

vtInteger: (VInteger: Integer; VType: Byte);

vtBoolean: (VBoolean: Boolean);

vtChar: (VChar: Char);

vtExtended: (VExtended: PExtended);

vtString: (VString: PShortString);

vtPointer: (VPointer: Pointer);

vtPChar: (VPChar: PChar);

vtObject: (VObject: TObject);

vtClass: (VClass: TClass);

vtWideChar: (VWideChar: WideChar);

vtPWideChar: (VPWideChar: PWideChar);

vtAnsiString: (VAnsiString: Pointer);

vtCurrency: (VCurrency: PCurrency);

vtVariant: (VVariant: PVariant);

vtInterface: (VInterface: Pointer);

**end**;

Cada registro posible tiene el campo *VType*, aunque no es fácil de ver a primera vista, porque se declara sólo una vez, junto con los datos de tamaño entero, en sí (generalmente, una referencia o un puntero).

Usando esta información, podemos, de hecho, escribir una función capaz de operar sobre distintos tipos de datos. En él ejemplo de la función *SumAll*, quiero ser capaz de sumar valores de distintos tipos, transformando cadenas en enteros, caracteres en su valor ordinal correspondiente, y añadiendo 1 a los valores True Booleanos. El código está basado en una instrucción *case*, y es bastante simple, aunque tengamos que desreferenciar punteros a menudo :

**function** SumAll (**const** Args: **array of const**): Extended;

**var**

I: Integer;

**begin**

Result := 0;

**for** I := Low(Args) **to** High (Args) **do**

**case** Args [I].VType **of**

vtInteger: Result :=

Result + Args [I].VInteger;

vtBoolean:

**if** Args [I].VBoolean **then**

Result := Result + 1;

vtChar:

Result := Result + Ord (Args [I].VChar);

vtExtended:

Result := Result + Args [I].VExtended^;

vtString, vtAnsiString:

Result := Result + StrToIntDef ((Args [I].VString^), 0);

vtWideChar:

Result := Result + Ord (Args [I].VWideChar);

vtCurrency:

Result := Result + Args [I].VCurrency^;

**end**; *// case*

**end**;

He añadido este código al ejemplo OpenArr, que hace una llamada a la función *SumAll* cuando se pulsa un botón dado :

**procedure** TForm1.Button4Click(Sender: TObject);

**var**

X: Extended;

Y: Integer;

**begin**

Y := 10;

X := SumAll ([Y \* Y, *'k'*, True, 10.34, '99999']);

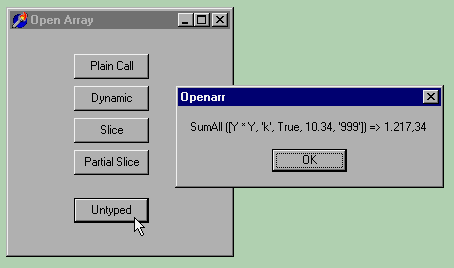
ShowMessage (Format (

*'SumAll ([Y\*Y, ''k'', True, 10.34, ''99999'']) => %n'*, [X]));

**end**;

Puede ver la salida de esta llamada, y el formulario del ejemplo OpenArr, en la Figura 6.2.

***Figura 6.2: El formulario form del ejemplo OpenArr, mostrando la ventana de mensajes cuando se pulsa el botón Untyped.***



**Convenios de llamada en Delphi**

La versión de 32 bits de Delphi ha introducido un nuevo enfoque a la transmisión de parámetros, llamada *fastcall*: cuandoquiera que sea posible, hasta tres parámetros pueden ser transmitidos en registros CPU, haciendo la llamada a la función mucho más rápida. El convenio de llamada rápida (usado por defecto en Delphi 3) se indica mediante la palabra clave *register*.

El problema es que este es el convenio por defecto, y las funciones que lo usan no son compatibles con Windows: las funciones de la API de Win32 tienen que ser declaradas usando el convenio de llamada *stdcall*, una mezcla del convenio de llamada original de Pascal para la API Win16 y el convenio de llamada *cdecl* del lenguaje C.  
  
No hay, en general, ninguna razón para no usar el nuevo convenio de llamada rápida, a no ser que haga llamadas externas Windows o defina funciones *callback* Windows. Veremos un ejemplo usando el convenio stdcall antes del fin de este capítulo. Podrá encontrar un sumario de los convenios de llamada de Delphi en la sección "Calling conventions" de la ayuda de Delphi.

**¿Qué es un método?**

Si ya ha trabajado con Delphi o ha leído los manuales, seguramente habrá oído hablar del término *método*. Un método es un tipo especial de función o procedimiento relacionado con un tipo de datos, una clase. En Delphi, cada vez que manejamos un evento, necesitamos definir un método, generalmente un procedimiento. En general, sin embargo, el término *método* se usa para indicar ambas funciones y procedimientos relacionados con una clase.

Ya hemos visto un cierto número de métodos en los ejemplos en este y los capítulos anteriores. Sigue un método vacío, añadido automáticamente por Delphi al código fuente de un formulario :

**procedure** TForm1.Button1Click(Sender: TObject);

**begin**

*{here goes your code}*

**end**;

**Declaraciones *forward***

Cuando necesite usar un identificador (de cualquier tipo), el compilador debe haber visto ya algún tipo de declaración para saber a qué apunta el identificador. Por esta razón, normalmente se proporciona una declaración completa antes de usar rutina alguna. De cualquier modo, hay casos en que esto no es posible. Si el procedimiento A hace una llamada al procedimiento B, y el procedimiento B hace lo propio con el A, cuando comience a escribir el código, tendrá que hacer una llamada a una rutina para la que el compilador aún no ha visto una declaración.

Si quiere declarar la existencia de un procedimiento o función con un cierto nombre y parámetros dados, sin proporcionar su código efectivo, puede escribir el procedimiento o función, seguido de la palabra clave *forward* :

**procedure** Hello; **forward**;

Más tarde, el código debería incluir una definición completa del procedimiento, pero este puede ser llamado incluso antes de ser totalmente definido. He aquí un ejemplo trivial, sólo para que se haga una idea :

**procedure** DoubleHello; **forward**;

**procedure** Hello;

**begin**

**if** MessageDlg (*'Do you want a double message?'*,

mtConfirmation, [mbYes, mbNo], 0) = mrYes **then**

DoubleHello

**else**

ShowMessage (*'Hello'*);

**end**;

**procedure** DoubleHello;

**begin**

Hello;

Hello;

**end**;

Este enfoque permite escribir recursividad mutua: *DoubleHello* hace una llamada a Hello, pero, por su parte, Hello podría llamar a *DoubleHello*. Por supuesto, tiene que haber una condición que ponga fin a la recursión, para evitar el desbordamiento de la pila. Puede encontrar este código, con ligeros cambios, en el ejemplo DoubleH.

Aunque la declaración de un procedimiento *forward* no es muy común en Delphi, hay un caso similar, que es mucho más frecuente. Cuando declara un procedimiento o función en la porción de la interfaz, dentro de una unidad (lea más sobre unidades en el [**siguiente capítulo**](http://www.marcocantu.com/epascal/Spanish/ch07str.htm)), se la considera una declaración *forward*, incluso aunque la palabra clave *forward* no esté presente. De hecho, no puede escribir el cuerpo de una rutina en la porción de la interfaz de una unidad. Al mismo tiempo, debe proporcionar en la misma unidad la implementación efectiva de cada rutina que haya declarado.  
  
Esto también es cierto para la declaración de un método en el interior de un tipo de clase que haya sido generado automáticamente por Delphi (al añadir un evento a un formulario o uno de sus componentes). Los gestores de eventos declarados en el interior de una clase *TForm* son declaraciones *forward*: el código será proporcionado en la porción de implementación de la unidad. He aquí un extracto del código fuente de un ejemplo anterior, con la declaración del método *Button1Click* :

**type**

TForm1 = **class**(TForm)

ListBox1: TListBox;

Button1: TButton;

**procedure** Button1Click(Sender: TObject);

**end**;

**Tipos de procedimiento**

Otra característica que hace único a Object Pascal es la presencia de tipos de procedimiento. Este tema es un aspecto realmente avanzado, que sólo unos pocos programadores en Delphi usarán regularmente. Sin embargo, ya que discutiremos temas relacionados en capítulos posteriores (en concreto, punteros de método, una técnica muy usada en Delphi), merece la pena echarle un vistazo rápido. Si es usted un programador en sus comienzos, puede saltarse esta sección por ahora, y volver a ella cuando se sienta preparado.

En Pascal, existe el concepto de tipo de procedimiento (que es similar al concepto de *puntero de función* del lenguaje C). La declaración de un tipo de procedimiento indica la lista de parámetros y, en el caso de las funciones, el tipo de salida. Por ejemplo, se puede declarar el tipo de procedimiento con un parámetro Integer transmitido por referencia como :

**type**

IntProc = **procedure** (**var** Num: Integer);

Este tipo de procedimiento es compatible con cualquier rutina que tenga exactamente los mismos parámetros (o la misma firma de función (*function signature*), parafraseando el argot de C). Sigue un ejemplo de una rutina compatible:

**procedure** DoubleTheValue (**var** Value: Integer);

**begin**

Value := Value \* 2;

**end**;

Nota: En la versión de 16 bits de Delphi, las rutinas deben ser declaradas usando la directiva *far*, para poder utilizadas como valores efectivos de un tipo de procedimiento.

Los tipos de procedimiento pueden ser utilizados con dos fines diferentes: puede declarar variables de un tipo de procedimiento o transmitir un tipo de procedimiento --un puntero de función-- como parámetro a otra rutina. Dadas las declaraciones precedentes de tipo y procedimiento, puede escribir este código:

**var**

IP: IntProc;

X: Integer;

**begin**

IP := DoubleTheValue;

X := 5;

IP (X);

**end**;

Este código tiene el mismo efecto que la versión siguiente, más breve :

**var**

X: Integer;

**begin**

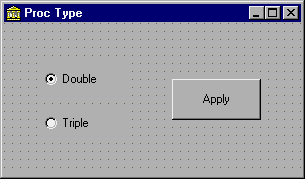
X := 5;

DoubleTheValue (X);

**end**;

La primera versión es claramente más compleja, así que ¿por qué deberíamos usarla? En algunos casos, ser capaz de decidir a qué función se debe hacer una llamada y efectuar ésta sólo más tarde, puede ser útil. Es posible construir un ejemplo complejo, usando este enfoque. Sin embargo, prefiero dejarle que explore uno bastante sencillo, que he llamado ProcType. Este ejemplo es más complejo que los que hemos visto hasta ahora, para hacer la situación un poco más realista.  
  
Sencillamente, cree un proyecto vacío, y coloque en él dos casillas circulares (*radio buttons*) y un botón, como se muestra en la Figura 6.3. Este ejemplo se basa en dos procedimientos. Uno de ellos se usa para doblar el valor del parámetro. Este procedimiento es similar a la versión que ya mostré en esta sección. Un segundo procedimiento se usa para triplicar el valor, y, por tanto, se llama TripleTheValue:

***Figura 6.3: El formulario del ejemplo ProcType.***



**procedure** TripleTheValue (**var** Value: Integer);

**begin**

Value := Value \* 3;

ShowMessage (*'Value tripled: '* + IntToStr (Value));

**end**;

Ambos procedimientos muesrtran lo que está pasando, para hacernos saber que han sido llamados. Esta es una característica de depuración simple que puede usar para comprobar si y cuándo una cierta porción de código es ejecutada, en vez de añadir una interrupción para averiguarlo.

Cada vez que el usuario pulsa el botón Apply, uno de las dos procedimientos es ejecutado, dependiendo del estado de las casillas circulares. De hecho, cuando se tienen dos casillas circulares en un formulario, sólo una de ellas puede estar activada a la vez.   
Este código podría haber sido implementado comprobando el valor de las casillas circulares dentro del código correspondiente al evento OnClick del botón Apply. Para mostrar el uso de tipos de procedimiento, en su lugar, he usado un enfoque más largo, pero interesante. Cada vez que el usuario hace clic en una de las casillas circulares, uno de los procedimientos se almacena en una variable:

**procedure** TForm1.DoubleRadioButtonClick(Sender: TObject);

**begin**

IP := DoubleTheValue;

**end**;

Cuando el usuario hace clic en el botón, el procedimiento que hemos almacenado, se ejecuta :

**procedure** TForm1.ApplyButtonClick(Sender: TObject);

**begin**

IP (X);

**end**;

Para permitir que tres funciones distintas accedan a las variable IP y X, tenemos que hacerlas visibles a todo el formulario; no pueden ser declaradas de forma local (dentro de uno de los métodos). Una solución a este problema es situar estas variables dentro de la declaración del formulario :

**type**

TForm1 = **class**(TForm)

...

**private**

*{ Private declarations }*

IP: IntProc;

X: Integer;

**end**;

Veremos qué significa esto exactamente en el [**próximo capítulo**](http://www.marcocantu.com/epascal/Spanish/ch07str.htm), pero, por el momento, necesitará modificar el código generado por Delphi para el tipo de clase, como se indica arriba, y añadir la definición del tipo de procedimiento que mostré anteriormente. Para inicializar estas dos variables con valores adecuados, podemos manejar el evento *OnCreate* del formulario (seleccione este evento en el Object Inspector después de activar el formulario, o simplemente haga doble clic en el mismo). Le sugiero que consulte el listado para estudiar los detalles del código fuente de este ejemplo.

Puede ver un ejemplo práctico del uso de tipos de procedimiento en el Capítulo 9, en la sección [**"Funciones *callback* de Windows"**](http://www.marcocantu.com/epascal/Spanish/ch09win.htm#callback).

**Sobrecarga de funciones**

La idea de sobrecarga es sencilla: el compilador le permite definir dos funciones o procedimientos usando el mismo nombre, supuesto que los parámetros sean distintos. Al comprobar los parámetros, de hecho, el compilador puede determinar a cuál de las versiones de la rutina se refiere usted.

Considere esta sucesión de funciones extraída de la unidad Math de la VCL:

**function** Min (A,B: Integer): Integer; **overload**;

**function** Min (A,B: Int64): Int64; **overload**;

**function** Min (A,B: Single): Single; **overload**;

**function** Min (A,B: Double): Double; **overload**;

**function** Min (A,B: Extended): Extended; **overload**;

Cuando efectúe una llamada a Min (10, 20), el compilador determina fácilmente que está llamando a la primera función del grupo, con lo que el valor de salida será un entero (Integer).

Las reglas básicas son dos :

* Cada versión de la rutina debe ser seguida por la palabra clave *overload*.
* Las diferencias deben estar en el número o tipo de los parámetros, o en ambos. El tipo de salida, por sus parte, no puede ser utilizado para distinguir entre dos rutinas.

He aquí tres versiones sobrecargadas de un procedimiento ShowMsg que añadí al ejemplo OverDef (una aplicación que hace evidente la sobrecarga y parámetros por defecto):

**procedure** ShowMsg (str: string); **overload**;

**begin**

MessageDlg (str, mtInformation, [mbOK], 0);

**end**;

**procedure** ShowMsg (FormatStr: string;

Params: **array of const**); **overload**;

**begin**

MessageDlg (Format (FormatStr, Params),

mtInformation, [mbOK], 0);

**end**;

**procedure** ShowMsg (I: Integer; Str: string); **overload**;

**begin**

ShowMsg (IntToStr (I) + *' '* + Str);

**e**nd;

Las tres funciones muestran una ventana de mensajes con una cadena, después de dar formato a la cadena de distintas maneras. Siguen las tres llamadas del programa :

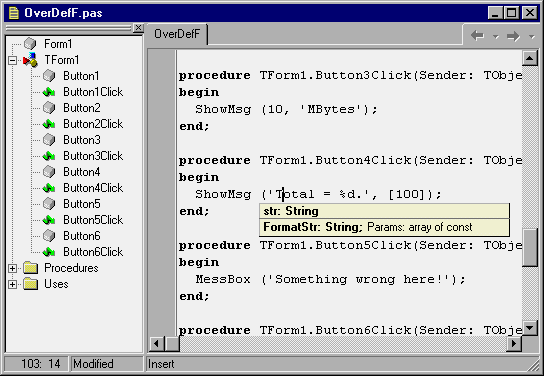
ShowMsg (*'Hello'*);

ShowMsg (*'Total = %d.'*, [100]);

ShowMsg (10, *'MBytes'*);

Lo que me sorprendió de forma positiva es que la tecnología de parámetros de código de Delphi (Code Parameters) funciona muy bien con procedimientos y funciones sobrecargados. Al introducir el paréntesis abierto tras el nombre de la rutina, se enumeran todas las alternativas disponibles. Al introducir los parámetros, Delphi usa su tipo para determinar cuál de las alternativas están aún disponibles. En la Figura 6.4 se puede ver, tras comenzar a teclear una cadena constante, que Delphi muestra sólo las versiones compatibles (omitiendo la versión del procedimiento ShowMsg que tiene un entero como primer parámetro).

***Figura 6.4: Las múltiples alternativas ofrecidas por Code Parameters para rutinas sobrecargadas son filtrados de acuerdo con los parámetros que ya están disponibles.***



El hecho de que cada versión de una rutina sobrecargada debe ser marcada adecuadamente implica que no se puede sobrecargar una rutina existente de la misma unidad que no esté marcada con la palabra clave *overload*. (El mensaje de error que se obtiene cuando se intenta es "Previous declaration of '<name>' was not marked with the 'overload' directive" -- la declaración anterior de <nombre> no está marcada con la directiva 'overload'.) Sin embargo, puede sobrecargar una rutina que fue declarada originalmente en una unidad distinta. Esto se hace para conseguir compatibilidad con versiones anteriores de Delphi, que permitían que distintas unidades usasen el mismo nombre de rutina varias veces. Observe, de cualquier modo, que este caso especial no es una característica extra de la sobrecarga, sino una indicación de los problemas con que uno se puede encontrar.

Por ejemplo, puede añadir el siguiente código a una unidad :

**procedure** MessageDlg (str: string); **overload**;

**begin**

Dialogs.MessageDlg (str, mtInformation, [mbOK], 0);

**e**nd;

Este código realmente no sobrecarga la rutina MessageDlg original. De hecho, si escribimos ...

MessageDlg (*'Hello'*);

... obtendremos un simpático mensaje de error, que indicará que algunos de los parámetros faltan. La única manera para efectuar una llamada a la versión local en vez de a la de la VCL es aludir explícitamente a la unidad local -- nada más lejos de la idea de sobrecarga :

OverDefF.MessageDlg (*'Hello'*);

**newParámetros por defecto**

Una característica nueva de Delphi 4 es que se puede dar un valor por defecto al parámetro de una función, y se puede hacer una llamada a la función con o sin dicho parámetro. Déjeme mostrarle un ejemplo. Podemos definir el siguiente encapsulamiento del método MessageBox del objeto global Application, que usa PChars en vez de cadenas, proveyendo dos parámetros por defecto:

**procedure** MessBox (Msg: string;

Caption: string = 'Warning';

Flags: LongInt = mb\_OK **or** mb\_IconHand);

**begin**

Application.MessageBox (PChar (Msg),

PChar (Caption), Flags);

**end**;

Con esta definición, podemos hacer una llamada al procedimiento en cualquiera de las siguientes formas :

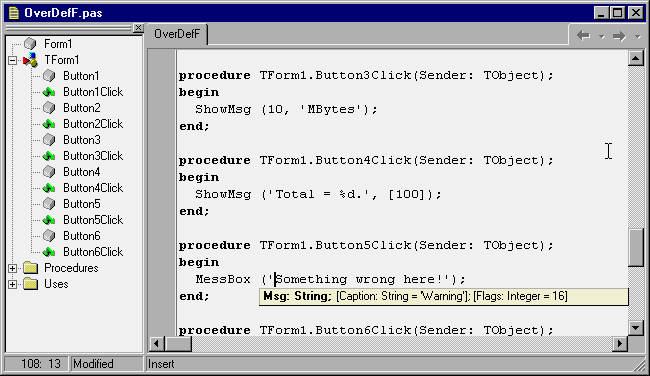
MessBox (*'Something wrong here!'*);

MessBox (*'Something wrong here!'*, *'Attention'*);

MessBox (*'Hello'*, *'Message'*, mb\_OK);

En la Figura 6.5 puede verse que el Code Parameters de Delphi usan un estilo diferente para indicar los parámetros que tienen un valor por defecto, de manera que se puede determinar con facilidad qué parámetros pueden omitirse.

***Figura 6.5: El Code Parameters marca con corchetes los parámetros que tienen valores por defecto; se pueden omitir éstos en la llamada.***



Observe que Delphi no genera código especial para ser compatible con los parámetros por defecto, ni tampoco crea copias múltiples de las rutinas. Los parámetros que faltan son, sencillamente, añadidos por el compilador al código de la llamada.

Hay una restricción importante que afecta al uso de los parámetros por defecto : no se puede uno "saltar" los parámetros. Por ejemplo, no es posible transmitir el tercer parámetro a la función tras haber omitido el segundo :

MessBox (*'Hello'*, mb\_OK); *// error*

Esta es la regla principal para los parámetros por defecto: en una llamada, sólo se puede omitir parámetros comenzando por el último. En otras palabras, para omitir un parámetro habrá de omitir también los siguientes.

Hay unas pocas reglas más para los parámetros por defecto :

* Los parámetros con valores por defecto deben estar al final de la lista de parámetros.
* Los valores por defecto deben ser constantes. Obviamente, esto limita los tupos que pueden usarse con los parámetros por defecto. Por ejemplo, un vector dinámico o un tipo de interfaz no puede tener un parámetro por defecto distinto de *nil*; además, los registros no pueden usarse.
* Los parámetros defecto tienen que ser transmitidos por valor o como *const*. Un parámetro de referencia (*var*) parameter no tiene nunca valor por defecto.

La presencia simultánea de parámetros por defecto y sobrecargas puede causar bastantes problemas, ya que estas características podrían entrar en conflicto. Por ejemplo, si añado al ejemplo anterior la siguiente nueva versión del procedimiento *ShowMsg* ...

**procedure** ShowMsg (Str: string; I: Integer = 0); **overload**;

**begin**

MessageDlg (Str + *': '* + IntToStr (I),

mtInformation, [mbOK], 0);

**end**;

... entonces, el compilador no se quejará -- se trata de una definición legal. Sin embargo, la llamada ...

ShowMsg ('Hello');

... es señalada por el compilador como Llamada sobrecargada ambigua a 'ShowMsg' (Ambiguous overloaded call to 'ShowMsg'). Dése cuenta de que este error aparece en una línea de código que se había compilado correctamente antes de la nueva definición sobrecargada. En la práctica, no hay manera de efectuar una llamada al procedimiento ShowMsg con un solo parámetro de cadena, ya que el compilador no sabría si queremos llamar a la versión que tiene sólo el parámetro de cadena o a aquella que tiene dicho parámetro más un parámetro entero con un valor por defecto.   
Cuando tiene tal duda, el compilador para y pide al programador que aclare sus intenciones.

**Conclusión**

Escribir procedimientos y funciones es un elemento clave de la programación, aunque en Delphi tenderá a escribir *métodos* -- procedimientos y funciones, conectados con clases y objetos.

En vez de pasar a discutir las características de orientación a objetos, los capítulos que siguen inmediatamente le darán algunos detalles sobre otros elementos de programación en Pascal, comenzando con las cadenas.

**Capítulo siguiente:** [**Manejo de cadenas (de caracteres)**](http://www.marcocantu.com/epascal/Spanish/ch07str.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 7 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch07str.htm) Manejo de cadenas** |

El manejo de cadenas de caracteres en Delphi es bastante sencillo, pero entre bastidores la situación es bastante compleja. Pascal sigue la forma tradicional de manejar cadenas, Windows tiene su propia manera de hacerlo, prestada del lenguaje C, y las versiones de 32 bits de Delphi incluyen un potente tipo de datos de cadena larga, que constituye el tipo de cadena por defecto en ese lenguaje.

**Tipos de cadenas (*strings*)**

En el Turbo Pascal de Borland y en el Delphi de 16 bits, el tipo de cadena típico es una sucesión de caracteres con un octeto (*byte*) al principio de aquella. Como la longitud se expresa en un solo byte, no puede exceder de 255 caracteres; pero este valor tan bajo crea muchos problemas a la hora de manipular cadenas. Cada cadena viene definida por un tamaño fijo (que, por defecto, es el máximo, 255), aunque puede declarar cadenas más cortas, para ahorrar espacio de memoria.

Un tipo *cadena* es muy similar a un tipo *vector*. De hecho, una cadena es casi un vector formado por caracteres. Esto se muestra en el hecho de que puede acceder a una cadena específica usando la notación [].

Para superar los límites de las cadenas en Pascal tradicionales, las versiones de Delphi en 32 bits permiten la existencia de cadenas largas. Hay, de hecho, tres tipos de cadenas:

* El tipo ShortString (*cadena corta*) correponde a las cadenas tradicionales de Pascal, descritas anteriormente. Estas cadenas tienen un límite de 255 caracteres y se corresponden con las cadenas en la versión de Delphi de 16 bits. Cada elemento de una cadena corta es del tipo ANSIChar (el tipo de carácter normal).
* El tipo ANSIString corresponde a las nuevas cadenas largas, de longitud variable. Estas cadenas son asignadas de forma dinámica, incluyen un contador de referencias, e implementan la técnica de copia por escritura (*copy-on-write*). El tamaño de tales cadenas es casi ilimitado (¡pueden almacenar hasta dos millardos de caracteres!). También están basados en el tipo ANSIChar.
* El tipo WideString es similar al ANSIString, pero está basado en el WideChar - almacena caracteres Unicode.

**Uso de cadenas largas**

Si sólo usamos el tipo de datos cadenas, obtenemos bien cadenas cortas o cadenas ANSI, dependiendo del valor de la directiva $H del compilador. $H+ (la opción por defecto) representa cadenas largas (el tipo ANSIString), que es lo que usan las componentes de la biblioteca Delphi.

Las cadenas largas en Delphi se basan en un mecanismo de contado de referencias, que contabiliza cuántas variables acceden a una misma cadena en la memoria. Este contado de referencias se usa también para liberar memoria cuando una cadena ya no se usa - esto es, cuando el contador de referencia alcanza el valor cero.

Si desea incrementar el tamaño de una cadena en la memoria pero hay otra cosa en la memoria adyacente, la cadena no puede crecer en esa posición, y por tanto debe hacerse una copia completa de la cadena en otro emplazamiento. Cuando ocurre esta situación, los algoritmos de ejecución de Delphi reasignan la variable por nosotros, de forma totalmente transparente. Simplemente establezca el tamaño de la cadena mediante el procedimiento SetLength, adjudicando de forma efectiva la cantidad requerida de memoria:

SetLength (String1, 200);

En realidad, el procedimiento SetLength realiza una petición de memoria, no una asignación directa de memoria. Reserva la cantidad de memoria requerida para usarla después, sin usar la memoria de hecho. Esta técnica se basa en una característica de los sistemas operativos Windows y es usado por Delphi en todas las asignaciones dinámicas de memoria. Por ejemplo, cuando establece un vector muy grande, su memoria se reserva, pero no se adjudica.  
  
Fijar la longitud de una cadena es raramente necesario. El único caso en que se debe asignar memoria a cadenas largas mediante SetLength es cuando hay que utilizarlas como parámetros para una función API (usando el *typecast* o modelado correspondiente), como le mostraré en breve.

**Un vistazo a las cadenas en memoria**

Para ayudarle a entender mejor los detalles de la gestión de memoria para las cadenas, he escrito un sencillo ejemplo StrRef. En este programa declaro dos cadenas globales: Str1 y Str2. Cuando se pulsa el primero de los dos botones, el programa asigna una cadena constante a la primera de las variables, y entonces asigna la segunda variable a la primera.

Str1 := *'Hello'*;

Str2 := Str1;

Aparte de trabajar con las cadenas, el programa muestra su estado interno en una casilla de lista, usando la siguiente función StringStatus :

**function** StringStatus (**const** Str: string): string;

**begin**

Result := *'Address: '* + IntToStr (Integer (Str)) +

*', Length: '* + IntToStr (Length (Str)) +

*', References: '* + IntToStr (PInteger (Integer (Str) - 8)^) +

*', Value: '* + Str;

**end**;

Es de importancia vital en la función *StringStatus* transmitir el parámetro de cadena como *const*. Transmitir este parámetro mediante copiado tendrá el efecto secundario de hacer una referencia adicional a la cadena mientras la función se ejecute. Por el contrario, transmitir el parámetro mediante un parámetro de referencia (*var*) o constante (*const*) no implica ulterior referencia a la cadena. En este caso usé un parámetro *const*, ya que no se espera de la función que modifique la cadena.

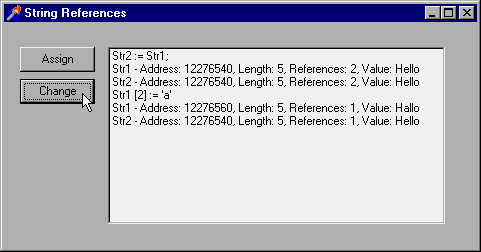
Para obtener la dirección de memoria de la cadena (útil para determinar su identidad actual y ver cuándo dos cadenas diferentes se refieren a la misma área de memoria), hice un *moldeado* (typecast) del tipo cadena al tipo entero, sin más. Las cadenas son, en la práctica, referencias, son punteros. Su valor contiene la posición en la memoria de la cadena.

Para extraer el contador de referencias, he basado el código en el hecho poco conocido de que el contador de la longitud y de referencias son, en realidad, almacenados en la cadena, antes del texto que la compone, y antes de la posición a que apunta la variable de cadena. El *offset* (negativo) es -4 para la longitud de una cadena (un valor que puede averiguarse más fácilmente usando la función Length), y -8 para el contador de referencias.

Tenga presente que esta información interna sobre offsets puede llegar a cambiar en futuras versiones de Delphi; tampoco hay garantía de que características similares que no aparecen en la documentación sigan disponibles en el futuro.

Ejecutando este ejemplo, debería obtener dos cadenas con el mismo contenigo, la misma posición de memoria, y un contador de referencias de valor 2, como se muestra en la parte superior de la casilla de lista, en la Figura 2.1. Ahora, si cambia el valor de una de las cadenas (no importa cuál), la posición de memoria de la cadena actualizada cambiará. Este es el efecto de la técnica de copia por escritura (*copy-on-write*).

***Figura 7.1: El ejemplo StrRef muestra el estado interno de dos cadenas, incluyendo el contador de referencia actual.***



Podemos, de hecho, producir este efecto, mostrado en la segunda parte de la casilla de lista de la Figura 7.1, escribiendo el siguiente código para el gestor del evento OnClick del segundo botón :

**procedure** TFormStrRef.BtnChangeClick(Sender: TObject);

**begin**

Str1 [2] := *'a'*;

ListBox1.Items.Add (*'Str1 [2] := ''a'''*);

ListBox1.Items.Add (*'Str1 - '* + StringStatus (Str1));

ListBox1.Items.Add (*'Str2 - '* + StringStatus (Str2));

**end**;

Observe que el código del método *BtnChangeClick* puede ser ejecutado sólo después del método *BtnAssignClick*. Para reforzar esto, el programa comienza con el segundo botón desactivado (su propiedad Enabled se establece como False); activa el botón al final del primer método. Puede usted extender este ejemplo y usar la función StringStatus para explorar el comportamiento de cadenas largas en muchas otras circunstancias.

**Cadenas Delphi y el PChars de Windows**

Otro punto importante a favor del uso de cadenas largas es que son de terminación nula. Esto significa que son totalmente compatibles con las cadenas de terminación nula copiadas por Windows del lenguaje C. Una cadena de terminación nula es una sucesión de caracteres seguida por un byte a que se da valor cero (o nulo). Esto puede expresarse en Delphi usando un vector de caracteres comenzado en cero, como suele hacerse para implementar cadenas en el lenguaje C. Esta es la razón por la que los vectores de caracteres de terminación nula son tan comunes en las funciones API de Windows (basadas en el lenguaje C). Como las cadenas largas de Pascal son totalmente compatibles con las cadenas de terminación nula en C, se puede, sencillamente, usar cadenas largas y moldearlas (*typecast*) a PChar cuando sea necesario transmitir una cadena a una función API de Windows.

Por ejemplo, para copiar el título de un formulario a una cadena PChar (usando la función API GetWindowText) y luego copiarla a la leyenda de un botón, se puede escribir el siguiente código :

**procedure** TForm1.Button1Click (Sender: TObject);

**var**

S1: String;

**begin**

SetLength (S1, 100);

GetWindowText (Handle, PChar (S1), Length (S1));

Button1.Caption := S1;

**end**;

Encontrará este código en el ejemplo LongStr. Observe que si escribe este código pero no asigna la memoria a la cadena con SetLength, el programa probablemente se derrumbará. Si usa PChar para transmitir un valor (y no para recibir un valor, como en el ejemplo de arriba), el código es aún más simple, porque no hay necesidad de definir una cadena temporal y de inicializarla. La siguiente línea de código transmite la propiedad Caption de una etiqueta como parámtro a una función API, sencillamente moldeándola a PChar:

SetWindowText (Handle, PChar (Label1.Caption));

Cuando necesite moldear una cadena WideString a un tipo compatible Windows, deberá usar PWideChar en vez de PChar para dicha conversión. *Wide strings* se usan a menudo para programas OLE y COM.

Habiendo presentado la parte agradeable de este asunto, ahora quiero centrarme en los defectos. Hay algunos problemas que podrían aparecer al convertir una cadena larga en un PChar. Esencialmente, el problema subyacente es que tras esta conversión usted se hacer responsable de la cadena y su contenido, y Delphi no le ayudará más en ello. Observe el siguiente pequeño cambio hecho al primer fragmento de código que aparece arriba, Button1Click:

**procedure** TForm1.Button2Click(Sender: TObject);

**var**

S1: String;

**begin**

SetLength (S1, 100);

GetWindowText (Handle, PChar (S1), Length (S1));

S1 := S1 + *' is the title'*; *// esto no funciona*

Button1.Caption := S1;

**end**;

Este programa se compila, pero cuando lo ejecute, se llevará una sorpresa: la leyenda del botón llevará el texto original del título de la ventana, sin el texto de la cadena constante que se le añadió. El problema consiste en que cuando Windows escribe en la cadena (dentro de la llamada a la API GetWindowText), no establece adecuadamente la longitud de la cadena larga Pascal. Delphi aún puede usar esta cadena como salida y hacerse una idea de dónde finaliza, pudiendo buscar la terminación nula; pero si usted le añade más caracteres tras la terminación nula, serán ignorados.

¿ Cómo podemos resolver este problema ? La solución pasa por decirle al sistema que convierta la cadena devuelta por la llamada a la API GetWindowText, en una cadena tipo Pascal. Sin embargo, si escribe el siguiente código ...

S1 := String (S1);

... el sistema lo ignorará, porque convertir un tipo de datos en sí mismo es una operación inútil. Para obtener la cadena larga tipo Pascal adecuada, tendrá que remoldear la cadena a un PChar y dejar que Delphi la vuelva a convertir en una cadena :

S1 := String (PChar (S1));

De hecho, puede ahorrarse la conversión a cadena, porque las conversiones de PChar a cadena son automáticas en Delphi. Esta es la versión final del código :

**procedure** TForm1.Button3Click(Sender: TObject);

**var**

S1: String;

**begin**

SetLength (S1, 100);

GetWindowText (Handle, PChar (S1), Length (S1));

S1 := String (PChar (S1));

S1 := S1 + *' is the title'*;

Button3.Caption := S1;

**end**;

Una alternativa es restablecer la longitud de la cadena tipo Delphi, usando la longitud de la cadena PChar, escribiendo :

SetLength (S1, StrLen (PChar (S1)));

Encontrará tres versiones de este código en el ejemplo LongStr, que tiene tres botones para ser ejecutado. De cualquier forma, si sólo tiene que acceder al título de un formulario, puede simplemente utilizar la propiedad Caption del objeto formulario mismo. No es necesario escribir todo este código, que lleva a confusión. Sólo lo hemos mostrado para ilustrar los problemas de la conversión de cadenas. Hay casos prácticos en que necesitará hacer llamadas a funciones API, y entonces tendrá que considerar esta compleja situación.

**Formateo de cadenas**

Usando el operado 'más' (+) y algunas de las funciones de conversión (como IntToStr) se puede construir cadenas complejas a partir de valores existentes. Sin embargo, hay otra forma de enfocar el formateo de números, valores de unidades monetarias, y otras cadenas, para conseguir una cadena final. Puede usar la potente función Format o una de sus funciones anejas.

La función Format requiere como parámetros una cadena con el texto básico y algunos marcadores de formato (normalmente indicados mediante el prefijo %) y un vector de valores, uno por cada marcador. Por ejemplo, para convertir el formato de dos números en el de una cadena, se puede escribir :

Format (*'First %d, Second %d'*, [n1, n2]);

donde n1 y n2 son dos valores enteros (Integer). El primer marcador se reemplaza por el primer valor, el segundo por el segundo, y así sucesivamente. Si el tipo de salida del marcador (indicado por la letra que sigue al %) no encaja con el tipo del parámetro correspondiente, se produce un error de ejecución. No disponer de ninguna comprobación durante la compilación es, de hecho, la mayor desventaja de usar la función Format.

La función Format usa un parámetro de vector abierto (un parámetro que puede tener un número arbitrario de valores), algo que discutiré hacia el final del presente capítulo. Por el momento, sin embargo, tenga en cuenta que sólo sintaxis de tipo vector de la lista de valores que se transmite como segundo parámetro.

Aparte de usar %d, se puede hacer uso de muchos otros marcadores definidos por esta función, y que se enumeran en la Tabla 7.1. Estos marcadores proporcionan una salida por defecto para el tipo de datos dado. De cualquier modo, puede usar especificadores de formatos para alterar la salida por defecto. Un especificador de anchura, por ejemplo, determina un número fijo de caracteres en la salida, mientras un especificador de precisión especifica el número de cifras decimales. Por ejemplo,

Format ('%8d', [n1]);

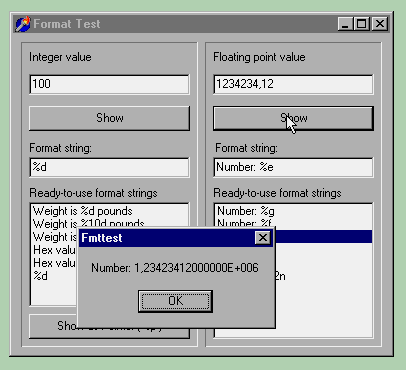
convierte el número n1 en una cadena de ocho caracteres, alineando el texto a la derecha (use el símbolo 'menos' para especificar justificación a la izquierda), llenándola con espacios en blanco.

***Tabla 7.1: Especificadores de tipo, para la función Format.***

|  |  |
| --- | --- |
| **ESPECIFICADOR DE TIPO** | **DESCRIPCIÓN** |
| d (decimal) | El valor entero correspondiente se convierte en una cadena de cifras decimales. |
| x (hexadecimal) | El valor entero correspondiente se convierte en una cadena de cifras hexadecimales. |
| p (puntero) | El valor de puntero correspondiente se convierte en una cadena de cifras hexadecimales. |
| s (cadena) | La cadena, carácter, o PChar correspondiente se se copia a la cadena de salida. |
| e (exponencial) | El valor de coma flotante correspondiente se convierte en una cadena escrita en notación exponencial. |
| f (coma flotante) | El valor de coma flotante correspondiente se convierte en una cadena escrita en notación de coma flotante. |
| g (general) | El valor de coma flotante correspondiente se convierte en una cadena escrita en notación decimal, lo más corta posible. |
| n (número) | El valor de coma flotante correspondiente se convierte en una cadena escrita en notación de coma flotante con separadores de millar. |
| m (dinero) | El valor correspondiente de coma flotante se convierte en una cadena que representa una cantidad en cierta moneda. La conversión se basa en las opciones regionales -- vea la el apartado "Currency and date/time formatting variables" ayuda de Delphi. |

El mejor camino para ver ejemplos de estas conversiones es experimentar con cadenas de formato usted mismo. Para hacer esto más sencillo, he escrito el programa FmtTest, que permite al usuario proporcionar cadenas de formato para números enteros y de coma flotante. Como puede observar en la Figura 7.2, este programa muestra un formulario dividido en dos partes. La parte izquierda es para números enteros (Integer), la derecha para números de coma flotante.

En cada parte hay una primera casilla de edición con el valor numérico que quiere cambiar a formato de cadena. Bajo la primera casilla de edición hay un botón para realizar la operación de formato y mostrar el resultado en una ventana de mensajes. Aparte hay otra casilla de edición, donde puede escribir una cadena de formato. Como alternativa, puede, sencillamente, hacer clic en una de las líneas del componente ListBox, más abajo, para seleccionar una cadena de formateo predefinida. Cada vez que introduzca una nueva cadena de formateo, se añade a la correspondiente casilla de lista (advierta que cerrando el programa perderá estos nuevos ítems).

***Figura 7.2: La salida de un valor de coma flotante del programa FmtTest.  
***

El código de este ejemplo simplemente usa el texto de los varios controles para producir la salida. Este es uno de los tres métodos conectados con los botones Show :

**procedure** TFormFmtTest.BtnIntClick(Sender: TObject);

**begin**

ShowMessage (Format (EditFmtInt.Text,

[StrToInt (EditInt.Text)]));

*// si el elemento no está, añádelo*

**if** ListBoxInt.Items.IndexOf (EditFmtInt.Text) < 0 **then**

ListBoxInt.Items.Add (EditFmtInt.Text);

**end**;

El código, básicamente, efectúa la operación de formateo, usando el texto de la casilla de edición EditFmtInt y el valor del control EditInt. Si la cadena de formato no está ya en la casilla de lista, se añade a la misma. Si el usuario, en lugar de ello, hace clic en un elemento de la casilla de lista, el código traslada tal valor a la casilla de edición :

**procedure** TFormFmtTest.ListBoxIntClick(Sender: TObject);

**begin**

EditFmtInt.Text := ListBoxInt.Items [

ListBoxInt.ItemIndex];

**end**;

**Conclusión**

Las cadenas de texto son, ciertamente, un tipo de datos muy común. Aunque se pueden usar sin reparos en la mayoría de los casos, sin entender cómo funcionan, este capítulo debe de haber dejado claro cómo se comportan exactamente las cadenas, haciendo posible usar todo el poder de este tipo de datos.0

Las cadenas se manejan en la memoria de una forma dinámica especial, como ocurre con los vectores dinámicos. Este es el tema del capítulo que sigue.

**Capítulo siguiente:** [**Memoria**](http://www.marcocantu.com/epascal/Spanish/ch08mem.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 8 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch08mem.htm) Memoria** |

**Nota del autor:** En un futuro, este capítulo cubrirá el manejo de la memoria, discutirá varias áreas de memoria, e introducirá vectores dinámicos. Por el momento, sólo esta última parte está disponible.

**Vectores dinámicos en Delphi 4**

Traditionalmente, en el lenguaje Pascal siempre se usaron vectores de tamaño fijo. Cuando se declara un tipo de dato usando el formato vectorial (*array*), se tiene que especificar el número de elementos del vector. Como saben los programadores expertos, existen unas pocas técnicas que pueden utilizarse para implementar vectores dinámicos, normalmente usando punteros y adjudicando y liberando manualmente la memoria requerida.

Delphi 4 introduce una implementación muy simple de los vectores dinámicos, modelándolos mediante el tipo dinámico de cadena larga que expuse anteriormente. Como las cadenas largas, los vectores dinámicos se adjudican de forma dinámica, e incluyen un contador de referencias, aunque no implementan la técnica de copia por escritura (*copy-on-write*). Esto no supone gran problema, ya que se puede vaciar un vector declarando su variable nula (*nil*).

Ahora, puede declarar un vector de forma simple, sin tener que especificar el número de elementos y luego adjudicarlos con un tamaño dado, utilizando el procedimiento SetLength. El mismo procedimiento nos servirá para cambiar el tamaño de un vector sin perder su contenido. Hay otros procedimientos referidos a las cadenas, como la función Copy, que pueden utilizarse sobre los vectores.

He aquí una pequeña muestra de código, sin mostrar la necesidad de declarar y adjudicar memoria al vector antes de empezar a usarlo:

**procedure** TForm1.Button1Click(Sender: TObject);

**var**

Array1: **array of** Integer;

**begin**

Array1 [1] := 100; *// error*

SetLength (Array1, 100);

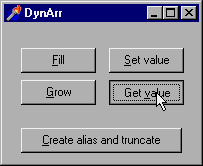
Array1 [99] := 100; *// OK*

...

**end**;

Si sólo se indica el número de elementos del vector, el índice siempre comienza a contar desde el 0. Los vectores genéricos en Pascal pueden tener cota inferior no nula e índices no enteros. Este no es el caso en los vectores dinámicos. Para averiguar la condición de un vector dinámico, puede usar las funciones Length, High y Low, como con cualquier vector. Sin embargo, en el caso de vectores dinámicos, Low siempre devuelve el valor 0, y High el valor menos uno. Esto implica que para un vector vacío High vale -1 (que, si lo pensamos, es un valor extraño, ya que es inferior al dado por Low).

***Figura 8.1: El formulario del ejemplo DynArr***



Tras esta breve introducción puedo mostrarle un sencillo ejemplo, llamado DynArr, que se muestra en la Figura 8.1. De hecho, es sencillo, ya que los vectores dinámicos no tienen nada de complejo. También lo utilizaré para mostrar algunos errores en que un programador puede incurrir. El programa declara dos vectores globales e inicializa el primero en el gestor de eventos OnCreate.

**var**

Array1, Array2: **array of** Integer;

**procedure** TForm1.FormCreate(Sender: TObject);

**begin**

*// adjudicar*

SetLength (Array1, 100);

**end**;

Esto iguala todos los valores a cero. Este código de inicialización hace posible comenzar a leer y escribir valores del vector sin más, sin temor a errores de memoria. (Asumiendo, por supuesto, que no intente acceder a nada que esté por encima del vector.) Para una inicialización aún mejor, el programa tiene un botón que escribe en cada celda del vector:

**procedure** TForm1.btnFillClick(Sender: TObject);

**var**

I: Integer;

**begin**

**for** I := Low (Array1) **to** High (Array1) **do**

Array1 [I] := I;

**end**;

El botón Grow permite modificar el tamaño del vector sin perder su contenido. Puede probar esto usando el botón *Get value* tras pulsar el botón Grow:

**procedure** TForm1.btnGrowClick(Sender: TObject);

**begin**

*// grow mantiene los valores existentes*

SetLength (Array1, 200);

**end**;

**procedure** TForm1.btnGetClick(Sender: TObject);

**begin**

*// extraer*

Caption := IntToStr (Array1 [99]);

**end**;

La única parte del código algo compleja corresponde al evento *OnClick* del botón Alias. El programa copia un vector al otro con el operador := , en realidad creando un *alias* (una nueva variable que apunta al mismo vector en memoria). Aquí, sin embargo, si modifica uno de los vectores, el otro también se ve afectado, ya que ambos se refieren a la misma área de memoria:

**procedure** TForm1.btnAliasClick(Sender: TObject);

**begin**

*// alias*

Array2 := Array1;

*// cambiar uno (cambian ambos)*

Array2 [99] := 1000;

*// mostrar el otro*

Caption := IntToStr (Array1 [99]);

El método *btnAliasClick* realiza una o dos operaciones. La primera es comprobar la igualdad de los vectores. Esto no comprueba los elementos que de hecho forman las estructuras, sino las áreas de memoria a que apuntan los vectores, comprobando si las variables son dos *alias* del mismo vector en memoria:

**procedure** TForm1.btnAliasClick(Sender: TObject);

**begin**

...

**if** Array1 = Array2 **then**

Beep;

*// truncar el primer vector*

Array1 := Copy (Array2, 0, 10);

**end**;

El segundo es una llamada a la función *Copy*, que no sólo mueve datos de un vector al otro, sino que también reemplaza el primer vector con uno nuevo, creado por la función. El efecto es que la variable *Array1* ahora apunta a un vector de 11 elementos, de tal forma que pulsar los botones *Get value* o *Set value* produce un error de memoria y muestra una excepción (a no ser que haya desactivado la comprobación de rango, en cuyo caso el error se produce igualmente, pero la excepción no se muestra). El código del botón Fill sigue funcionando perfectamente tras este cambio, ya que qué miembros del vector se han de modificar viene determinado por sus cotas actuales.

**Conclusión**

Por el momento, este capítulo cubre solamente variables dinámicas, que son un recurso importante para administrar la memoria, pero sólo son una parte del todo. Añadiré más material en el futuro.

La estructura de memoria descrita en este capítulo es típica de la programación en Windows, un tema que introduciré en el siguiente capítulo (sin extenderme al uso del VCL).

**Capítulo siguiente:** [**Programación bajo Windows**](http://www.marcocantu.com/epascal/Spanish/ch09win.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 9 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch09win.htm) Programación bajo Windows** |

Delphi proporciona una encapsulación completa de la API de Windows de bajo nivel, utilizando Object Pascal y la biblioteca de componentes visual (VCL), de forma que rara vez es necesario construir aplicaciones Windows utilizando Pascal tradicional y haciendo llamadas directas a funciones API de Windows. Con todo, los programadores que quieran utilizar algunas técnicas especiales no incluidas en la VCL aun tienen esa posibilidad el Delphi. Sólo es necesario este método en ciertos casos muy especiales, como el desarrollo de nuevas componentes Delphi basadas en llamadas a la API poco habituales, pero no mencionaré los detalles aquí. En su lugar, examinaremos algunos elementos de la interacción de Delphi con el sistema operativo y un par de técnicas de que se pueden aprovechar los programadores de Delphi.

**Handles en Windows**

Entre los tipos de datos introducidos por Windows en Delphi, los *handles* representan el grupo más importante. El nombre de este tipo de datos es *THandle*, y es definido en la unidad Windows como:

**type**

THandle = LongWord;

Los tipos de datos *handle* se implementan como números, pero no se usan como tales. En Windows, un *handle* es una referencia a una estructura de datos interna del sistema. Por ejemplo, cuando trabaja con una ventana (o un formulario Delphi), el sistema le da un *handle* a la ventana. El sistema le informa de que la ventana con que está trabajando lleva el número 142, por ejemplo. Desde ese momento, su aplicación puede pedirle al sistema que opere sobre la ventana número 142 — moviéndolo, cambiando su tamaño, reduciéndolo a un icono, etc. Muchas funciones API de Windows tienen, de hecho, un *handle* como primer parámetro. Esto no es sólo cierto para funciones que operan sobre ventanas; otras funciones API de Windows tienen como su primer parámetro un handle GDI, un handle de menú, un handle de instancias, un handle de mapa de bits u otro de los muchos tipos de handle.

En otras palabras, un handle es un código interno que se usa para referirse a un elemento específico manejado por el sistema, incluyendo una ventana, un mapa de bits, un icono, un bloque de memoria, un cursor, una fuente de caracteres, un menú, etc. En Delphi, rara vez necesitará acceder a los handles directamente, ya que se encuentran ocultos dentro de formularios, mapas de bits y otros objetos en Delphi. Resultan útiles cuando quiere hacer llamadas a funciones API de Windows no implementadas en Delphi.

Para completar esta descripción, he aquí un ejemplo sencillo que muestra el funcionamiento de los handles bajo Windows. El programa WHandle tiene un formulario sencillo, que contiene sólo un botón. En el código, respondo al evento OnCreate del formulario y al evento OnClick del botón, como se muestra en la siguiente definición textual del formulario principal:

**object** FormWHandle: TFormWHandle

Caption = *'Window Handle'*

OnCreate = FormCreate

**object** BtnCallAPI: TButton

Caption = *'Call API'*

OnClick = BtnCallAPIClick

**end**

**end**

Tan pronto como el formulario se crea, el programa busca el *handle* de la ventana que corresponde al formulario, accediendo a la propiedad Handle del formulario mismo. Hacemos una llamada a IntToStr para convertir el valor numérico del *handle* en una cadena de caracteres, y añadimos aquel al título del formulario, como se ve en la Figura 9.1:

**procedure** TFormWHandle.FormCreate(Sender: TObject);

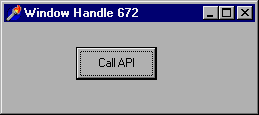
**begin**

Caption := Caption + *' '* + IntToStr (Handle);

**end**;

Ya que *FormCreate* es un método de la clase del formulario, puede acceder a otras propiedades y métodos de la misma clase, directamente. Por tanto, en este procedimiento podemos apuntar, sencillamente, al Caption del formulario y a su propiedad Handle, de forma directa.

***Figura 9.1: El ejemplo WHandle muestra el handle de la ventana del formulario. Cada vez que ejecute este programa, obtendrá un valor distinto.***



Si ejecuta este programa varias veces, en general obtendrá distintos valores para el handle. Este valor, de hecho, viene determinado por Windows, y es remitido a la aplicación. (Los handles nunca son establecidos por el programa, y no tienen valores predefinidos; son determinados por el sistema, que genera nuevos valores cada vez que usted ejecuta un programa.)

Cuando el usuario pulsa el botón, el programa simplemente hace una llamada a una función API, SetWindowText, que cambia el texto o título de la ventana transmitido como primer parámetro. Para ser más precisos, el primer parámetro de esta función API es el *handle* de la ventana que queremos modificar:

**procedure** TFormWHandle.BtnCallAPIClick(Sender: TObject);

**begin**

SetWindowText (Handle, *'Hi'*);

**end**;

Este código tiene el mismo efecto que el gestor de eventos anterior, que cambió el texto de la ventana dando un nuevo valor a la propiedad Caption (título) del formulario. En este caso, hacer una llamada a una función API no tiene sentido, ya que hay una técnica en Delphi más sencilla. Algunas funciones API, sin embargo, no tienen correspondencia en Delphi, como veremos en ejemplos más avanzados en el libro.

**Declaraciones externas**

Otros elementos importante para la programación bajo Windows viene representada por declaraciones externas. Utilizada en un principio para enlazar el código Pascal a funciones externas escritas en lenguaje ensamblador, la declaración externa se usa en la programación Windows para hacer una llamada a una función desde una DLL (una biblioteca de enlace dinámico). En Delphi, hay varias declaraciones tales en la unidad Windows:

*// declaración típica*

**function** LineTo (DC: HDC; X, Y: Integer): BOOL; **stdcall**;

*// declaration externa (en lugar de código)*

**function** LineTo; **external** *'gdi32.dll'* **name** *'LineTo'*;

Esta declaración significa que el código de la función LineTo está almacenado en la biblioteca dinámica GDI32.DLL (una de las más importantes en Windows), con el mismo nombre que estamos usando en nuestro código. Dentro de una declaración externa, de hecho, podemos especificar que nuestra función se refiera a la función de una DLL que originalmente tenía otro nombre.

Rara vez necesitará escribir declaraciones como la que acabamos de ilustrar, ya que ya están enumeradas en la unidad Windows y muchas otras unidades de sistema en Delphi. La única razón por la que pudiera necesitar escribir esta declaración externa es que necesitase hacer llamadas a funciones desde una DLL que se haya hecho a medida, o para hacer llamadas a funciones de Windows que no aparecen en la documentación.

Nota: en la versión de Delphi de 16 bits, la declaración externa utilizaba el nombre de la biblioteca sin la extensión, e iba seguida por la directiva de nombre (como en el código que aparece arriba) o por una directiva de índice alternativa, seguida por el número ordinal de la función dentro de la DLL. El cambio refleja un cambio en el sistema en relación a cómo se accede a las bibliotecas: a pesar de que Win32 aún permite acceder a las funciones de las DLL por su número, Microsoft ha declarado que esto no será posible en el futuro. Tenga en cuenta, también, que la unidad Windows reemplaza las unidades WinProcs y WinTypes de la versión de 16 bits de Delphi.

**Funciones *Callback* de Windows**

Hemos visto en el [**Capítulo 6**](http://www.marcocantu.com/epascal/Spanish/ch06proc.htm) que Object Pascal incluye tipos de procedimiento. Una práctica habitual de tipos de procedimiento es facilitar funciones *callback* a una función API de Windows.

Pero, ¿qué es una función *callback*? Se trata de una cierta función API que realiza una acción dada sobre una cierta cantidad de elementos internos del sistema, como todas las ventanas de cierto tipo. Tal función, también llamada función enumerada, requiere como parámetro la acción que ha de ejecutarse en cada elemento, que se transmite como una función o procedimiento compatible con un tipo de procedimiento dado. Windows usa las funciones callback en otras circunstancias, pero limitaremos nuestro estudio a este caso sencillo.

Ahora, considere la función API EnumWindows, que tiene el prototipo siguiente (copiado del archivo de ayuda de Win32):

BOOL EnumWindows(

WNDENUMPROC lpEnumFunc, *// dirección de la función callback*

LPARAM lParam *// valor definido por la aplicación*

);

Por supuesto, esta es la definición en lenguaje C. Podemos echar un vistazo al interior de la unidad Windows para obtener la definición correspondiente en lenguaje Pascal:

**function** EnumWindows (

lpEnumFunc: TFNWndEnumProc;

lParam: LPARAM): BOOL; **stdcall**;

Consultando el archivo de ayuda, encontramos que la función pasada como parámetro debería ser del siguiente tipo (de nuevo en C):

BOOL CALLBACK EnumWindowsProc (

HWND hwnd, *// handle of parent window*

LPARAM lParam *// application-defined value*

);

Esto se corresponde con la siguiente definición de tipo de procedimiento en Delphi:

**type**

EnumWindowsProc = **function** (Hwnd: THandle;

Param: Pointer): Boolean; **stdcall**;

El primer parámetro es el handle de la ventana principal a que le toca el turno, mientras que el segundo es el valor que hemos transmitido al efectuar la llamada a la función *EnumWindows*. De hecho, en Pascal, el tipo *TFNWndEnumProc* no está bien definido. Esto significa que necesitamos proporcionar una función con los parámetros adecuados y entonces utilizarla como puntero, tomando la dirección de la función, en vez de llamarla. Desgraciadamente, esto también significa que el compilador no proporcionará ayuda alguna en caso de que se produzca un error en el tipo de uno de los parámetros.

Windows exige a los programadores que sigan la convención de llamada *stdcall* cada vez que llamen a una función API de Windows o transmitan una función *callback* al sistema. Delphi, por defecto, usa una convención distinta, más eficiente, indicada por la palabra clave *register*.

He aquí la definición de una función compatible adecuada, que pasa el título de la ventana a una cadena, y lo añade a un ListBox de un formulario dado:

**function** GetTitle (Hwnd: THandle; Param: Pointer): Boolean; **stdcall**;

**var**

Text: string;

**begin**

SetLength (Text, 100);

GetWindowText (Hwnd, PChar (Text), 100);

FormCallBack.ListBox1.Items.Add (

IntToStr (Hwnd) + *': '* + Text);

Result := True;

**end**;

El formulario tiene un ListBox que cubre casi toda su área, junto con un pequeño panel en la parte superior, que incluye un botón. Cuando se pulsa dicho botón, se efectúa una llamada a la función API *EnumWindows*, y la función *GetTitle* se usa como parámetro :

**procedure** TFormCallback.BtnTitlesClick(Sender: TObject);

**var**

EWProc: EnumWindowsProc;

**begin**

ListBox1.Items.Clear;

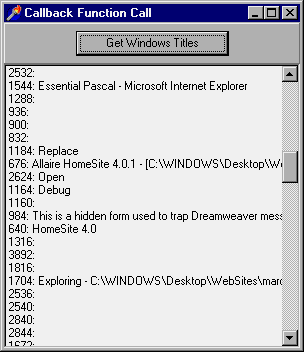
EWProc := GetTitle;

EnumWindows (@EWProc, 0);

**end**;

Podría igualmente haber efectuado la llamada a la función sin almacenar el valor en una variable de tipo de procedimiento previdamente, pero quería dejar claro qué ocurre en este ejemplo. El efecto de este programa es, de hecho, bastante interesante, como se ver en la Figura 9.2. El ejemplo Callback muestra una lista de las ventanas principales que se encuentran presentes actualmente en el sistema. La mayoría de ellas son ventanas ocultas que normalmente no ve (y muchas de ellas no tienen ni siquiera título).

***Figura 9.2: La salida del ejemplo Callback, enumerando las ventanas principales actuales (visibles u ocultas).***



**Un breve programa Windows**

Para completar la introducción a la programación bajo Windows en lenguaje Pascal, quisiera mostrarle una aplicación muy simple, pero completa, construida sin usar el VCL. El programa simplemente toma el parámetro de línea de comandos (almacenado por el sistema en la variable global *cmdLine*) y luego extrae información de él con las funciones de Pascal ParamCount y ParamStr Pascal. La primera de estas funciones devuelve el número de parámetros; la segunda devuelve el parámetro en una posición dada.

Aunque los usuarios rara vez usan parámetros de línea de comandos en un entorno de interfaz de usuario gráfica (GUI), los parámetros de línea de comandos de Windows son importantes para el sistema. Por ejemplo, una vez que se ha definido una asociación entre una extensión de archivo y una aplicación, se puede ejecutar un programa con sólo seleccionar un archivo asociado. En la práctica, cuando usted pincha dos veces (doble click) en un archivo, Windows inicia el programa asociado y transmite el archivo asociado como parámetro de línea de comandos.

Sigue el código fuente completo del proyecto (un archivo DPR, no PAS):

**program** Strparam;

**uses**

Windows;

**begin**

*// show the full string*

MessageBox (0, cmdLine,

*'StrParam Command Line'*, MB\_OK);

*// show the first parameter*

**if** ParamCount > 0 **then**

MessageBox (0, PChar (ParamStr (1)),

*'1st StrParam Parameter'*, MB\_OK)

**else**

MessageBox (0, PChar (*'No parameters'*),

*'1st StrParam Parameter'*, MB\_OK);

**end**.

El código de salida usa la función API *MessageBox*, simplemente para evitar introducir todo el VCL en el proyecto. Un programa Windows puro como el de arriba, tiene, de hecho, la ventaja de dejar una huella pequeña: el archivo ejecutable del programa comporta sólo 16 kB.

Para proporcionar un parámetro de línea de comandos a este programa, puede usar el comando de menú de Delphi Run > Parameters. Otra técnica es abrir el Explorador de Windows, localizar el directorio que contiene el archivo ejecutable del programa, y arrastrar el archivo que se quiere ejecutar al archivo ejecutable. El Explorador de Windows iniciará el programa usando el nombre del archivo arrastrado como parámetro de línea de comandos. La Figura 9.3 muestra tanto el Explorador como la salida correspondiente.

***Figura 9.3: Puede proporcionar un parámetro de línea de comandos al ejemplo StrParam arrastrando un archivo sobre el archivo ejecutable, en el Explorador de Windows.***



**Conclusión**

En este capítulo hemos visto una introducción a bajo nivel a la programación Windows, discutiendo los *handles* y un programa Windows muy simple. Para tareas de programación habituales en Windows, en general usará las posibilidades de desarrollo en un entorno visual que proporciona Delphi, basadas en el VCL. Pero esto sobrepasa el ámbito de este libro, que se circunscribe al lenguaje Pascal.

El siguiente capítulo cubre los variantes, un añadido muy extraño al sistema de tipos de Pascal, introducido para proporcionar total compatibilidad OLE (*Object Linking and Embedding*).

**Capítulo siguiente:** [**Variantes**](http://www.marcocantu.com/epascal/Spanish/ch10var.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 10 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch10var.htm) Variantes** |

Para proporcionar plena compatibilidad con OLE, la versión de 32 bits de Delphi incluye el tipo de datos variante (*Variant*). Aquí, quiero discutir este tipo de datos desde una perspectiva general. El tipo Variant tiene, de hecho, un efecto omnipresente en todo el lenguaje, y la biblioteca de componentes Delphi también lo usa de maneras no relacionadas con la programación OLE.

**Los variantes no tienen tipo**

En general, se pueden usar variantes para almacenar cualquier tipo de datos y realizar numerosas operaciones y conversiones de tipo. No pase por alto que esto contradice el estilo general del lenguaje Pascal y los buenos hábitos de programación. Los variantes son comprobados y computados durante la ejecución, no durante la compilación. El compilador no le advertirá de posibles errores en el código, que pueden ser descubiertos sólo probándolo un buen número de veces. Normalmente, se puede considerar que las porciones de código que usan variantes son *código interpretado* porque, como en tal código, muchas operaciones no pueden ser realizadas hasta la ejecución del programa. Esto afecta especialmente a la velocidad del mismo.

Ahora que les he advertido contra el uso del tipo Variant, es hora de echarle un vistazo a qué se puede hacer con él. Básicamente, una vez que ha declarado una variable variante como la siguiente ...

**var**

V: Variant;

... puede asignarle valores de distintos tipos :

V := 10;

V := *'Hello, World'*;

V := 45.55;

Una vez que tenga el valor variante, puede copiarlo a cualquier tipo de datos - ya sea compatible o no. Si asigna un valor a un tipo de datos incompatible, Delphi realiza una conversión, si puede. Si no puede, muestra un error de ejecución. De hecho, un variante almacena la información de tipo junto a los datos, permitiendo cierto número de operaciones de ejecución. Estas operaciones pueden resultarle fáciles de manejar, pero son tanto lentas como inseguras.

Considere el siguiente ejemplo (llamado VariTest), que es una extensión del código que aparece arriba. Coloqué tres cajas de edición en un formulario nuevo, añadí unos cuantos botones, y escribí el siguiente código para el evento OnClick del primer botón:

**procedure** TForm1.Button1Click(Sender: TObject);

**var**

V: Variant;

**begin**

V := 10;

Edit1.Text := V;

V := *'Hello, World'*;

Edit2.Text := V;

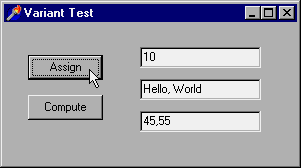
V := 45.55;

Edit3.Text := V;

**end**;

Es divertido, ¿no? Aparte de asignar un variante que contiene una cadena a la propiedad Text de un componente de edición, puede asignarle al Text un variante que contenga un entero o un número de coma flotante. Como puede ver en la Figura 10.1, funciona bien.

***Figura 10.1: La salida del ejemplo VariTest después de pulsar el botón Assign.***



Aún peor, se pueden usar variantes para calcular valores, como puede verse en el código correspondiente al segundo botón:

**procedure** TForm1.Button2Click(Sender: TObject);

**var**

V: Variant;

N: Integer;

**begin**

V := Edit1.Text;

N := Integer(V) \* 2;

V := N;

Edit1.Text := V;

**end**;

Es arriesgado escribir este tipo de código - y esto es decir poco. Si la primera casilla de edición contiene un número, todo va bien. Si no, se produce una *excepción*. En fin, puede escribirse código de este tipo, pero no se lo aconsejo, a no ser que tenga una imperiosa razón para hacerlo. Mejor, cíñase a los tipos de datos tradicionales de Pascal y a la posibilidad de comprobación de código durante la compilación. En Delphi en la VCL (Visual Component Library), normalmente los variantes son usados sólo para proporcionar compatibilidad OLE y para acceder a campos en una base de datos.

**Estudio en profundidad de los variantes**

Delphi incluye un tipo de registro variante, TVarData, que tiene la misma distribución de memoria que el tipo Variant. Puede usarse para averiguar cuál es el auténtico tipo de un variante dado. La estructura TVarData incluye al tipo del variante, indicado por *VType*, algunos campos de reserva, y el valor auténtico.

Los valores posibles del campo VType se corresponden con los tipos de datos que pueden usarse en la automatización OLE, que, a menudo, son tipos OLE o tipos variantes. He aquí una lista alfabética completa de los tipos de variante disponibles:

* varArray
* varBoolean
* varByRef
* varCurrency
* varDate
* varDispatch
* varDouble
* varEmpty
* varError
* varInteger
* varNull
* varOleStr
* varSingle
* varSmallint
* varString
* varTypeMask
* varUnknown
* varVariant

Encontrará descripciones de estos tipos en el apartado "Values in variants", en la ayuda de Delphi.

Hay también muchas funciones que permiten operar sobre variantes, que puede utilizar para hacer conversiones de tipos específicas, o para pedir información sobre el tipo de un variante (tome como ejemplo la función VarType mencionada). La mayoría de estas funciones de conversión de tipo y asignación son, en realidad, llamadas automáticamente cuando se escribe expresiones usando variantes. Otras rutinas de apoyo a variantes operan, en verdad, en vectores variantes (véase el apartado "Variant support routines" en la ayuda de Delphi).

**Los variantes son leeeentos**

El código en que se hace uso del tipo Variant es lento, no sólo al convertir tipos de datos, sino también cuando se añaden dos valores de variante que contienen un entero, cada uno. ¡ Son casi tan lentos como el código interpretado de VisualBasic ! Para comparar la velocidad de un algoritmo basado en variantes con uno que, teniendo el mismo código, está basado en enteros, examine el ejemplo VSpeed.

Este programa ejecuta un bucle, contando su velocidad y mostrando el estado en una barra de progreso. Aquí está el primero de los dos bucles, muy similares, basado en enteros y variantes respectivamente :

**procedure** TForm1.Button1Click(Sender: TObject);

**var**

time1, time2: TDateTime;

n1, n2: Variant;

**begin**

time1 := Now;

n1 := 0;

n2 := 0;

ProgressBar1.Position := 0;

**while** n1 < 5000000 **do**

**begin**

n2 := n2 + n1;

Inc (n1);

**if** (n1 **mod** 50000) = 0 **then**

**begin**

ProgressBar1.Position := n1 **div** 50000;

Application.ProcessMessages;

**end**;

**end**;

*// we must use the result*

Total := n2;

time2 := Now;

Label1.Caption := FormatDateTime (

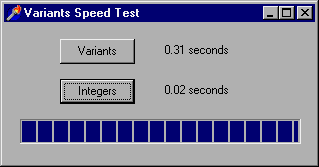
*'n:ss'*, Time2-Time1) + *' seconds'*;

**end**;

Merece la pena examinar el código de temporización, porque es algo que se puede adaptar fácilmente a cualquier prueba de rendimiento. Como se ve, el programa usa la función Now para averiguar la hora actual, y la función FormatDateTime para dar el tiempo transcurrido, pidiendo sólo los minutos ("n") y los segundos ("ss") en la cadena de formato. Puede usar en vez de esto la función *GetTickCount* del API de Windows, que devuelve una medida muy precisa de los milisegundos transcurridos desde que se inició el sistema operativo.

En este ejemplo la diferencia de velocidad es, de hecho, tan grande, que lo notará aun sin calcularlo sin precisión. De cualquier modo, puede ver los resultados de mi propia computadora en la Figura 10.2. En la práctica, los valores dependerán de la máquina en que ejecute el programa, pero la proporción no cambiará mucho.

***Figura 10.2: Las velocidades respectivas de un mismo algoritmo, en sus versiones en enteros y variantes (el tiempo requerido, en realidad, depende de cada computadora), ejemplificado por VSpeed.***



**Conclusión**

Los variantes son tan diferentes de los tipos de datos tradicionales de Pascal, que he decidido hablar de ellos en este breve capítulo aparte. Aunque su papel se restringe a la programación OLE, pueden ser útiles para escribir programas de forma rápida y chapucera, sin tener que plantearse nada sobre tipos de datos. Como hemos visto, esto afecta de lleno al rendimiento.

Ahora que hemos cubierto la mayor parte de las características del lenguaje Delphi, pasaremos a comentar la estructura general de un programa y la posibilidad de modularización que nos ofrecen las unidades.

**Capítulo siguiente:** [**Programas y unidades**](http://www.marcocantu.com/epascal/Spanish/ch11unit.htm)

|  |  |
| --- | --- |
| Logo | * [**www.marcocantu.com**](http://www.marcocantu.com/) * [**Marco's Delphi Books**](http://www.marcocantu.com/books) * [**Essential Pascal - Web Site**](http://www.marcocantu.com/epascal) * [**Essential Pascal - Local Index**](http://www.marcocantu.com/epascal/Spanish/default.htm) |
| **Marco Cantù Pascal Esencial** | **Capítulo 11 [Actualizar](http://www.marcocantu.com/EPascal/Spanish/ch11unit.htm) Programas y unidades** |

Las aplicaciones escritas en Delphi hacen un uso intensivo de unidades, o módulos de programa. De hecho, las unidades fueron la base de la modularidad del lenguaje antes de que se introdujeran las clases. En una aplicación Delphi, cada formulario viene respaldado por su unidad correspondiente. Cuando se añade un nuevo formulario a un proyecto (con el botón de herramienta correspondiente o el comando de menú Archivo > Nuevo formulario...), en realidad, Delphi añade una nueva unidad, que define la clase para el nuevo formulario.

**Unidades**

Aunque cada formulario se define mediante una unidad, el recíproco no es cierto. Las unidades no tienen por qué definir formularios. Pueden, simplemente, definir y hacer disponible una colección de rutinas. Seleccionando el comando Archivo > Nuevo y luego el icono Unit en la página New del Object Repository, se añade una unidad en blanco al proyecto actual. Esta unidad en blanco contiene el siguiente código, delimitando las secciones en que se divide una unidad :

**unit** Unit1;

**interface**

**implementation**

**end**.

El concepto de unidad es sencillo. Una unidad tiene un único nombre, que corresponde a su nombre de archivo, una sección de interfaz que declara qué es visible a otras unidades, y una sección de implementación con el código auténtico y otras declaraciones ocultas. Finalmente, la unidad puede tener una sección de inicialización opcional con cierta porción de código de inicio, que se ejecuta cuando el programa se carga en la memoria; también puede tener una sección de finalización, opcional, que se ejecutaría al terminar el programa.

La estructura general de una unidad, con todas sus secciones posibles, es la siguiente:

**unit** unitName;

**interface**

*// otras unidades a que necesitamos referirnos*

**uses**

A, B, C;

*// definición del tipo exportado*

**type**

newType = TypeDefinition;

*// constantes exportadas*

**const**

Zero = 0;

*// variables globales*

**var**

Total: Integer;

*// lista de funciones y procedimientos exportados*

**procedure** MyProc;

**implementation**

**uses**

D, E;

*// variable global oculta*

**var**

PartialTotal: Integer;

*// todas las funciones exportadas deben ser codificadas*

**procedure** MyProc;

**begin**

*// ... código del procedimiento MyProc*

**end**;

**initialization**

*// parte de inicialización, opcional*

**finalization**

*// código de limpieza, opcional*

**end**.

La cláusula *uses*, al principio de la sección de interfaz, indica a qué otras unidades tenemos que acceder en la porción de interfaz de la unidad. Esto incluye las unidades que definen los tipos de datos a que nos referimos en la definición de otros tipos de datos, como las componentes dentro de un formulario que estemos definiendo.

La segunda cláusula *uses*, al principio de la sección de implementación, indica otras unidades, a que necesitamos acceder sólo en el código de implementación. Cuando necesite referirse a otras unidades del código de las rutinas y métodos, deberá añadir elementos en esta segunda cláusula *uses*, en vez de hacerlo en la primera. Todas las unidades a que se refiera deben estar presentes en el directorio del proyecto o un directorio de la ruta de búsqueda (puede establecer la ruta de búsqueda de un proyecto en la página Directories/Conditionals del cuadro de diálogo Options del proyecto).

Los programadores en C++ deberán tener en cuenta que la instrucción *uses* no se corresponde con una directiva *include*. El efecto de una instrucción *uses* es importar sólo la porción de interfaz precompilada de las unidades listadas. La porción de implementación de la unidad se considera sólo cuando tal unidad se compila. Las unidades a que se refiera pueden estar tanto en formato de código fuente (PAS) o formato compilado (DCU), pero la compilación tiene que haber tenido lugar en la misma versión de Delphi.

El interfaz de una unidad puede declarar bastantes elementos diferentes, incluyendo procedimientos, funciones, variables globales y tipos de datos. En las aplicaciones Delphi, los tipos de datos son los que, probablemente, se utilicen más a menudo. Delphi automáticamente sitúa un tipode datos de clase nueva en una unidad cada vez que usted cree un formulario. De cualquier modo, contener definiciones de formulario no es la única finalidad de las unidades en Delphi.

Puede seguir utilizando unidades convencionales, con funciones y procedimientos, y puede hacer uso de unidades con clases que no se refieran a formularios u otros elementos visuales.

**Unidades y accesibilidad**

En Pascal, las unidades son la clave para el encapsulamiento y la accesibilidad, y son, probablemente, aún más importantes que las palabras claves privadas y públicas de una clase. (De hecho, como veremos en el capítulo 12 <en preparación>, el efecto de la palabra clave privada está relacionada con la visibilidad de la unidad que contiene la clase.) La visibilidad de un identificador (p.ej. una variable, un procedimiento, una función o un tipo de datos) es la porción del código en que el identificador es accesible. La regla básica es que un identificador tiene sentido sólo donde es visible — esto es, sólo dentro del bloque en que fue declarada. No se puede usar un identificador fuera de donde es visible. He aquí algunos ejemplos.

* Variables locales: Si se declara una variable dentro del bloque que define una rutina o un método, no se puede usar esta variable fuera de ese procedimiento. El campo de acción del identificador abarca todo el procedimiento, incluyendo rutinas anidadas (a no ser que un identificador definido en la rutina anidada anule una definición externa con el mismo nombre). La memoria para esta variable se almacena en la pila cuando el programa se ejecuta la rutina que lo define. Tan pronto como finalice la ejecución de la rutina, la memoria en la pila se libera automáticamente.
* Variables globales ocultas: Si se declara un identificador en la porción de implementación de una unidad, no se puede usar aquella fuera de la unidad, pero se puede usar en cualquier bloque y procedimiento definido dentro de la unidad. La memoria para esta variable se adjudica tan pronto como comienza el programa, y se mantiene mientras no finalice. Se puede utilizar la sección de inicialización de la unidad para dar un valor inicial específico.
* Variables globales: si declara un identificador en la porción de la interfaz de la unidad, su campo de acción se extiende a cualquier otra unidad que utilice la unidad en que se declara. Tal variable ocupa memoria y tiene la misma vigencia que una del tipo anterior; la única diferencia es su visibilidad.

Cualquier declaración en la porción de interfaz de una unidad es accesible desde cualquier parte del programa que incluya la unidad en la cláusula *uses*. Las variables de clases de formulario se declaran del mismo modo, de tal forma que se podrá referir a un formulario (y a sus campos públicos, métodos, propiedades y componentes) desde el código de cualquier otro formulario. Por supuesto, es un hábito de programación poco recomendable declarar todo como global. Aparte de los problemas de ocupación de memoria obvios, usar variables globales hace que el mantenimiento y la actualización del programa sean más difíciles. En resumen, será mejor que use el menor número posible de variables globales.

**Unidades como *Namespaces***

La instrucción *uses* es la técnica habitual para acceder el marco de vigencia de otra unidad. Con ella podrá acceder a las definiciones de la unidad. Pero podría ocurrir que dos unidades a que se refiere declaren el mismo identificador; esto es, podría encontrarse con dos clases o dos rutinas con el mismo nombre.

En este caso puede, simplemente, usar el nombre de unidad como prefijo del nombre del tipo o rutina definida en aquella. Por ejemplo, puede referirse al procedimiento *ComputeTotal* definido en la unidad *Totals* dada como *Totals.ComputeTotal*. Esto no será necesario muy a menudo, a no ser que incurra en el error de dar el mismo nombre a dos cosad distintas en un programa.

De cualquier manera, si echa un vistazo a la biblioteca VCL y los archivos de Windows, encontrará que algunas funciones Delphi tienen el mismo nombre (pero, en general, distintos parámetros) que ciertas funciones API de Windows disponibles en el mismo Delphi. Un ejemplo es el sencillo procedimiento *Beep*.

Cree un nuevo programa en Delphi, añada un botón y escriba el siguiente código:

**procedure** TForm1.Button1Click(Sender: TObject);

**begin**

Beep;

**end**;

Entonces, tan pronto como pulse el botón oirá un breve sonido. Ahora, pase a la instrucción uses de la unidad y modifique ligeramente el aspecto actual del código ...

**uses**

Windows, Messages, SysUtils, Classes, ...

... situando la unidad *SysUtils* antes de la unidad *Windows* :

**uses**

SysUtils, Windows, Messages, Classes, ...

Si, ahora, intenta recompilar este código, dará con un error de compilación: "Not enough actual parameters." (No hay suficientes parámentros actuales). El problema consiste en que la unidad Window define otra función Beep con dos parámetros. Dicho de forma más general, lo que ocurre en las definiciones de las primeras unidades se podría ocultar con definiciones correspondientes en unidades posteriores. La solución segura es, de hecho, bastante simple :

**procedure** TForm1.Button1Click(Sender: TObject);

**begin**

SysUtils.Beep;

**end**;

Este código se compilará independientemente del orden de las unidades en las instrucciones *uses*. Aparte de estos, son pocos los conflictos de nombre que aparecen en Delphi, sencillamente porque el código de Delphi está, en general, contenido en métodos de clases. Tener dos métodos con el mismo nombre en clases distintas no crea ningún problema. Los problemas aparecen sólo con las rutinas globales.

**Unidades y programas**

Una aplicación Delphi consiste en dos tipos de archivo de código fuente: una o más unidades y un archivo de programa. Las unidades se pueden considerar archivos secundarios, a que se puede referir con la parte principal de la aplicación, el programa. En teoría, esto es cierto. En la práctica, el archivo de programa es, habitualmente, un archivo generado de forma automática, y tiene un papel limitado. Sólo tiene que iniciar el programa, ejecutando el formulario principal. El código del archivo de programa, o *Delphi project file* (DPR) puede ser editado tanto manualmente como usando el Project Manager y alguna de las opciones de proyecto relacionadas con el objeto de la aplicación y los formularios.

La estructura del archivode programa es normalmente mucho más sencilla que la estructura de las unidades. He aquí el código fuente de un archivo de programa que nos sirve de ejemplo :

**program** Project1;

**uses**

Forms,

Unit1 **in** *‘Unit1.PAS’ {Form1DateForm}*;

**begin**

Application.Initialize;

Application.CreateForm (TForm1, Form1);

Application.Run;

**end**.

Como se ve, sólo aparece una sección *uses* y el código principal de la aplicación, encerrada por las palabras clave *begin* y *end*. La instrucción uses del programa es especialmente importante, porque se usa para administrar la compilación y el enlazamiento de la aplicación.

**Conclusión**

Al menos por el momento, este capítulo sobre la estructura de una aplicación Pascal escrita en Delphi o en una de las últimas versiones de Turbo Pascal, es el último del libro. Puede usted enviarme correo electrónico con sus comentarios y peticiones.

Si después de esta introducción al lenguaje Pascal quiere sumergirse en los elementos orientados a objeto del Object Pascal de Delphi, puede consultar mi libro ***Mastering Delphi 5*** (Sybex, 1999) [N. del T.: La traducción al español de *Mastering Delphi 4* está disponible bajo el título *Delphi 4*]. Para más información sobre estos y otros libros avanzados escritos por mí (y otros autores) puede consultar mi sitio en la Red, [**www.marcocantu.com**](http://www.marcocantu.com/). El mismo sitio mantiene versiones actualizadas de este libro y de los ejemplos de código que aparecen en él.

[**Volver a la página principal**](http://www.marcocantu.com/epascal/Spanish/default.htm)

© Copyright Marco Cantù, Wintech Italia Srl 1995-99  
© Copyright de la traducción, Rafael Barranco-Droege, 2000  
[El traductor agradecerá cualquier comentario o crítica dirigidos a: b-d *seguido de* @bigfoot.com]