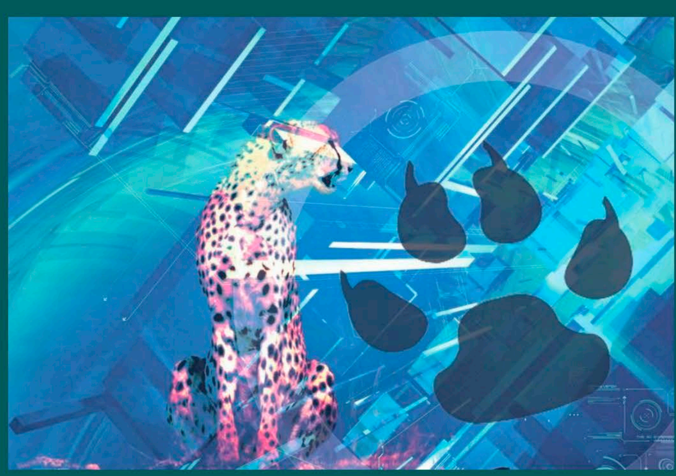
# Aprenda a programar con Lazarus



David Arboledas Brihuega

Desde **www.ra-ma.es** podrá descargar material adicional.

# Aprenda a programar con
# Lazarus

*David Arboledas Brihuega*

Profesor de informática

La ley prohíbe
Copiar o Imprimir este libro

Aprenda a programar con Lazarus
© David Arboledas Brihuega

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-9964-511-7
Todos los derechos reservados © RA-MA, S.A. Editorial y Publicaciones, Madrid, España.

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación y diseño portada: Antonio García Tomé

ISBN: 978-84-9964-488-2

E-Book desarrollado en España en Octubre de 2015

# Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

*"Descarga del material adicional del libro"*

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

Antonia Domínguez González

*In memoriam*

# ÍNDICE

**INTRODUCCIÓN** ........ **13**

**CAPÍTULO 1. EL ARTE DE PROGRAMAR** ........ **17**

1.1 LENGUAJES DE PROGRAMACIÓN ........ 19
1.2 PASCAL, UN LENGUAJE DE PROPÓSITO GENERAL ........ 20
1.3 LAZARUS ........ 21

**CAPÍTULO 2. LAZARUS Y PASCAL** ........ **25**

2.1 EL IDE DE LAZARUS ........ 25
2.2 MODO CONSOLA FRENTE A INTERFAZ GRÁFICA ........ 28
2.3 EL PRIMER PROGRAMA ........ 30
2.4 ESTRUCTURA DE UN PROGRAMA PASCAL ........ 32
2.5 COMENTARIOS ........ 34
2.6 LOS NOMBRES EN PASCAL ........ 35
2.7 DIRECTIVAS DEL COMPILADOR ........ 36

**CAPÍTULO 3. TIPOS, VARIABLES Y CONSTANTES** ........ **37**

3.1 TIPOS ........ 38
3.1.1 Enteros ........ 38
3.1.2 Ordinales ........ 39
3.1.3 Reales ........ 40
3.1.4 Lógicos ........ 40
3.1.5 Carácter ........ 41
3.1.6 Cadena ........ 42
3.1.7 Tipo enumerado ........ 43
3.2 CAMBIO Y FORZADO DE TIPOS ........ 44

3.3 VARIABLES ........ 47
3.3.1 Sentencias básicas de asignación ........ 49
3.4 CONSTANTES ........ 50
3.5 PUNTEROS ........ 50
3.6 UN PROGRAMA DE EJEMPLO ........ 53

**CAPÍTULO 4. TIPOS ESTRUCTURADOS DE DATOS ........ 57**
4.1 ARREGLOS ........ 57
4.1.1 Arreglos estáticos ........ 58
4.1.2 Arreglos dinámicos ........ 60
4.2 CADENAS ........ 61
4.3 REGISTROS ........ 63
4.3.1 La sentencia *with…do* ........ 66
4.4 CONJUNTOS ........ 67
4.5 FICHEROS ........ 69

**CAPÍTULO 5. OPERADORES Y EXPRESIONES ........ 73**
5.1 OPERADORES ARITMÉTICOS ........ 74
5.2 OPERADORES LÓGICOS ........ 75
5.3 OPERADORES RELACIONALES ........ 78
5.4 PRECEDENCIA DE OPERADORES ........ 78
5.5 UN PROGRAMA DE EJEMPLO ........ 80

**CAPÍTULO 6. SENTENCIAS DE CONTROL ........ 81**
6.1 ESTRUCTURAS DE SELECCIÓN ........ 82
6.1.1 Sentencia *if* ........ 82
6.1.2 Estructura *case of* ........ 84
6.2 ESTRUCTURAS REPETITIVAS ........ 86
6.2.1 El bucle *for* ........ 86
6.2.2 La sentencia *for in* ........ 88
6.2.3 El bucle *while* ........ 90
6.2.4 La estructura *repeat* ........ 92
6.3 SENTENCIAS DE CONTROL EN LOS BUCLES ........ 94
6.3.1 *Break* ........ 94
6.3.2 *Continue* ........ 96
6.4 LAS EXCEPCIONES ........ 97
6.4.1 Flujo de programa ........ 98
6.4.2 Clases de excepciones ........ 100

**CAPÍTULO 7. FUNCIONES Y PROCEDIMIENTOS ........ 103**
7.1 FUNCIONES ........ 104
7.1.1 Declaración y definición de una función ........ 104
7.1.2 Parámetros de referencia ........ 106

7.1.3 Argumentos por defecto ... 107
7.1.4 Sobrecarga de funciones ... 108
7.2 PROCEDIMIENTOS ... 108
7.2.1 Definición de un procedimiento ... 108
7.2.2 Declaración ... 109
7.2.3 Argumentos ... 109
7.2.4 La rutina *Exit* ... 111
7.3 RUTINAS RECURSIVAS ... 112

**CAPÍTULO 8. PROGRAMACIÓN ORIENTADA A OBJETOS ... 115**
8.1 CLASES Y OBJETOS ... 115
8.2 GESTIÓN DE MEMORIA ... 119
8.3 ENCAPSULADO ... 120
8.3.1 Niveles de acceso a los campos ... 121
8.3.2 Encapsulado con propiedades ... 124
8.4 DEFINICIÓN DE OBJETOS CON UNIDADES ... 127
8.4.1 Uso de las unidades ... 130

**CAPÍTULO 9. HERENCIA Y POLIMORFISMO ... 131**
9.1 LA HERENCIA ... 131
9.1.1 La cláusula *inherited* ... 133
9.1.2 La cláusula *Self* ... 136
9.2 EL POLIMORFISMO ... 138
9.2.1 Métodos virtuales, dinámicos y abstractos ... 142
9.2.2 Polimorfismo multiplataforma ... 143

**CAPÍTULO 10. LAZARUS Y LA PROGRAMACIÓN VISUAL ... 145**
10.1 LA ESTRUCTURA DE UN PROGRAMA GRÁFICO ... 146
10.2 EL INSPECTOR DE PROYECTO ... 150
10.3 EL EDITOR DE CÓDIGO FUENTE ... 151
10.3.1 Navegación rápida ... 152
10.3.2 Compleción automática de código ... 153
10.4 EL PRIMER PROGRAMA GRÁFICO ... 155
10.4.1 Modificar el título e icono del programa ... 157
10.5 EL INSPECTOR DE OBJETOS ... 158
10.5.1 La pestaña *Restringido* ... 159
10.6 LA PALETA DE COMPONENTES ... 160

**CAPÍTULO 11. CONTROLES VISUALES ... 163**
11.1 LA CLASE *TCONTROL* ... 164
11.2 EL NOMBRE DE LOS COMPONENTES ... 165
11.3 PROPIEDADES DE ACTIVACIÓN Y VISIBILIDAD ... 166
11.4 EVENTOS ... 167

11.5 LOS COMPONENTES DE VISUALIZACIÓN ........ 168
11.5.1 *TLabel* ........ 168
11.5.2 *TStaticText* ........ 170
11.5.3 *TBevel* ........ 171
11.5.4 *TStatusBar* ........ 171
11.6 LOS COMPONENTES DE ENTRADA DE TEXTO ........ 174
11.6.1 *TEdit* y *TLabeledEdit* ........ 174
11.6.2 *TMaskEdit* ........ 176
11.6.3 Edición de números enteros y reales ........ 178
11.7 SELECCIÓN DE OPCIONES ........ 183
11.7.1 *TCheckBox* y *TRadioButton* ........ 183
11.7.2 Listas ........ 187
11.7.3 Rangos ........ 191

**CAPÍTULO 12. LA INTERFAZ DE USUARIO ........ 195**
12.1 FORMULARIOS DE VARIAS PÁGINAS ........ 195
12.1.1 *TPageControl* y *TTabSheet* ........ 196
12.1.2 La interfaz de un asistente ........ 201
12.2 EL CONTROL *TTOOLBAR* ........ 204
12.2.1 El ejemplo *ToolBarDemo* ........ 205
12.3 EL COMPONENTE *TACTIONLIST* ........ 207
12.3.1 Acciones predefinidas en Lazarus ........ 209

**CAPÍTULO 13. CLASES NO VISUALES ........ 217**
13.1 LA CLASE *TPERSISTENT* ........ 217
13.2 COLECCIONES ........ 218
13.3 LISTAS Y LISTAS DE CADENA ........ 223
13.3.1 Pares nombre-valor ........ 224
13.3.2 Trabajar con listas de cadenas ........ 225
13.4 FLUJOS DE DATOS ........ 228
13.4.1 La clase *TStream* ........ 228
13.4.2 Clases de flujos ........ 229
13.4.3 Flujos de datos de archivo ........ 230
13.4.4 Compresión de flujos de datos con *ZStream* ........ 236
13.4.5 Criptografía de un flujo de datos con *Blowfish* ........ *240*

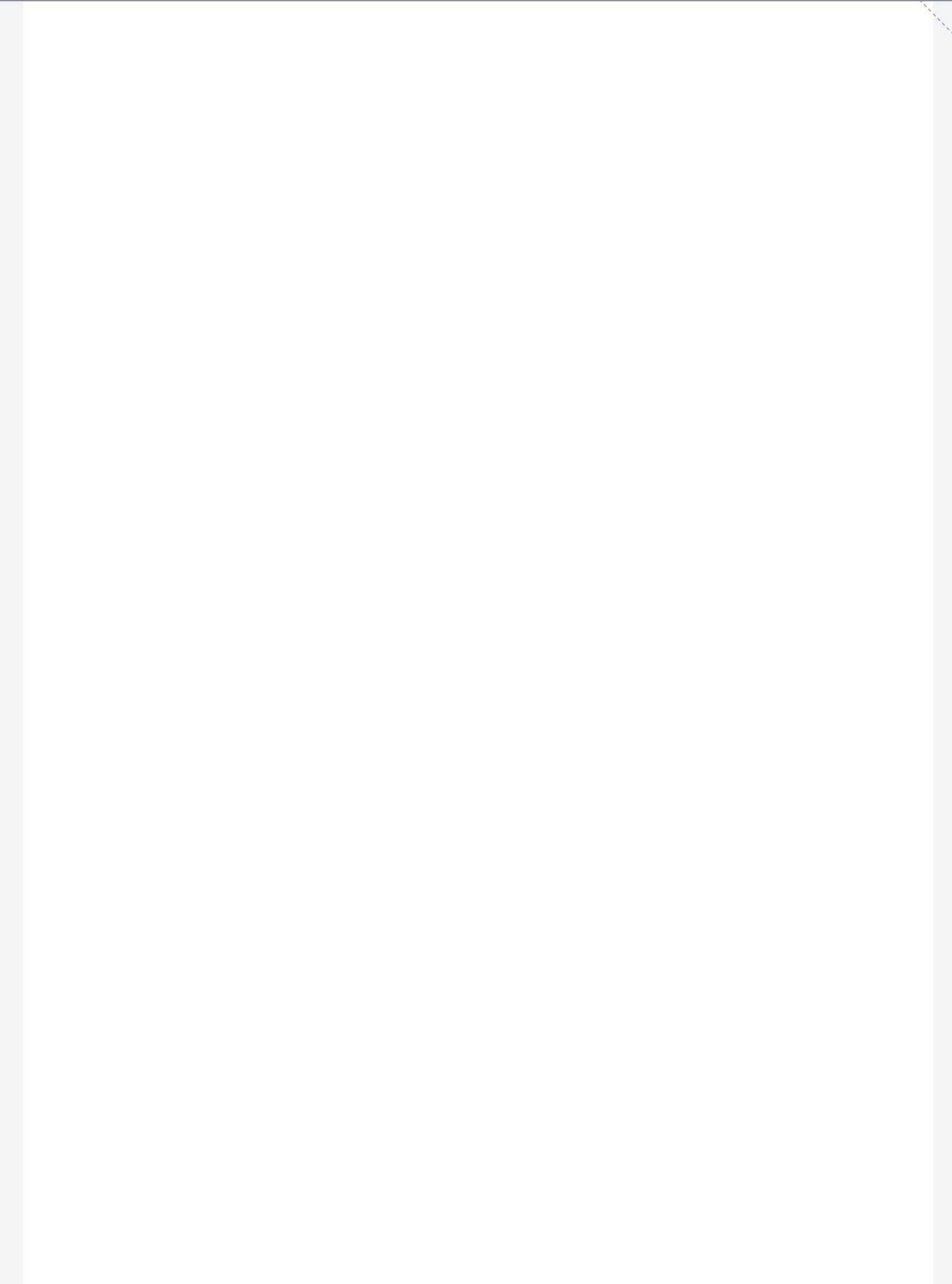
**CAPÍTULO 14. MANIPULACIÓN DE FICHEROS ........ 243**
14.1 REGISTROS Y CAMPOS DE UN ARCHIVO ........ 243
14.2 OPERACIONES SOBRE FICHEROS ........ 245
14.3 TIPOS DE FICHEROS ........ 246
14.4 EL TIPO FILE ........ 246
14.4.1 Asignación de un fichero ........ 247

14.5 ARCHIVOS DE TEXTO ..... 248
14.5.1 Apertura de archivos de texto ..... 248
14.5.2 Lectura/Escritura de datos en archivos de texto ..... 249
14.5.3 Cierre de los ficheros de texto ..... 250
14.6 ARCHIVOS BINARIOS ..... 255
14.6.1 Apertura de archivos binarios ..... 256
14.6.2 Lectura/Escritura de datos en archivos binarios ..... 256
14.6.3 Cierre de los ficheros ..... 257
14.6.4 Otras operaciones con archivos binarios ..... 259
14.7 ARCHIVOS SIN TIPO ..... 262
14.7.1 Procedimientos de apertura ..... 262
14.7.2 Procedimientos de entrada y salida de datos ..... 263
14.8 LOS FICHEROS EN LA POO ..... 266
14.8.1 Archivos de texto ..... 266
14.8.2 Archivos binarios ..... 269
14.9 UN ÚLTIMO APUNTE: NOMBRES DE FICHEROS ..... 274

**CAPÍTULO 15. LOS LÍMITES DE LA PROGRAMACIÓN ..... 277**
15.1 RECURSIVIDAD Y NÚMEROS FACTORIALES ..... 277
15.2 PERMUTACIONES ..... 282
15.3 TIEMPO Y COMPUTACIÓN ..... 289
15.4 DE EVENTOS A HILOS ..... 290
15.4.1 Multihilos en Lazarus ..... 291

**CAPÍTULO 16. TÉCNICAS DE DEPURACIÓN ..... 295**
16.1 PREVENCIÓN DE ERRORES ..... 296
16.2 UNIDADES DE PRUEBA ..... 296
16.2.1 Un ejemplo de prueba ..... 297
16.3 MENSAJES DEL COMPILADOR ..... 301
16.4 LAS ASERCIONES ..... 303
16.5 OBSERVADOR DE CÓDIGO ..... 305
16.6 REFACTORIZACIÓN ..... 308
16.7 SEGUIMIENTO DE LAS VARIABLES ..... 310
16.7.1 La directiva *{$DEFINE DEBUG}* ..... 310
16.7.2 Funciones de depuración ..... 313
16.7.3 Interrupciones ..... 315
16.7.4 El servidor de depuración ..... 316
16.8 LA UNIDAD *HEAPTRC* ..... 317
16.9 EL DEPURADOR GDB ..... 319

**ÍNDICE ALFABÉTICO ..... 325**



# INTRODUCCIÓN

Los ordenadores pueden hacer casi cualquier cosa que la mente humana sea capaz de imaginar; sin embargo, esto solo es posible con las instrucciones adecuadas. Pero ¿qué idioma habla un ordenador? La respuesta es fácil: el lenguaje de máquina; el cómo, más complicado, pues trasladar el lenguaje natural humano a código máquina no es sencillo. De esta última labor se encargan los lenguajes de programación.

Un lenguaje de programación es un idioma formal e inventado, diseñado para expresar procesos que pueden ser llevados a cabo por las computadoras. Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

Programar una máquina a bajo nivel es un proceso muy complicado y, además, específico de los circuitos, por lo que la tendencia ha sido emplear lenguajes de alto nivel, que se caracterizan por encontrarse más cercanos al lenguaje natural que al lenguaje de máquina. Se trata de idiomas independientes de la arquitectura del ordenador, por lo que, en principio, un programa escrito en un lenguaje de alto nivel se puede migrar a otra máquina sin grandes complicaciones. Permiten al programador olvidarse por completo del funcionamiento interno de la máquina para la que está implementando el programa.

Pascal es un lenguaje de programación de propósito general y de alto nivel, denominado así en honor al científico Blaise Pascal. Fue desarrollado por el profesor suizo Niklaus Wirth entre los años 1968 y 1969, y publicado en 1970. Su objetivo fue crear un lenguaje que facilitara el aprendizaje de programación a sus alumnos. Con

el tiempo, su utilización se extendió más allá del ámbito académico para convertirse en una herramienta para la creación de aplicaciones de todo tipo.

Pascal emplea un reducido vocabulario de palabras en inglés reservadas para su uso, lo que lo hace realmente atractivo para empezar a programar; además, enseña buenos hábitos de programación a un novel.

En el año 1989, con el compilador de Turbo Pascal 5.5, la compañía estadounidense Borland introdujo la programación orientada a objetos en Pascal. A mediados de los 90, con el resurgimiento de los sistemas tipo Linux, Pascal quedó relegado a un segundo plano en beneficio del lenguaje C. En ese momento solo Delphi, un Pascal orientado a objetos para Windows, consiguió mantenerse. Casi todo el mundo pretendía migrar a C++ o a Java, pero su enorme complejidad para los noveles hizo que al final Delphi siguiera existiendo; hoy sigue en el mercado con nuevas versiones por parte de la compañía Embarcadero.

Sin embargo, mientras Turbo Pascal se convertía en Delphi, comenzaba a gestarse el proyecto Free Pascal, un compilador de Pascal orientado a objetos pero con tres características esenciales que, a la larga, fueron enormes ventajas: se trataba de un proyecto de código abierto, completamente gratuito y multiplataforma, lo que hoy permite programar aplicaciones orientadas a objetos para más de una quincena de arquitecturas diferentes sin costes de licencias. Estas tres indudables ventajas han ido relegando en muchos campos a Delphi en beneficio de Lazarus.

El entorno de desarrollo integrado (IDE) para Lazarus se basa en una extensión orientada a objetos del lenguaje de programación Pascal. Si Delphi fue el lenguaje elegido por Borland para el entorno de desarrollo que hoy conocemos como Embarcadero Delphi, Free Pascal es el motor que corre bajo Lazarus. De hecho, el binomio Lazarus/Free Pascal es el que permite desarrollar programas tipo Delphi en un gran número de plataformas y a coste nulo. Como proyecto de código abierto, en contraposición al software propietario de Delphi, todo su código fuente está puesto a disposición de cualquier programador. Aprovechando esa libertad, un grupo de programadores y otros colaboradores corrigen y mejoran el proyecto sin cobrar por ello de ninguna empresa. Además, y este es un punto muy interesante, es un software completamente gratuito. No hay que pagar licencias ni siquiera para desarrollar aplicaciones profesionales.

Lazarus integra, en una única aplicación, todas las herramientas y funciones que necesitará para programar y obtener un ejecutable partiendo solo de la idea. Es, como hemos dicho, una herramienta RAD que poco tiene que envidiar al buque insignia de Embarcadero.

Como software libre, existen muchas versiones. A veces surge una nueva compilación cada día. Sin embargo, tanto Free Pascal como el IDE de Lazarus publican y numeran nuevas versiones solo cuando han sido largamente probadas, incluso en aplicaciones comerciales, y cuando los errores hallados hasta la fecha han sido corregidos.

Para resolver cualquier duda que pueda surgirle, tiene a su disposición a la comunidad que colabora en el proyecto desde la wiki *http://wiki.freepascal.org/*. Así mismo, también puede participar, resolviendo o planteando problemas, a través de las listas de correo y foros. Puede acceder al foro principal de Lazarus desde la dirección *http://forum.lazarus.freepascal.org/index.php?action=forum*. Si lo necesita, podrá también descargar y consultar toda la documentación del proyecto: *http://sourceforge.net/projects/lazarus/files/Lazarus%20Documentation*. Existen varios tutoriales gratuitos en la Red que ofrecen alternativas al material que hemos recogido en este libro. Uno muy interesante está en la dirección *http://wiki.lazarus.freepascal.org/Lazarus_Tutorial*.

El libro se ha escrito con la idea de que sirva de guía completamente práctica. El estudiante podrá tener en su pantalla su primer ejecutable escrito con Lazarus desde el Capítulo 2. No es un manual definitivo ni exhaustivo, pues no describe todas las funcionalidades incluidas en el programa, pero, en cambio, se centra en cómo aprender a programar.

Trabaje concienzudamente con el código fuente propuesto, que tiene a su disposición como material descargable, modifíquelo y observe los cambios; visite los foros y lea revistas de programación. Solo a través de la práctica podrá dominar Lazarus.

Aunque se ha puesto todo el esmero posible en asegurar la perfecta ejecución del contenido de la obra en Windows y Linux, los errores son propios de las personas, y en un libro intervienen muchas, por lo que las erratas existirán. Así pues, si encuentra algún error en el texto o en el código, no dude en ponerlo en conocimiento de la editorial a través del correo *editorial@ra-ma.com*. De este modo permitirá corregirlas en siguientes ediciones y ahorrará a futuros lectores importantes quebraderos de cabeza.

Para finalizar, quiero dar las gracias a Editorial Ra-Ma por su confianza y buen hacer para llevar a término este libro, así como a todos aquellos alumnos que han participado, directa o indirectamente reproduciendo el código de los ejemplos aquí propuestos.

No quiero terminar esta breve introducción sin antes recordar y mostrar mi agradecimiento a todo el equipo humano que ha estado y está detrás de este proyecto, y a todas las personas que, aun no siendo desarrolladoras del programa, colaboran haciendo uso de él, presentando y aportando continuamente propuestas e ideas.

# 1

# EL ARTE DE PROGRAMAR

Programar un ordenador consiste en lograr que la máquina haga exactamente lo que el programador desea, de la forma más eficiente posible y sin errores.

Todo ordenador está formado por dos subsistemas complementarios y esenciales: el **hardware**, que constituye la parte física y tangible de la máquina, y el **software**, que es su parte lógica, los programas que dan las instrucciones necesarias a la CPU o microprocesador para que ejecute las tareas deseadas. El uno, sin el otro, sirve para poco.

Toda la electrónica de los ordenadores se fundamenta en componentes de estado sólido, en general, y transistores, en particular. Esto hace que en su nivel más bajo, los microprocesadores solo puedan interpretar estados altos y bajos de tensión. En otras palabras, es como si solo existieran interruptores, cada uno de los cuales únicamente puede estar en uno de los dos estados: *on* u *off*, abierto o cerrado, alto o bajo. Si a uno de esos estados lo identificamos con el 1, por ejemplo, al otro le corresponde el 0; es decir, asociamos a los niveles eléctricos los dígitos binarios 0 o 1. Cada uno de estos dígitos se denomina en informática **bit**, acrónimo inglés de *binary digit*.

La evolución del software a lo largo de los años ha ido pareja al desarrollo en complejidad del hardware. Con el tiempo, los microprocesadores han ido integrando en su chip centenares de millones de transistores, incluso más de mil millones en algunos modelos, y en espacios cada vez más reducidos.

Así pues, ya conocemos que los ordenadores digitales solo pueden manejar estos estados discretos binarios a los que hemos denominado *bits* y representado

por los dígitos 0 y 1. La pregunta que ahora se plantea es la siguiente: ¿cómo se consigue trasladar nuestro lenguaje natural al código máquina que lee e interpreta un ordenador? Esa es la pregunta clave. Hemos de ser capaces de traducir el algoritmo que resuelve cualquier problema a **lenguaje de máquina** o **código máquina**, el único interpretable y ejecutable por un microprocesador.

Para empezar, hemos de representar nuestro lenguaje natural con cadenas de bits; de igual modo, después habrá que convertir secuencias de 0 y 1 en algo entendible por las personas. En definitiva, tenemos que ser capaces de establecer comunicaciones con los ordenadores.

La información numérica es fácil de convertir del sistema decimal en el que las personas nos movemos al sistema binario que manejan los ordenadores digitales. Observe la siguiente tabla:

| **Decimal** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Binario** | 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 | 1000 | 1001 |

Hemos conseguido relacionar en ella todos los dígitos que empleamos en nuestro mundo real con los del sistema binario, y viceversa. ¿Qué ocurre, entonces, con los caracteres alfabéticos? La pregunta también tuvo una fácil respuesta. Los ingenieros consiguieron desarrollar códigos alfanuméricos que relacionaban una combinación binaria específica con cada carácter. A nivel conceptual estos códigos no presentan ninguna dificultad, ya que por muy grande que sea el número de caracteres, siempre será finito, por lo que el único problema reside en consensuar con la comunidad informática qué combinación binaria se asigna a cada carácter. Por ejemplo, al usar el **código ASCII**, el patrón `1010011` se corresponde con la letra S (mayúscula).

Para programar los ordenadores, por tanto, necesitamos asignar significados reales a los patrones de bits. No solo para los caracteres alfanuméricos, sino también para las instrucciones. Los ordenadores necesitan datos, pero también instrucciones y, de algún modo, necesitamos decirle: “toma los números enteros 2 y 3, súmalos y devuelve el resultado en coma flotante”, por ejemplo. Esto, como puede imaginar, ya no es tan sencillo, pues esos datos e instrucciones que acabamos de poner por escrito habrá de suministrárselos en código máquina (de nuevo, cadenas de ceros y unos). Y para esto, precisamente, han ido surgiendo los diferentes lenguajes de programación.

## 1.1 LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación es un lenguaje formal y artificial diseñado para expresar procesos que pueden ser llevados a cabo por las computadoras. Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

Como no podemos trabajar directamente en binario, en código máquina, los primeros intentos consistieron en implementar traductores para reemplazar los bits por palabras o abstracción de palabras y letras del idioma inglés, lo que se conoció como **lenguaje ensamblador**. Este lenguaje es de bajo nivel y específico de la arquitectura de la máquina o circuito que desea programarse, lo que lo hace aún más complicado. Por ejemplo, en lenguaje ensamblador, para un procesador de arquitectura x86, la sentencia

```
MOV AL, 037h
```

asigna el valor hexadecimal 37, el número decimal 55, al registro `AL`.

El lenguaje ensamblador lee la sentencia de arriba y traduce su equivalente binario en lenguaje de máquina, que resulta ser `1011 0000 00110111`. El código máquina generado por el ensamblador consiste en dos bytes. El primero contiene empaquetada la instrucción `MOV` y el código del registro hacia donde se mueve el dato, mientras que el segundo byte especifica el número que se asignará al registro (Figura 1.1).

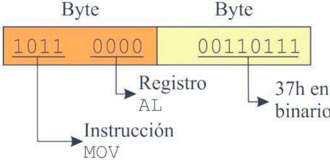



**Figura 1.1.** Código máquina de la instrucción ensamblador MOV AL, 037h

Al ser un lenguaje específico de los circuitos que se programan, rápidamente puede quedar obsoleto, pues los cambios en el hardware avanzan a un ritmo frenético, y, con ellos, el conjunto de instrucciones necesarias. Por tanto, desde el punto de vista del diseño, lo que necesita un programador es disponer de un lenguaje que pueda aplicar a distintas plataformas, independientemente de la arquitectura del microprocesador o del sistema operativo que aquel maneje. Este, en principio, es el objetivo que buscan desde finales de los años 50 del siglo pasado los lenguajes de programación de alto nivel.

Un **lenguaje de alto nivel** se caracteriza por encontrarse más cercano al lenguaje natural que al lenguaje de máquina. Se trata de lenguajes independientes de la arquitectura del ordenador, por lo que, en principio, un programa escrito en un lenguaje de alto nivel se puede migrar a otra máquina sin grandes complicaciones. Permiten al programador olvidarse por completo del funcionamiento interno de la máquina para la que está implementando el programa.

Una vez escrito el programa, para que se pueda ejecutar, deberá traducirse a código máquina, bien mediante **compilación**, bien mediante **interpretación**.

## 1.2 PASCAL, UN LENGUAJE DE PROPÓSITO GENERAL

**Pascal** es un lenguaje de programación de propósito general y de alto nivel, denominado así en honor al científico Blaise Pascal. Fue desarrollado por el profesor suizo Niklaus Wirth entre los años 1968 y 1969, y publicado en 1970. Su objetivo era crear un lenguaje que facilitara el aprendizaje de programación a sus alumnos. Con el tiempo, su utilización se extendió más allá del ámbito académico para convertirse en una herramienta para la creación de aplicaciones de todo tipo.

**Pascal** se caracteriza por ser un lenguaje de programación **estructurado** y fuertemente **tipado**. Esto implica que:

- El código está dividido en porciones fácilmente legibles llamadas *funciones* o *procedimientos*. De esta forma, Pascal facilita la utilización de la programación estructurada.

- El tipo de dato de todas las variables debe ser declarado previamente para que su uso quede habilitado.

El lenguaje emplea un reducido vocabulario de palabras en inglés reservadas para su uso, lo que lo hace realmente atractivo para empezar a programar; además, enseña buenos hábitos de programación.

La Real Academia Española recopila unas 88.500 palabras en su diccionario. Si se incluyen términos técnicos y otros no recogidos, podemos cifrar en cerca de 280.000 las palabras de nuestro idioma. Si comparamos tal cifra con las poco más de 70 palabras reservadas que emplea Free Pascal en su versión orientada a objetos actual, nos damos cuenta de lo sencillo que resulta este lenguaje para aprender a programar.

En el año 1989, con el compilador de Turbo Pascal 5.5, Borland introdujo la programación orientada a objetos en Pascal. A mediados de los 90, con el resurgimiento de los sistemas tipo Linux, Pascal quedó relegado a un segundo plano en beneficio del lenguaje C. En ese momento solo Delphi, un Pascal orientado a objetos para Windows, consiguió mantenerse. Casi todo el mundo pretendía migrar a C++ o Java, pero su enorme complejidad para los novatos hizo que al final Delphi siguiera existiendo, y hoy sigue en el mercado con nuevas versiones por parte de la compañía Embarcadero.

Sin embargo, mientras Turbo Pascal se convertía en Delphi, comenzaba a gestarse el proyecto **Free Pascal**, un compilador de Pascal orientado a objetos pero con tres características esenciales que, a la larga, fueron enormes ventajas: se trataba de un proyecto de **código abierto**, completamente **gratuito** y **multiplataforma**, lo que hoy permite programar aplicaciones orientadas a objetos para más de quince arquitecturas diferentes sin costes de licencias. Estas tres indudables ventajas han ido relegando en muchos campos a Delphi en beneficio de Lazarus.

## 1.3 LAZARUS

Ya hemos comentado que cualquier lenguaje de alto nivel, como Pascal, permite a los programadores implementar aplicaciones dando instrucciones a la CPU con una sintaxis más parecida a nuestro modo de hablar. Lógicamente, el microprocesador no entiende Pascal, ni C, ni Java; por tanto, el código Pascal deberá ser traducido a señales electrónicas digitales que sean interpretables. Este proceso de **traducción** es lo que denominamos **compilación**, y produce, si no se encuentran errores, un módulo llamado **objeto**. Este módulo es la traducción a lenguaje de máquina del código fuente escrito por el programador, pero aún no es ejecutable. Es necesario someterlo, a su vez, a otro proceso conocido como **enlazado**, tras el cual se obtendría el programa ejecutable (Figura 1.2).

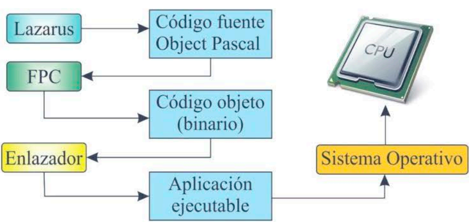



**Figura 1.2.** De Lazarus a la CPU. Desarrollo completo de una aplicación

Lazarus, como entorno de desarrollo integrado, emplea el compilador Free Pascal para producir su propio código objeto. Integra, en una única aplicación, todas las herramientas y funciones que necesitará para programar y obtener un ejecutable partiendo solo de la idea. Es, como hemos dicho, una herramienta RAD que poco tiene que envidiar al buque insignia de Embarcadero.

El IDE de Lazarus le proporcionará:

- Un fantástico editor de código fuente diseñado para que rápidamente pueda leer y escribir aplicaciones.

- Un completo conjunto de herramientas visuales para que pueda programar cómodamente la interfaz de usuario.

- Tres bibliotecas listas para ejecutar rutinas de código y todos sus componentes: RTL, FCL y LCL.

- Herramientas para compilar, ensamblar y enlazar sus módulos con el único objetivo de conseguir una aplicación lista para ejecutarse.

- Herramientas para analizar, probar, documentar y depurar el código fuente que escriba.

Además, este entorno de desarrollo incluye múltiples componentes auxiliares que le ayudarán a:

- Diseñar formularios especializados.
- Depurar programas.
- Convertir proyectos Delphi a Lazarus.
- Analizar y probar módulos de código.
- Documentar rutinas, bibliotecas y programas.

Los programadores experimentados se enfrentan a menudo con problemas a la hora de compilar sus programas; así pues, no se preocupe de los que pueda encontrar usted que por primera vez se acerca a este mundo. Sin embargo, si tenía dudas de si este libro era para usted, la respuesta es sí. Lazarus es una elección perfecta para aprender a programar por varias razones: Lazarus emplea Pascal, y Pascal es un lenguaje fácil de aprender, con una estructura sencilla y un reducido número de palabras reservadas. Además, enseña buenos hábitos de programación.

Ilustremos su sencillez con un ejemplo: el tradicional "Hola mundo" con el que se presentan todos los lenguajes de programación, Pascal incluido. En un lenguaje como C, sería algo así:

```
#include <stdio.h>
int main(void) {
  printf("Hola mundo.\n");
  return 0;
}
```

¡Quince componentes léxicos para escribir en el terminal dos palabras! La cosa no mejora mucho en C++:

```
#include <iostream>
int main(void) {
  std::cout << "Hola mundo." << std::endl;
}
```

Y en Java resulta complicado hasta lo inverosímil:

```
public class HolaMundo {
  public static void main(String [] args) {
    System.out.println("Hola mundo.");
  }
}
```

¿Cómo se le puede explicar a alguien que no ha visto un programa antes tal galimatías de órdenes? Pascal va directo al grano:

```
begin
  writeln('Hola mundo');
end.
```

Naturalmente, este programa no es determinante a la hora de escoger un lenguaje como primera opción, pero sí debería suscitar una seria reflexión por parte de quien va a aprender su primer lenguaje, y de quien va a enseñar a profanos.

Las ventajas de Pascal, desde luego, no acaban aquí. Lazarus/FPC nació como proyecto de código abierto, en contraposición al software propietario de Delphi. Esto significa que todo su código fuente está puesto a disposición de cualquier persona. Aprovechando esa libertad, un grupo de programadores y otros colaboradores corrigen y mejoran el proyecto sin cobrar por ello de ninguna empresa. Además, y este es un punto muy interesante, es un software completamente gratuito. No hay que pagar licencias ni siquiera para desarrollar aplicaciones profesionales.

Como software libre, existen muchas, muchas versiones. A veces, diariamente, surge una nueva compilación. Sin embargo, tanto FPC como el IDE de Lazarus publican y numeran nuevas versiones solo cuando han sido largamente probadas, incluso en aplicaciones comerciales desarrolladas con Lazarus/FPC, y cuando los errores hallados hasta la fecha han sido corregidos.

Para resolver cualquier duda que pueda surgirle, tiene a su disposición a la comunidad que colabora en el proyecto desde la wiki *http://wiki.lazarus. freepascal. org*. Así mismo, también puede participar resolviendo, o planteando problemas, a través de las listas de correo y foros. Puede acceder al foro principal de Lazarus desde *http://www.lazarus.freepascal.org/index.php?action=forum*. Si lo necesita, podrá también descargar y consultar toda la documentación del proyecto: *http://sourceforge. net/projects/lazarus/files/Lazarus%20Documentation*. Existen varios tutoriales gratuitos en la Red que ofrecen alternativas al material que hemos recogido en este libro. Uno muy interesante está en la dirección *http://wiki.lazarus.freepascal.org/ Lazarus_Tutorial*.

Lazarus se instala con una colección de proyectos de ejemplo. Puede acceder a ellos para estudiarlos a través del navegador que encontrará en el menú **Herramientas | Proyectos Ejemplo…** Así mismo, existe un excelente repositorio de código fuente que podrá usar en sus propios proyectos –también puede contribuir con código propio–, en *http://sourceforge.net/projects/lazarus-ccr*.

Todas las capturas de pantalla de este libro se han realizado en las versiones **1.0.14** de **Lazarus**, publicada el 16 de noviembre de 2013, y **2.6.2** del **compilador Free Pascal**, del 23 de febrero de 2013.

Windows es la plataforma Lazarus más popular, al menos así lo recogen las estadísticas de descarga de SourceForge, por lo que la hemos elegido para escribir el libro. Si usted trabaja en Linux o Mac Os X, es posible que las capturas sean algo diferentes a las que observa en su pantalla. Sin embargo, Lazarus es completamente multiplataforma, por lo que todos los ejemplos propuestos se compilan en todas las plataformas, tanto en 32 como en 64 bits.

# 2

# LAZARUS Y PASCAL

El entorno de desarrollo integrado (IDE) para Lazarus se basa en una extensión orientada a objetos del lenguaje de programación Pascal conocida como **Object Pascal** o Pascal orientado a objetos. Si Delphi fue el lenguaje elegido por Borland para el entorno de desarrollo que hoy conocemos como Embarcadero Delphi, **Free Pascal** es el motor que corre bajo Lazarus. De hecho, el binomio Lazarus/Free Pascal es el que permite desarrollar programas tipo Delphi en un gran número de plataformas y a coste cero.

Hoy, la mayoría de los lenguajes de programación soportan programación orientada a objetos (POO). Estos lenguajes se basan en tres conceptos básicos: la **encapsulación**, normalmente implementada mediante clases, la **herencia** y el **polimorfismo**. Aunque se puede escribir código en Object Pascal sin entender las características principales del lenguaje, no es posible dominar el entorno de desarrollo de Lazarus sin comprender el lenguaje de programación. Por tanto, a lo largo de este capítulo, estudiará las características principales de Pascal y Lazarus para que comience a programar lo antes posible.

## 2.1 EL IDE DE LAZARUS

Asumimos que ya ha descargado e instalado el programa en su ordenador, y que está familiarizado con su sistema operativo.

Lazarus siempre abre un proyecto vacío, aunque puede configurarlo para que edite el último proyecto en el que trabajó (Figura 2.1).

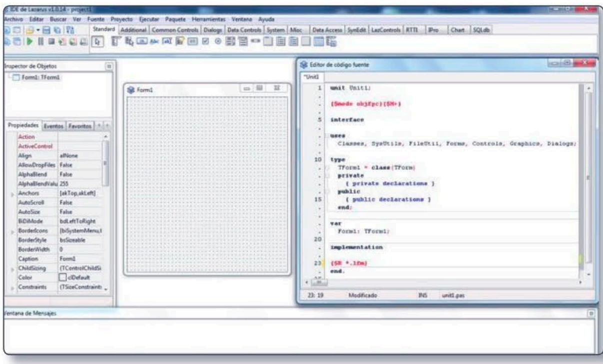


**Figura 2.1.** El entorno de Lazarus corriendo bajo Windows

Un **proyecto** de Lazarus es una colección de archivos relacionados con una aplicación específica. Los proyectos contienen la información necesaria para construir la aplicación.

El entorno de desarrollo de Lazarus permite programar aplicaciones tanto en modo **consola** como en modo **gráfico**.

Además de las barras de menús, de título y herramientas, el entorno de Lazarus arranca con los siguientes elementos:

- La Paleta de componentes.
- El Inspector de objetos.
- El Editor de código fuente.
- El Editor de formularios.
- La Ventana de mensajes.

El Editor de formularios puede no ser visible porque esté oculto tras el Editor de código fuente. Si este fuera su caso, pulse la tecla **[F12]** para que aquel pase al primer plano. Una nueva pulsación llevará de nuevo al fondo al Editor de formularios.

El nombre que por defecto Lazarus asigna a cualquier proyecto es **`project1`**, título que se muestra en la barra de título en la parte superior izquierda de la ventana

del IDE, junto al número de versión. En el momento en que lo guarde en disco podrá asignarle el nombre requerido, al que se añadirá automáticamente la extensión `.lpr` (***La**zarus **pr**oject*).

Para guardar un proyecto, podrá elegir cualquiera de las tres alternativas siguientes:

1. Vaya al menú **Proyecto | Guardar proyecto** o **Proyecto | Guardar proyecto como…**
2. Pulse el icono del **disquete** en la barra de herramientas.
3. Pulse la combinación de teclas **[Ctrl] + [S]**.

**IMPORTANTE**
Guarde cada proyecto en su propia carpeta, de este modo evitará que se produzcan errores futuros.

A lo largo del libro, y para evitar posibles confusiones, nos referiremos a la **Paleta** para indicar la Paleta de componentes; al **Editor**, como Editor de código fuente, y al **Formulario** como Editor de formularios.

En todo momento podrá cerrar cualquier ventana pulsando sobre el icono del **aspa** de su barra de título, a través del atajo de teclado **[Ctrl] + [F4]**, o eligiendo la opción **Cerrar** de su menú contextual.

Todas las ventanas del proyecto se listan en el menú **Ventana**. Si en cualquier momento necesita encontrar una ventana, localícela en este menú y haga clic sobre su nombre para fijar el foco sobre ella.

**NOTA**
A lo largo del libro asumiremos que los atajos de teclado son los que se instalan por defecto, pues también es posible que cada usuario los asigne desde el menú **Herramientas | Opciones | Editor | Atajos de teclado**.

## 2.2 MODO CONSOLA FRENTE A INTERFAZ GRÁFICA

Lazarus le permitirá programar dos estilos diferentes de programa, aquellos que funcionan en modo **consola** y los que lo hacen con una **interfaz gráfica** de usuario (Figura 2.2).

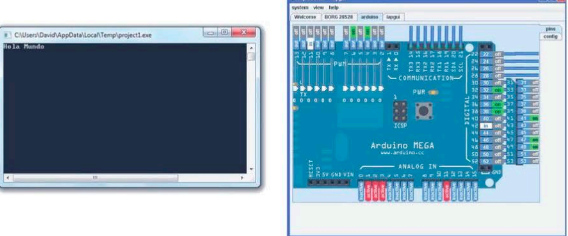



**Figura 2.2.** Ejecución de un programa en modo consola (izquierda) y con interfaz gráfica (derecha)

Los programas que se ejecutan en modo consola poseen una interfaz textual y, generalmente, se les pasan los datos a través de una línea de comandos. Están diseñados para hacer una tarea particular de forma sobresaliente, y son realmente rápidos. Eran los únicos programas que podían diseñarse en los comienzos de la programación. Los programas con interfaz gráfica, por otro lado, son más llamativos, más pesados y más complicados de implementar por su programación basada en eventos.

Lógicamente, dedicaremos los primeros capítulos del libro a diseñar programas en modo consola, por ser estos más sencillos para los principiantes y la base que se requerirá para hacer aplicaciones gráficas posteriores.

Lazarus arranca siempre con un proyecto gráfico en blanco; sin embargo, puede desecharlo y elegir un proyecto en modo consola. Vaya al menú **Proyecto | Nuevo proyecto** y elija la opción **Programa** (Figura 2.3). Esta opción le permitirá implementar un programa con Free Pascal. Su código fuente será mantenido automáticamente por Lazarus.

Cuando pulse el botón **Aceptar**, Lazarus le mostrará en el **Editor** el esqueleto de una aplicación Pascal en modo consola con el siguiente código fuente:

```
program Project1;
  {$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
begin
end.
```

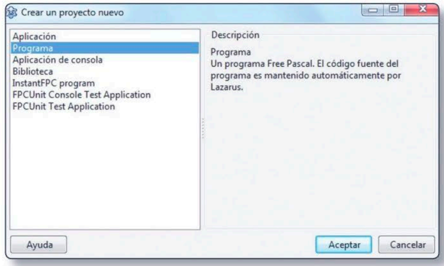



**Figura 2.3.** Creación de un programa Free Pascal en modo consola

Fíjese en que Lazarus ha llamado al programa `Project1` y le ha asignado el fichero `project1.lpr`. Sin embargo, el programa solo existe de momento en memoria hasta que no lo guarde en disco. Cree una nueva carpeta llamada `capitulo2` y seleccione **Proyecto | Guardar proyecto como…** En el cuadro de diálogo, dele el nombre `primerproyecto` y guárdelo.

Si observa ahora el contenido de la carpeta `capitulo2`, verá que Lazarus ha creado tres archivos:

```
primerproyecto.lpi
primerproyecto.lpr
primerproyecto.lps
```

**IMPORTANTE**
Es recomendable escribir todos los nombres de programas y archivos en letras minúsculas para evitar errores al trabajar en distintas plataformas, como Linux o Mac.

El fichero `.lpr` es el archivo de Pascal tal como se muestra en el **Editor** y significa, indistintamente, ***L**azarus **pr**ogram* o ***L**azarus **pr**oject*. Los archivos `.lpi` (***L**azarus **p**rogram **i**nformation*) y `.lps` (***L**azarus **p**roject **s**ettings*) son ficheros de texto en formato XML que emplea Lazarus para almacenar información relativa al proyecto para que pueda restaurar la sesión en cualquier momento tal y como la dejó al grabar el proyecto.

## 2.3 EL PRIMER PROGRAMA

Siempre que comienza un proyecto, sea en modo consola o en modo interfaz gráfica, Lazarus le mostrará el esqueleto básico en Pascal que debe poseer su aplicación. Así pues, aprovechando dicha estructura básica, añadamos el código que necesitamos para compilar nuestro primer programa. Sitúe el cursor entre las líneas **begin** y **end** y escriba:

```
writeln('Hola mundo');
readln;
```

Su programa se parecerá a esto:

```
program primerproyecto;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

begin
  writeln('Hola mundo');
  readln;
end.
```

Observe que en la barra de estado, en la parte inferior del **Editor**, aparecen el nombre y ruta completos del archivo, y la etiqueta **Modificado**, y, en su parte superior, con un asterisco *, el nombre del proyecto (`*primerproyecto.lpr`). El asterisco indica que el proyecto se ha modificado y aún no se ha guardado.

Para compilar el programa y ver su ejecución, elija **Ejecutar | Ejecutar**, pulse **[F9]** o el botón ▶ de la barra de herramientas. Verá una serie de mensajes en la **Ventana de mensajes**; si todo ha ido bien, leerá:

```
El proyecto "primerproyecto" se ha construido correctamente
```

En ese momento, aparecerá una ventana de Windows en modo consola con la ejecución del programa, que le muestra el texto `Hola mundo`, que se corresponde con la sentencia Pascal `WriteLn('Hola mundo')`. Pulse la tecla **[Enter]** para cerrar el programa (Figura 2.4).

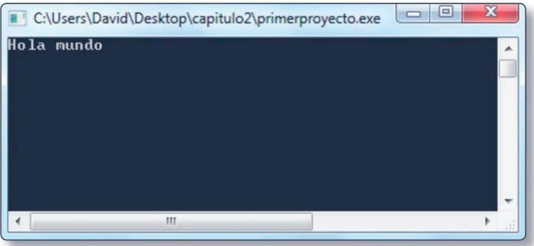



**Figura 2.4.** El primer ejecutable corriendo en Windows

Si observa ahora el contenido de la carpeta `capitulo2`, encontrará un nuevo archivo: `primerproyecto.exe` (en Windows) y `primerproyecto` (en Linux). Además, hallará dos carpetas de nombres `lib` y `backup`, que Lazarus creará con cada nuevo proyecto (Figura 2.5).

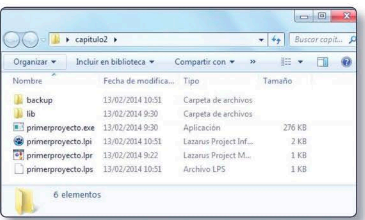



**Figura 2.5.** Archivos y carpetas del primer proyecto

La carpeta **backup** contiene las copias de seguridad de los tres archivos primerproyecto (primerproyecto.lpi.bak, primerproyecto.lpr.bak, primerproyecto.lps.bak). La carpeta **lib**, por el contrario, contiene una nueva carpeta según la plataforma en la que esté trabajando, carpeta que posee los binarios que resultan del procedo de compilación y enlazado. Puede olvidarse de aquí en delante de esta carpeta. Si en algún momento se borra, Lazarus la creará de nuevo cuando sea necesario.

## 2.4 ESTRUCTURA DE UN PROGRAMA PASCAL

La sintaxis del lenguaje Pascal es bastante explícita y más legible que la del lenguaje C, por ejemplo. Así pues, su extensión orientada a objetos sigue la misma senda y ofrece la misma potencia que otros lenguajes POO, desde Java a C#. Pascal se caracteriza por ser un lenguaje de programación estructurado fuertemente tipado y con una rígida sintaxis, pero esto no es un inconveniente, ni muchísimo menos.

Todo programa Pascal presenta una estructura similar mínima:

- Una **cabecera**, que consiste en la palabra reservada **program**, seguida del nombre que haya elegido para el proyecto y acabada en un punto y coma.

- La **cláusula** opcional **uses**, que indica a qué otras unidades tenemos que acceder, lo que incluye las unidades que definen los tipos de datos a que nos referimos. Los programas en Pascal se construyen a partir de módulos o **unidades** que se almacenan y compilan de forma independiente.

- El **cuerpo** del programa, constituido por una serie de instrucciones entre las palabras reservadas **begin** y **end**.

Así pues, y para resumir, podemos decir que un programa en Pascal tiene la siguiente estructura general:

```
program nombre;
uses unidadA, unidadB,… unidadN;
begin
  // Aquí se escribe el código
end.
```

Mire ahora de nuevo el código fuente del primer programa que hizo:

```
program primerproyecto;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

begin
  writeln('Hola mundo');
  readln;
end.
```

Observará que, en efecto, obedece a la estructura general de un programa, pero, además, incluye comentarios en color entre llaves. Los comentarios hacen que el código parezca más complejo, pero, gracias a ellos, es más fácil de leer.

Las palabras en **negrita** se corresponden con las declaraciones en Pascal, sus palabras reservadas o claves, como **`begin`** o **`end`**.

Si eliminamos los comentarios del código, resulta la estructura general de todo programa Pascal:

```
program primerproyecto;
uses
  cthreads, Classes;

begin
  writeln('Hola mundo');
  readln;
end.
```

Lazarus ha supuesto que podríamos necesitar las unidades `cthreads` y `Classes`. En realidad, para este ejemplo tan sencillo, no son necesarias, puesto que las sentencias `writeln` y `readln` se encuentran en la unidad **`system`**, que se carga siempre por defecto en cualquier proyecto de Lazarus. De modo que podemos eliminar la cláusula **`uses`** y dejar el código como sigue:

```
program primerproyecto;

begin
  writeln('Hola mundo');
  readln;
end.
```

Si lo compila obtendrá el resultado que se muestra en la figura 2.4.

## 2.5 COMENTARIOS

Las partes del texto coloreadas que aparecen en el código fuente son **comentarios**. Si se fija bien, en el programa que ha escrito aparecen dos tipos de comentarios en dos colores distintos: **`{$mode objfpc}`** y **`{ you can add units after this }`**.

En lo que respecta al compilador de Free Pascal, los comentarios pueden pertenecer a una de estas dos categorías:

- **Directivas del compilador**. Lazarus las indica en color rojo y empiezan con los símbolos **`{$`**.
- **Comentarios generales**. El IDE los colorea en azul y pueden indicarse con los símbolos **`//`**, **`{`** o **`(*`**.

El Editor colorea los comentarios al vuelo. Si desea modificar las opciones de color, seleccione **Opciones**, del menú **Herramientas** y vaya hasta el campo **Editor | Visualizar | Colores** (Figura 2.6).

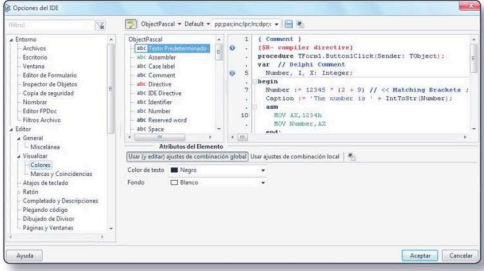



**Figura 2.6.** Ajustes de combinación de colores del entorno de desarrollo Lazarus

Los comentarios generales son ignorados por el compilador a la hora de construir las aplicaciones, pero son de gran ayuda para los programadores, sobre todo para los analistas. Pueden abarcar desde una sola línea (para los que se suele usar `//`) hasta varias, con los símbolos `{ }` o `(* *)`.

Las directivas del compilador son también comentarios, puesto que están entre llaves, pero todas ellas comienzan con el símbolo `$` para indicar al compilador que son instrucciones para él. Por tanto, aunque no sean código Pascal, el compilador no puede ignorarlas. Si bien no influyen en la compilación del código, sí lo hacen en cómo el compilador trata dicho código, y, por tanto, pueden llegar a ser esenciales.

## 2.6 LOS NOMBRES EN PASCAL

Como ya hemos comentado, una aplicación en Pascal está formada por una o varias unidades, lo que permite una programación modular. Como programador, tiene plena libertad para nombrar como desee al programa, sus unidades y variables en los mismos, siempre y cuando se atenga a las siguientes **restricciones** y **sugerencias**:

- Los nombres en Pascal solo pueden albergar caracteres alfanuméricos (a…z, 0…9) y el guión bajo (_).
- El primer carácter de un nombre puede ser cualquiera de los anteriores, salvo los dígitos. Así, `radio2` es un nombre correcto, pero `2radio` no.
- Aunque no es una restricción, se recomienda siempre usar nombres y variables en minúsculas. De este modo se evitan problemas y confusiones a la hora de trabajar en distintas plataformas.
- Otra recomendación, y parece ser una norma no escrita, es emplear para todas las variables en el código fuente la tipografía conocida como *CamelCase*, que se caracteriza porque las palabras van unidas entre sí sin espacios; con la peculiaridad de que la primera letra de cada palabra se encuentra en mayúscula para hacer más legible el conjunto. Por ejemplo: `RadioCirculo`, `EjeMayor`, etc.
- Pascal no distingue entre mayúsculas y minúsculas, por lo que las variables `radio`, `Radio` o `RaDiO` son idénticas para el lenguaje. Si viene de otros lenguajes, como Java o C#, tenga cuidado.

- Cuando Lazarus asigna automáticamente un nombre, lo hace siguiendo un patrón: `project1`, `Form1` o `Button1`. No obstante, estos nombres no dicen nada, por lo que lo más conveniente es cambiarlos lo antes posible por nombres que tengan algún significado en el contexto de la programación. Por ejemplo, si en una aplicación aparecen las variables `a`, `b` y `c` de tipo `ra1`, esos nombres dicen muy poco a un analista. Si en ese mismo programa las variables son `Perro`, `Gato` y `Pez`, de tipo `TMascota`, ayudarán mucho más a la hora de seguir el flujo del programa.

## 2.7 DIRECTIVAS DEL COMPILADOR

Una **directiva de compilación** es una orden abreviada que instruye al compilador para suponer una determinada condición o realizar una acción dada durante el proceso de compilación de un programa. Por tanto, en nuestro entorno de desarrollo, controlará la naturaleza del código producido por el compilador de Free Pascal (FPC).

Afortunadamente para los principiantes, rara vez se tendrán que preocupar por estas directivas, pues las esenciales ya estarán insertadas por Lazarus en el código fuente abierto como plantilla.

No obstante, aunque no tenga que preocuparse por ellas en mucho tiempo, no viene mal conocer sus peculiaridades.

Las directivas de compilación se dividen realmente en tres tipos básicos:

- **Directivas conmutadoras**. Activan o desactivan alguna condición mediante el nombre de la directiva y un símbolo: + para activar y − para desactivar. Por ejemplo, la directiva **`$H`** determina si FPC trata las cadenas como `AnsiString`, sin límite de tamaño, **`{$H+}`**; o `ShortString`, con un límite de 255 caracteres, **`{$H-}`**.
- **Directivas parámetros**. Proporcionan parámetros al compilador, tales como nombres de archivos o valores para asignación de memoria. Por ejemplo, **`{$mode objfpc}`** indica al compilador de Free Pascal que estamos usando Pascal orientado a objetos como dialecto de Pascal.
- **Directivas condicionales**. Controlan la compilación condicional de partes del código fuente. Por ejemplo, **`{$IFDEF WINDOWS}`** indica a FPC que si el programa se compila para Windows, debe incluir el código Pascal presente entre **`{$IFDEF WINDOWS}`** y **`{$ENDIF}`**. Si el programa se compila para Linux, FPC obviará el código.

# 3

# TIPOS, VARIABLES Y CONSTANTES

Los ordenadores trabajan con dos tipos de entidades: **instrucciones** y **datos**. Los **datos** constituyen la representación simbólica de un atributo o variable cuantitativa. Describen hechos empíricos, sucesos y entidades, tales como las fechas, los salarios de los empleados o las notas de un examen. Los datos representan la información que el programador manipula en la construcción de una solución o en el desarrollo de un algoritmo.

Por el contrario, las **instrucciones**, escritas como código en el programa, representan órdenes para la CPU del ordenador. A través de ellas, el programador le indica cómo operar con los datos y cómo interactuar con el resto de dispositivos del computador.

En este capítulo aprenderá todo lo que necesita conocer sobre los datos y cómo Pascal ayuda a organizarlos en las siguientes categorías:

- **Tipos**. Indican la naturaleza de los datos y se declaran con la palabra reservada `type`.
- **Variables**. Son identificadores simbólicos asociados a un espacio en memoria que contienen una cierta información. Se declaran mediante la orden `var`.
- **Constantes**. Son valores que no pueden modificarse durante la ejecución de un programa. Se declaran con la palabra `const`.

## 3.1 TIPOS

De las tres categorías que acabamos de definir, la más importante en Pascal es la de **tipo**. Como ya dijimos, Pascal es un lenguaje **fuertemente tipado**, lo que lejos de ser una desventaja, es una enorme ventaja para adquirir buenas técnicas de programación en todos los principiantes. Siempre es preferible localizar un error en tiempo de compilación que en tiempo de ejecución.

En Pascal, cada dato pertenece a un tipo y posee un tamaño en bytes definido, lo cual es esencial para gestionar la memoria. En la actualidad, ni los compiladores ni los sistemas operativos tienen dificultad para trabajar con grandes estructuras de datos, pero esto no ha sido siempre así, por supuesto. En cualquier caso, siempre debe aplicar la idea del ahorro en lo que respecta al uso de variables adecuadas. De este modo, la gestión de memoria, la depuración del programa y su funcionamiento serán más correctos.

El uso de **números**, **caracteres** y **valores lógicos** es corriente en programación. De hecho, casi todos los lenguajes disponen de ellos *a priori*, por lo que se les llama también tipos de datos **predefinidos** o **estándar**. Ciertamente, es posible ampliar nuestro lenguaje con otros objetos a la medida de los problemas que se planteen. Estos se construyen usando los datos predefinidos como las piezas más elementales, por lo que también se les llama *tipos de datos básicos* a los tipos predefinidos.

Los tipos de datos se caracterizan mediante sus **dominios**, es decir, el conjunto de valores que pueden adoptar, y las operaciones y funciones definidas sobre esos dominios.

### 3.1.1 Enteros

El tipo entero se corresponde con el concepto de número entero que empleamos en matemáticas, pero, atendiendo al tamaño o número de posibles valores, se diferencian los siguientes seis tipos de enteros predefinidos en Pascal: **`byte`**, **`shortint`**, **`word`**, **`integer`**, **`longint`**, **`int64`**.

Cada uno de estos tipos tiene un dominio determinado por el tamaño asignado en memoria. En principio, deben usarse los tipos que consuman menos espacio en memoria para acelerar el funcionamiento del programa y ocupar la menor cantidad posible de RAM durante su ejecución.

Por ejemplo, la edad de una persona es un ejemplo muy claro de tipo `byte`, pues el dominio de esta variable es un entero entre 0 y 255 ($2^8$). Ninguno de los otros cinco tipos de enteros predefinidos tendría ningún sentido, por lo que debe evitarse.

A continuación le presentamos una tabla con los **tipos de enteros** predefinidos en Pascal, su tamaño en memoria y el dominio asignado.

| Dominio | Tamaño en bytes | Tipo predefinido |
|---|---|---|
| **[0, 255]** | 1 | `byte` |
| **[−128, +127]** | 1 | `shortint` |
| **[−32.768, +32.767]** | 2 | `integer` |
| **[0, 65.535]** | 2 | `word` |
| **$[-2^{31}, 2^{31}-1]$** | 4 | `longint` |
| **$[-2^{63}, 2^{63}-1]$** | 8 | `int64` |

**Tabla 3.1.** Tipos numéricos enteros predefinidos

## 3.1.2 Ordinales

Pascal tiene la posibilidad de utilizar variables enteras de tipo ordinal; esto significa que, dado un valor, es capaz de calcular cuál es el siguiente valor permitido para esa variable. Como son números enteros y ordenados, se dice que son de **tipo ordinal**. Como en el caso anterior, elegiremos uno u otro tipo en función de la memoria necesaria.

Aparentemente, los tipos ordinales no tienen mucho sentido frente a los reales, sin embargo, no es cierto. Aplicaciones en las que figuren documentos de identidad, números de teléfono o cuentas bancarias pueden necesitar grandes números, estos son de tipo entero y se almacenan ordenados. De hecho, es esto lo que llevó a incluir enteros ordinales de 32 bits en los compiladores.

Todos los tipos de 1, 2 y 4 bytes recogidos en la tabla 3.1 son ordinales: `byte`, `shortint`, `integer`, `word` y `longint`; así como el tipo de 4 bytes **`longword`**, definido desde Delphi 4, y cuyo rango son los números enteros desde el 0 al 4.294.967.295.

### 3.1.3 Reales

Los números reales admiten la expresión ordinaria y la notación científica. Así, escribimos en Pascal el número π en notación decimal como 3.1415926, y la carga del electrón, en notación científica, como −1.6E−19. Dependiendo de la precisión que se requiera, podrán emplearse distintos tipos. En general, estos son números en coma flotante de 4, 8 y 10 bytes en su expresión.

Los números reales permiten trabajar con magnitudes muy grandes o muy pequeñas respecto de la unidad, positivas o negativas. En cualquier caso, si se trabaja con reales de 32 y 64 bits, se consiguen entre 8 y 16 cifras significativas de precisión.

Free Pascal usa la FPU del microprocesador para todos sus cálculos en coma flotante. El tipo real nativo **`real`** es dependiente del propio microprocesador o de las posibilidades de emulación, como se observa en la tabla 3.2.

Las variables de tipo real realmente compatibles en Turbo Pascal aparecen en la siguiente tabla:

| Dominio | Tamaño en bytes | Cifras significativas | Tipo predefinido |
|---|---|---|---|
| **Según CPU** | 4 o 8 | ¿? | `real` |
| **1.5E−45, 3.4E38** | 4 | 7 - 8 | `single` |
| **5.0E−324, 1.7E308** | 8 | 15 - 16 | `double` |
| **1.9E−4932, 1.1E4932** | 10 | 19 - 20 | `extended` |
| | 8 | 19 - 20 | `currency` |

**Tabla 3.2.** Tipos numéricos reales predefinidos en Free Pascal

El tipo **`currency`** (moneda) es un número real en coma flotante que se maneja internamente como un entero de 64 bits. Esto se hace así para evitar los errores de redondeo. De forma automática, se escala con un factor 10.000 para mostrar cuatro decimales.

### 3.1.4 Lógicos

El **tipo booleano** es la característica más importante de cualquier lenguaje de programación. Indica el valor verdadero (**`true`**, 1) o falso (**`false`**, 0) de una proposición cuando esta solo admite dos valores mutuamente excluyentes.

Las decisiones para controlar el flujo de programa mediante los operadores adecuados se toman, normalmente, por evaluación de funciones lógicas.

El tipo predefinido en Pascal para hacer referencia a una variable lógica es **`boolean`**, cuyos valores posibles son solo `true` o `false`. Su tamaño en memoria es de 1 byte.

Con el tiempo, ha sido obligatorio definir en Free Pascal nuevos tipos "lógicos" a medida que se han ido necesitando. Así ha ocurrido con **`byteBool`**, **`wordBool`** y **`longBool`**, de los tipos `byte`, `word` y `longint`, respectivamente. Todos ellos son perfectamente compatibles con `boolean` y han surgido por compatibilidad con ciertas rutinas de C y C++, donde cualquier entero distinto de cero significa `true`. Lo mismo ocurre con muchas funciones API de Windows, que devuelven 0 ante un error y cualquier entero distinto de cero en caso contrario.

### 3.1.5 Carácter

El **tipo carácter** es un elemento del conjunto de caracteres válidos del lenguaje. En Pascal, el tipo **`Char`** tiene exactamente un tamaño de 1 byte y contiene un símbolo ASCII.

Para indicar al compilador un carácter, este se escribe entrecomillado, empleando para tal fin la comilla simple (`'`) que aparece en el teclado junto con el signo de interrogación `?`, como en el ejemplo siguiente: `'a'` o `'A'`. También es posible indicar un carácter a través de su código ASCII, para lo cual se antepone el signo almohadilla `#` a su valor ordinal. Por ejemplo, `#97` sería para el compilador lo mismo que `'a'`.

La mayoría de los caracteres de control no se pueden teclear directamente en un teclado. Para ello, empleamos en el código fuente el símbolo # seguido de su código ASCII. De modo que podrá incluir los siguientes caracteres de control en el código fuente así:

```
#8  Retroceso
#9  Tabulador
#10 Nueva línea
#12 Nueva página
#13 Retorno de carro
#27 Escape
```

**NOTA**
Cuando tenga que representar la comilla simple, deberá escribirla dos veces seguidas, de modo que `''` representa el carácter comilla simple.

Debe tener en cuenta que algunos caracteres ASCII no son imprimibles (en principio, aquellos que van del 0 al 31) pero sí tienen un significado importante en programación, como salto de línea, retorno de carro, fin de fichero, etc.

El tipo **`WideChar`** en Free Pascal tiene 2 bytes de longitud y contiene un carácter codificado en UTF-16 según el estándar de codificación de caracteres Unicode. Puede especificar un carácter Unicode por su código, precedido, como en el caso anterior, por el símbolo `#`. Por ejemplo, `#$03b1` define la letra griega α.

Para distinguir `Char` de `WideChar`, la unidad del sistema define el tipo `AnsiChar`, que es idéntico al tipo `Char`. En la versión 2.6.2 de Free Pascal se anuncia la posibilidad de que `Char` sea simplemente un alias para `WideChar` y `AnsiChar` en próximas versiones.

### 3.1.6 Cadena

Free Pascal soporta el tipo **`String`** tal y como se definió en Turbo Pascal, es decir, como una secuencia de caracteres con un tamaño predefinido.

Como vimos al hablar de las directivas de compilación, Pascal podía trabajar con cadenas sin límite de tamaño, a las que llama `AnsiString`, y con cadenas cortas de una longitud máxima de 255 caracteres, definidas por el tipo `ShortString`.

`Si` se especifica la directiva `{$H-}`, entonces el compilador considerará que la longitud máxima de cualquier cadena será de 255 caracteres, salvo que se indique una longitud inferior en su declaración. Por ejemplo, en:

```
type
  Nombre = String[12];
  Calle = String;
```

`Nombre` puede contener un máximo de 12 caracteres, mientras que `Calle` podrá tener hasta 255.

**IMPORTANTE**
Para evitar errores entre plataformas, cuando trabaje con cadenas cortas, es mejor establecer la longitud con la función **`SetLength`**.

Las cadenas ilimitadas se declaran con el tipo predefinido `AnsiString`. Internamente se tratan como un puntero, de los cuales hablaremos más adelante, aunque su funcionamiento es tan transparente que se manipulan como si de una cadena corta se tratara.

Si se especifica la directiva `{$H+}`, entonces el compilador tratará cualquier cadena sin un especificador de longitud como ilimitada. Si se especifica la longitud, la considerará como una cadena corta.

Si la cadena está vacía (`''`) entonces la representación interna del puntero es **`Nil`**. Si la cadena no es nula, el puntero apunta a una estructura en memoria.

El compilador puede convertir un tipo de cadena en otro sin ningún problema, por lo que podrá mezclarlas en el código sin preocuparse demasiado por ello.

### 3.1.7 Tipo enumerado

El **tipo enumerado** hace referencia a variables que solo toman unos valores discretos que se especifican en la declaración. Se trata de un tipo ordinal en el que cada valor tiene un orden específico y posee un valor previo, excepto el primero; y uno posterior, excepto el último. Explícitamente deben declarar el nombre de cada posible valor y su orden específico en la secuencia. Por ejemplo,

```
type
  TDiasSemana = (lunes, martes, miercoles, jueves,
    viernes, sabado, domingo);
```

Note cómo hemos usado paréntesis `()` para delimitar el conjunto de elementos en el tipo enumerado, comas para separar los distintos elementos y el signo = en la declaración.

**NOTA**
Como el tipo enumerado que hemos definido no es uno predefinido de Pascal, comenzamos el nombre con la letra mayúscula **`T`**, de **tipo**. Es solo una convención en Delphi y Lazarus, pero ayudará a leer el código y a seguir el programa. De un vistazo, se distinguirán los nombres de los tipos de los nombres de las variables.

Asociado al tipo enumerado aparece el concepto de **subrango**, que no es más que un subconjunto restringido del enumerado:

```
type
  TDiasLaborables = (lunes..viernes)
```

Aquí `TDiasLaborables` es un subrango de 5 elementos del tipo enumerado `TDiasSemana`. Además, Free Pascal le permite asignar un valor ordinal a los elementos enumerados; por ejemplo, si realizamos la siguiente declaración:

```
type
  TPrimos = (dos = 2, tres = 3, cinco = 5, siete = 7,
    once = 11, trece = 13);
```

Tendremos un tipo enumerado donde el nombre de cada posible valor en el tipo se corresponde exactamente con su valor ordinal. Si la declaración la hubiéramos hecho así:

```
type
  TPrimos = (dos, tres, cinco, siete, once, trece);
```

Aunque los nombres de los elementos sean idénticos, sus valores ordinales serían como los de cualquier tipo enumerado por defecto. Comenzaría en el cero y se incrementaría de uno en uno hasta el último:

```
Ord(dos)=0, Ord(tres)=1, Ord(cinco)=2…
```

## 3.2 CAMBIO Y FORZADO DE TIPOS

En programación, el **forzado de tipos** consiste en indicar al compilador que trate a una variable de un determinado tipo como si fuera de otro. En realidad, el mecanismo empleado no modifica el elemento original, tan solo hace que se le trate de otra manera.

Pascal es un lenguaje fuertemente tipado, así que hay una necesidad imperiosa de convertir, e incluso forzar, tipos continuamente a lo largo del proceso de codificación. Por ejemplo, estudie el siguiente programa:

```
var
  a: byte = 8;
  b: byte = 3;
begin
  writeln('El valor de a - b es ', a - b);
end.
```

Lazarus le mostrará en la ejecución que el resultado es 5, sin embargo, para hacer eso, internamente el compilador ha realizado un cambio en el tipo de las variables de `byte` a `string`. Esto es así porque los procedimientos `Write/WriteLn` y `Read/ReadLn` requieren cadenas para trabajar con ellas. Así pues, la conversión se ha realizado de forma transparente al usuario. No obstante, la mayoría de las rutinas de Pascal no son tan versátiles y solo podrán trabajar con un tipo de dato.

Hay varias maneras de trabajar de forma sencilla con un lenguaje tan fuertemente tipado como Pascal:

- Usando funciones de cambio de tipo, como verá en unos instantes.
- Forzando la conversión de tipos, como haremos un poco más adelante.
- Utilizando el tipo `variant`, particularmente útil en programación de aplicaciones COM/OLE.

Pascal admite numerosas funciones para convertir explícitamente valores de un tipo a otro. Las principales rutinas para este fin son las siguientes:

| Rutina | Tipo convertido | Tipo resultado | Descripción |
|---|---|---|---|
| `Chr(b: byte): char` | Byte | Char | Convierte un valor entero en el carácter apropiado |
| `Ord(x: ordinal): integer` | Ordinal | Integer | Devuelve el entero de un ordinal |
| `Val(s: string; v: Number Type): integer` | String | Numeric | Convierte una cadena numérica entera a su número |
| `Trunc(r: real value): int64` | Extended | Int64 | Devuelve la parte entera del número en coma flotante |
| `Round(r: real value): int64` | Extended | Int64 | Redondea al entero |
| `Int(r: real value): realvalue` | Extended | Extended | Devuelve la parte entera del argumento |
| `IntToStr(value: integer): string` | Int64 | String | Convierte un entero en cadena |
| `DateToStr(a: TDateTime): string` | TDateTime | String | Devuelve la fecha formateada |
| `StrPas(p: PChar): shortstring` | PChar | shortstring | Trunca la cadena si posee más de 255 caracteres |
| `StrToInt(s: string):integer` | String | Integer | Convierte una cadena en entero |
| `FloatToStr(r: realvalue): string` | Extended | String | Convierte el número real en cadena |

**Tabla 3.3.** Principales rutinas de conversión de tipos en Free Pascal

Tenga en cuenta que el compilador espera siempre que ambos miembros de las declaraciones de asignación sean de tipos compatibles, pues en caso contrario informará con un mensaje de error. Si esto ocurre y ambos tipos tienen el mismo tamaño en memoria, todavía podríamos solucionarlo mediante el **forzado de tipos**, para lo que usamos el nombre del tipo y entre paréntesis la expresión que ha provocado el error. Por ejemplo, al compilar el código:

```
var
  i: integer;
  c: char;

begin
  i:= 'A';     // tipos incompatibles
  c:= 68;      // tipos incompatibles
end.
```

El compilador mostrará dos errores en los que nos informará de dos tipos incompatibles. Esto es así porque hemos declarado las variables `i` y `c` como de tipo entero y carácter, respectivamente, y luego les hemos asignado valores distintos. Sin embargo, como ocupan el mismo tamaño en memoria, podremos forzar sus tipos y escribir:

```
begin
  i:= integer('A');     // forzado a entero
  c:= char(68);      // forzado a carácter
end.
```

Ahora el código se compilará sin problemas, ya que la variable `i` tendrá el valor 65, que es el código ASCII de A, y la variable `c` será el carácter D, pues es quien tiene por código ASCII el valor 68.

Si los tipos que desea forzar ocupan diferente tamaño, pero son tipos ordinales, podrá hacerlo libremente sin que el compilador le informe de error alguno. Sin embargo, tenga en cuenta que, aun cuando no le informe de ningún error, podría obtener un comportamiento inesperado en la ejecución del programa. Si el forzado se hace de un tipo de mayor tamaño a otro de menor, inevitablemente perderá información en el proceso. Lo va a entender perfectamente con el siguiente programa:

```
var
  b: byte;
  i: integer = 1000;

begin
  b:= byte(i);     // convierte 1000 a 8 bits
  writeln ('El valor  de b ahora es ', b, ' no ',i);
  readln;
end.
```

Cuando lo ejecute verá cómo 1.000 se ha convertido ahora en el número 232 (Figura 3.1). ¿Por qué se ha producido esto? La razón es sencilla. Hemos forzado a almacenar el número decimal 1.000 como `byte`, que ocupa 8 bits. Sin embargo, 1.000 en binario requiere 10 bits: 11 1110 1000. Al forzar el tipo a 8 bits se pierden los dos más significativos y se almacena en memoria como 1110 1000. Al imprimir en pantalla este valor, `WriteLn` lo ha convertido a cadena y nos muestra su valor entero, que ahora resulta ser 232.

Así pues, a pesar de que el compilador no haya mostrado ningún error, podrá obtener un resultado inesperado. Tenga mucho cuidado cuando fuerce un tipo a otro que ocupe un menor espacio en memoria.

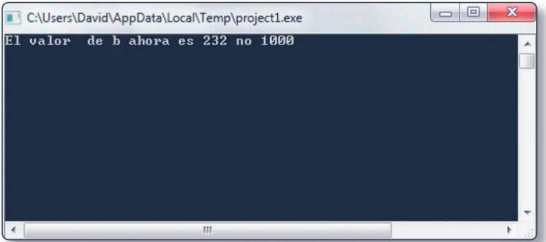



**Figura 3.1.** Pérdida de información en el forzado de tipos

## 3.3 VARIABLES

Son identificadores simbólicos asociados a un espacio en memoria que contienen una cierta información. Los tipos en Pascal solo resultan útiles cuando se declara una variable de dicho tipo, lo que se hace con la palabra reservada **`var`**. La existencia de variables se explica en programación porque el ordenador tiene que almacenar la información en algún lugar de la memoria y referirse a ellas en cualquier momento. Además, el valor de las variables podrá siempre modificarse en tiempo de ejecución

El compilador asocia cada variable del programa con una dirección de memoria en la que se almacena el contenido de aquella. Cuando el programa deja de usar la variable, el compilador se las arregla para liberar esa dirección de memoria.

Al igual que los tipos, Pascal obliga a **declarar las variables** antes de que se puedan usar. Esta declaración tiene la forma:

```
var NombreVariable: TipoVariable;
```

Puede declarar varias variables del mismo tipo en la misma línea separándolas por comas; por ejemplo:

```
var largo, ancho, alto: integer;
```

Las variables, mientras que el programador no les asigne un contenido, no poseen información utilizable por el programa. Algunos lenguajes inicializan con el valor cero o nulo hasta que el programador les asigne otro, pero Pascal no lo hace así, salvo que sean variables globales. Si se le olvida, el compilador le mostrará la advertencia:

```
Warning: Variable "x" does not seem to be initialized
```

Para inicializar una variable dentro de una rutina, lo hacemos en el mismo momento en que vamos a usarla, y lo escribimos así:

```
var contador: integer = 1;
```

Una vez declarada cualquier variable, se puede hacer uso de ella en el programa a través del **operador de asignación :=**. Por ejemplo, las siguientes son asignaciones válidas en Pascal:

```
var
  // declaraciones con :
  activo: boolean;
  longitud: integer;

begin
  // asignaciones con :=
  activo:= True;
  longitud:= 35;
end;
```

**IMPORTANTE**
El signo = solo se usa para definir tipos, los : para declarar variables, y el operador := para asignar valores a las variables.

El compilador de Free Pascal revisará el código que ha escrito en el momento que intente construir o compilar el proyecto. Cualquier mensaje de advertencia o error se le mostrará en la **Ventana de mensajes**. Una **advertencia** (`Warning`) no detendrá la compilación, pero debería prestarle atención para evitar futuros errores. Si se produce un **error** (`Error`), sí que se detendrá la compilación y, a menos que lo solvente, no podrá construir el programa.

No lo habíamos mencionado, pero se habrá dado cuenta de que la variable siempre se encuentra en el lado izquierdo de la asignación, y en su lado derecho, el valor que almacenará. Por ejemplo, no podríamos asignar

```
True:= activo;
```

porque el compilador se detendría para mostrar un error: "`Error: Variable identifier expected.`" Esto ocurriría, y a los principiantes les pasa mucho, porque `True` es una palabra reservada en Pascal y, por tanto, no puede usarse como nombre de variable.

### 3.3.1 Sentencias básicas de asignación

El compilador de Free Pascal ha tomado prestado de C la forma corta de realizar asignaciones con las variables, como se observa en la Tabla 3.4.

| Asignación | Resultado |
|---|---|
| **`A := Expr`** | *Expr* se almacena en A |
| **`A += Expr`** | *Expr* se suma a A y se almacena en A |
| **`A -= Expr`** | Resta de A *Expr* y se almacena en A |
| **`A *= Expr`** | Multiplica A por *Expr* y se almacena en A |
| **`A /= Expr`** | Divide A por *Expr* y se almacena en A |

**Tabla 3.4.** Operadores de asignación en Free Pascal

Este tipo de expresiones son muy útiles en el caso de que una variable se actualice a partir de su valor antiguo. Sin embargo, es importante tener presente que las variables a las que hace referencia se han tenido que inicializar previamente con su valor correcto. De otra forma obtendríamos errores en tiempo de ejecución, porque la variable puede tener un valor diferente de cero en el momento de la reserva de memoria por parte del compilador.

Es esencial entender que las asignaciones no son ecuaciones ni se debe despejar nada. Son solo instrucciones ejecutivas que deben interpretarse o leerse como "asigna a la variable A el resultado de la expresión que aparece después del operador :=".

## 3.4 CONSTANTES

Las **constantes** son valores que no pueden modificarse durante la ejecución de una aplicación. Se declaran en el programa dentro de la zona declarativa de las constantes, definida por la palabra reservada `const`. Como norma no escrita se estima que si un mismo objeto aparece más de tres veces en el programa, debiera definirse como constante. Esta restricción mejora la lectura del código de un programa y nos permite modificar en un único punto del código fuente un valor que se utiliza en varias zonas del mismo.

A diferencia de las variables, las constantes no necesitan ser explícitamente declaradas por tipo. El compilador decidirá a partir del valor de la constante cuál es el tipo más apropiado. Cualquier número ordinario, real, carácter o cadena puede ser una constante.

Por ejemplo, son constantes válidas para Lazarus, las siguientes:

```
const
  pi = 3.141592;
  Avogadro = 6.022E+23;
  NombreIDE = 'Lazarus';
```

## 3.5 PUNTEROS

Un **puntero** es una variable que referencia una dirección de memoria. En general, los punteros se emplean mucho en programación, aunque en unos lenguajes más que en otros. Su función principal es manejar los datos almacenados en la zona de memoria dinámica. Pascal, a diferencia de otros lenguajes, da un control absoluto al programador en cuanto a si quiere o no usarlos. No obliga a manejarlos, ni deja su control absoluto en manos del compilador.

En Pascal existe un tipo **puntero general** que permite apuntar a cualquier objeto o variable en memoria mediante su **dirección de memoria**, que se encuentra

a través del operador `@`. Además, podemos definir un **puntero tipado**, que indica el tipo de datos que almacena la dirección de memoria a la que aquel apunta.

Como en todos los lenguajes, los punteros permiten acceder a bajo nivel, del mismo modo que hace el compilador para almacenar datos e instrucciones. Los punteros poseen en memoria un tamaño fijo: 4 bytes en una máquina de 32 bits.

El programador podrá acceder a los datos o instrucciones almacenados en una dirección específica de memoria sin necesidad de saber el nombre de la variable, tan solo necesita conocer una dirección válida de memoria.

Para definir un puntero general se emplea la palabra **`pointer`**, mientras que para un **puntero tipado** empleamos el operador **`^`**. Este operador se usa de dos maneras. Si lo situamos justo antes del tipo, el acento circunflejo denota un tipo que representa punteros hacia las variables de ese tipo. Si se pone justo después de la variable definida como puntero tipado, se devuelve el valor almacenado en la dirección de memoria a la que apunta el puntero.

Las dos formas de definir un puntero son, entonces, las siguientes:

```
var
  p: pointer;     // p es un puntero general
  PByte: ^Byte  // PByte es puntero tipado, tipo Byte
```

Note cómo usamos como primera letra en la declaración del puntero tipado la **`P`** mayúscula. Es, de nuevo, otra convención, pero muy útil a la hora de seguir el flujo de programa y su depuración.

Lazarus maneja punteros a casi todos los tipos de datos estudiados: `char`, `integer`, `currency`, `extended`, etc.

Es recomendable para adquirir unos buenos hábitos de programación que evite usar punteros generales, pues como apuntan a cualquier objeto o variable en memoria, sin conocer su tipo, pueden resultar bastante peligrosos si provoca desbordamientos de memoria. Si necesita usar punteros en Pascal, úselos tipados.

Le presentamos ahora un breve programa, **`punteros`**, que hace uso de un puntero a variables de tipo `longint` (`PEnteroLargo`) para acceder a los datos a los que apunta de dos formas distintas: a través del propio puntero mediante la variable `intPtr`, de tipo puntero a enteros, y mediante el nombre de la variable (`unEntero`). Tanto `unEntero` como `intPtr^` hacen referencia al mismo valor.

Abra un nuevo proyecto de Lazarus en modo consola, llámelo `punteros` y ajuste la plantilla de código que le aparece en pantalla para que se parezca a lo siguiente:

```
program punteros;
{$mode objfpc}{$H+}
type
   PEnteroLargo = ^longint; // declara el puntero
 var
   unEntero : longint = 127;
   intPtr: PEnteroLargo;  //variable tipo puntero
begin
  intPtr := @unEntero; // asigna dirección memoria
  writeln  ('El valor de intPtr^ es: ', intPtr^);
  Inc(intPtr^, 3); // aumentamos en 3 el valor
  writeln ('El valor de unEntero tras Inc(intPtr^, 3)
            es: ', unEntero);
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Compile y ejecute el programa, ¿obtiene los resultados esperados? Veamos más despacio cómo funciona el programa. Tras definir la variable de tipo puntero `intPtr`, se recoge en esta última mediante `@unEntero` la dirección de memoria donde se ha almacenado el número 127 (Figura 3.2).

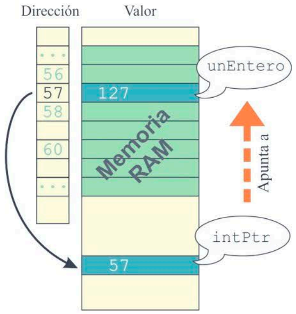
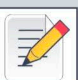



**Figura 3.2.** Un puntero contiene la dirección de memoria de otra variable

Para recuperar e imprimir ahora el valor de la variable `unEntero` situada en esa dirección de memoria, empleamos el operador ^ aplicado a la variable tipo puntero: `intPtr^`.

A continuación, hemos efectuado una operación aritmética con el puntero con el procedimiento `Inc(intPtr^, 3)`, que suma 3 al valor 127. Los punteros admiten el uso de los operadores + y -, pero siempre es preferible usar los procedimientos **`Inc()`** y **`Dec()`** para aumentar o disminuir el valor de la variable a la que apuntan.



**NOTA**
Para indicar que un puntero no apunta a ninguna dirección de memoria, se emplea la constante predefinida **`nil`**. Se trata de un modo alternativo para dar valor a un puntero. Lazarus siempre asigna el valor `nil` a las variables que representan objetos antes de que estos se creen.

## 3.6 UN PROGRAMA DE EJEMPLO

Este es un buen momento para emplear Lazarus y escribir un breve programa con el que poner de manifiesto los conocimientos fundamentales adquiridos a lo largo del capítulo.

Abra Lazarus y vaya al menú **Proyecto | Nuevo proyecto...** Seleccione **Programa** de la lista de la ventana de diálogo **Crear un proyecto nuevo** para crear una aplicación en modo consola, tal y como ya hizo antes. Desde el mismo menú **Proyecto**, elija **Guardar proyecto como...** y dele el nombre `tipos.lpr`.

Lazarus escribirá el siguiente código por usted:

```
program tipos;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

begin
end.
```

Borre todo el bloque **uses** (no va a ser necesario para este programa) y copie las siguientes líneas de código en el **Editor** para que su programa se parezca al que aquí se le muestra. Pulse **[F9]** para guardarlo, compilarlo y ejecutarlo:

```
program tipos;
{$mode objfpc}{$H+}
var
  enteroCorto: SmallInt= -25;
  inicio: boolean= False;

const
  Nombre= 'tipos';
  tab= #9; //tabulador
begin
  WriteLn('Este programa se llama ',tab,'"',Nombre,
         '"',sLineBreak);
  WriteLn('Ahora enteroCorto tiene el valor: ',
         enteroCorto);
  WriteLn('Y la variable inicio vale: ',inicio,
         sLineBreak);
  enteroCorto += 200; // sumamos 200 a enteroCorto
  WriteLn('Despues de sumar 200, enteroCorto vale: '
         ,enteroCorto);
  inicio:= not inicio; // ahora inicio es True
  WriteLn('La negacion de inicio lo convierte en: ',
         inicio, sLineBreak);
  WriteLn('El valor mas alto de SmallInt valdra: ',
         High(SmallInt));
  WriteLn('y el valor mas bajo sera: ',Low(SmallInt));
{$IFDEF WINDOWS}
  ReadLn;
{$ENDIF}
end.
```

**TRUCO**
Si necesita borrar rápidamente una línea de código en el **Editor**, puede usar el atajo de teclado **[Ctrl] + [Y]**.

Observe cómo hemos vuelto a usar la orden `WriteLn` para mostrar en la consola los valores de la constante y de las dos variables definidas, una de tipo `boolean` y otra de tipo `SmallInt`. Así mismo, hemos empleado la variable global **`sLineBreak`**, declarada en la unidad `System`, de modo que al escribir `WriteLn(sLineBreak)`, se inserta automáticamente una línea en blanco en la consola. El mismo efecto conseguiríamos escribiendo simplemente `WriteLn;`

La salida del programa se parecerá a esto:

```
Este programa se llama  "tipos"

Ahora enteroCorto tiene el valor: -25
Y la variable inicio vale: FALSE

Despues de sumar 200, enteroCorto vale: 175
La negacion de inicio lo convierte en: TRUE

El valor mas alto de SmallInt valdra: 32767
y el valor mas bajo sera: -32768
```

# 4

# TIPOS ESTRUCTURADOS DE DATOS

En el capítulo anterior introdujimos el concepto de tipo y vimos algunos básicos, como `char`, `integer`, `boolean`, `double`, etc. Lazarus, sin embargo, también puede trabajar con tipos estructurados, que combinan los anteriores para edificar bloques de mayor complejidad, acordes con la forma de construir objetos en la vida real. Por ejemplo, al conjunto de empleados de una fábrica podemos asociarle una estructura de datos con información como nombre del empleado, sueldo devengado, cargo, antigüedad, etc.

Lazarus brinda la posibilidad de definir diversas estructuras de datos de acuerdo con las necesidades propias del usuario a partir de los tipos estándar estudiados en el capítulo anterior.

En realidad, ya conoce tres tipos de datos estructurados: los tipos enumerados, los subrangos y el tipo cadena, pues Object Pascal considera a estos últimos como arreglos de caracteres.

En primer lugar, le presentaremos los arreglos como tipos estructurados de datos de un mismo tipo y continuaremos con los registros y conjuntos. Para finalizar, estudiaremos cómo trabajar con ficheros, fundamentalmente de texto.

## 4.1 ARREGLOS

Un **arreglo**, o *array* en inglés, es un tipo de dato estructurado que consta de una zona de almacenamiento continuo que contiene una serie de elementos del mismo tipo con un identificador común, eso a pesar de representar múltiples

elementos. Desde el punto de vista lógico, un arreglo se puede ver como un conjunto de elementos ordenados en fila, al que llamamos **vector**; o dispuestos en filas y columnas para formar una **matriz**. Cada elemento del arreglo se referencia por la posición que ocupa dentro del mismo. Dichas posiciones se denominan **índices** y siempre son correlativas (Figura 4.1).

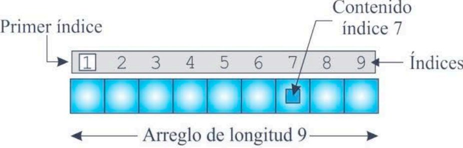



**Figura 4.1.** Arreglo unidimensional o vector de longitud 9

Los elementos del arreglo pueden ser de cualquiera de los tipos básicos estudiados, pero siempre todos ellos del mismo tipo.

Los arreglos pueden ser **estáticos**, en el sentido de que el programador establece la memoria asignada en el momento de compilar el programa, o **dinámicos**, cuando la memoria se va asignando en tiempo de ejecución según las necesidades.

### 4.1.1 Arreglos estáticos

Los arreglos estáticos se definen en Lazarus con la palabra reservada `array` más la dimensión del arreglo, según se ve a continuación:

```
type nombreArreglo = array[dimension] of tipoBase;
```

Donde `dimension` es el número de elementos del arreglo y `tipoBase` la naturaleza de los datos que contendrá. Por ejemplo, para declarar un vector de 5 elementos de tipo `shortInt` que se llame `TLista`, escribiríamos:

```
type TLista = array[1..5] of shortInt;
```

Los elementos individuales de un arreglo se referencian por su índice, según la dimensión declarada en aquel. De modo que si declaramos ahora la variable `Lista` de tipo `TLista`, podríamos escribir:

```
  var Lista : TLista;

begin
  Lista[1] := -12;
  Lista[2] := 14;
  Lista[3] := 3;
  Lista[4] := 0;
  Lista[5] := -121;
end;
```

El resultado es un vector que contiene como componentes los elementos asignados más arriba (Figura 4.2).

**Figura 4.2.** Arreglo de 5 elementos tipo shortInt

Cuando se crea la variable `Lista` no está vacía, simplemente se trata de una región de memoria reservada para que nuestro programa pueda usarla. Su contenido, hasta que no se inicialice, es indeterminado.

Si por error asigna un dato a un componente del arreglo que no existe, como ocurriría si hubiésemos escrito en el ejemplo anterior `Lista[6] := 4`, el programa hubiera compilado, pero FPC nos habría informado de ello con la advertencia,

```
Warning: range check error while evaluating constants
```

pues no reservamos espacio alguno en memoria para un sexto elemento. Los problemas podrían surgir en tiempo de ejecución y variarían desde un comportamiento inesperado hasta una excepción grave que cuelgue la aplicación.

Los arreglos pueden ser multidimensionales, como dijimos más arriba, para formar **matrices**. Cada una de las dimensiones se especifica con un índice diferente, separado del siguiente por una coma. Por ejemplo, si quisiera declarar una matriz 4 × 4 que contuviese solo caracteres, tendría que escribir:

```
type TLista = array[1..4, 1..4] of Char;
```

Cuando ahora cree una variable de tipo `TLista` reservará una región de memoria con la estructura mostrada en la figura 4.3.



**Figura 4.3.** Matriz 4 × 4 para 16 elementos tipo Char

La matriz declarada podrá contener 16 elementos tipo `Char`. Ahora tendremos un índice por cada dimensión. El primero referencia las filas y el segundo las columnas, tal y como se observa en la figura de más arriba.

Dentro de los arreglos estáticos, Free Pascal define un tipo estructurado de datos que se denomina **arreglo de constantes**. Se declara como una lista de valores separados por comas y entre paréntesis. Cada elemento de la lista ha de pertenecer al tipo básico declarado en el arreglo. Se emplean bastante a menudo para inicializar tablas.

Aunque a primera vista puede parecerle algo extraño este concepto, lo entenderá fácilmente con el siguiente ejemplo de los días de la semana:

```
type TDiasSemana = array[1..7] of string;
const diasSemana: TDiasSemana = ('Lunes', 'Martes',
        'Miércoles', 'Jueves', 'Viernes', 'Sábado',
        'Domingo');
```

En primer lugar, se declara un arreglo de 7 elementos de tipo cadena; a continuación, la lista de constantes de tipo cadena que se corresponde con los siete días de la semana. Ahora cada día puede referenciarse por su índice, de modo que `diasSemana[1]` es el `Lunes`, `diasSemana[2]` el `Martes`, y así sucesivamente.

### 4.1.2 Arreglos dinámicos

Los **arreglos dinámicos** son aquellos que pueden crecer o menguar conforme los elementos se agregan o se eliminan en tiempo de ejecución, por lo que no es necesaria la gestión de memoria. Se liberan automáticamente al finalizar el procedimiento, función o clase donde están alojados. También se emplean para enviar un número indeterminado de parámetros a funciones y procedimientos.

Para crear un arreglo dinámico, lo que hacemos es declararlo, pero sin especificar su tamaño:

```
type TClientes = array of string;
```

Antes de poder recoger elementos en el arreglo, hay que indicar su tamaño con la función **SetLength**. Por ejemplo:

```
  var clientes : TClientes;
begin
SetLength(clientes, 3);
clientes[0] := 'Irene';
clientes[1] := 'Miriam';
clientes[2] := 'Marta';
end;
```

Esto crea un arreglo de tres cadenas de valores: `Irene`, `Miriam` y `Marta`. La ventaja de hacerlo así y no con un arreglo estático es que en cualquier momento podemos expandir el arreglo si es necesario llamando de nuevo a la función `SetLength`.

A diferencia de los arreglos estáticos, el primer elemento es el cero y no el uno. Como hemos dicho antes, no es necesario liberar un arreglo dinámico de memoria, pues una vez finalizado el procedimiento, la función o la clase donde estaba alojado, se libera la memoria reservada. Si aun así desea asegurarse de que la memoria que el compilador reservó se libera inmediatamente, solo es necesario hacer lo siguiente:

```
clientes := nil;
```

## 4.2 CADENAS

Las cadenas, que ya las estudiamos en el capítulo anterior, son en realidad un tipo estructurado de datos. El tipo cadena, que Pascal denomina **shortString**, es en realidad un arreglo de caracteres de índice cero y longitud fija de 255 caracteres.

Lazarus usa el tipo `AnsiString` como tipo `String` por defecto mediante la directiva `{$H+}`. Estas cadenas se comportan como arreglos dinámicos de caracteres y pueden ser tan largas como sea necesario. Es más, el compilador maneja por nosotros la memoria que debe reservar en cada momento y automáticamente la libera cuando no sea necesaria. Además, el compilador de Free Pascal garantiza

que las cadenas tipo `AnsiString` están completamente vacías cuando se crean por primera vez. Todo esto ocurre de forma transparente, por lo que el programador solo tiene que preocuparse del código que debe escribir.

Por defecto, el **Editor** de Lazarus codifica todo el texto en el formato de longitud variable UTF-8, lo que significa que el programador puede trabajar de forma totalmente transparente con todos los caracteres alfabéticos europeos, ligaduras, símbolos matemáticos y de monedas, etc.

Una cadena de longitud variable se puede referenciar de forma exacta a como lo hacemos en los arreglos, es decir, si `cd` es de tipo `String`, entonces `cd[i]` es el i-ésimo carácter en `cd`. Evidentemente, el índice será positivo y no mayor que `Length(cd)`. Cada carácter en `cd` será de tipo `AnsiChar`.

Para habituarse a la forma de trabajar que nos imponen las cadenas, abra un nuevo proyecto de consola en Lazarus, como ya ha hecho antes, y guárdelo con el nombre **`cadenas`**. Modifique la cláusula **`uses`** para que incluya las unidades `strutils` y `sysutils`, añada una variable de tipo cadena de nombre `cd` y copie el código necesario para que su programa quede como el siguiente:

```
program cadenas;
{$mode objfpc}{$H+}
uses strutils, sysutils;
  var cd: string;

begin
  WriteLn('Escriba una palabra o frase y pulse [Enter]');
  readln(cd);     // lee la entrada del usuario
  WriteLn('Ha escrito: ', cd);
  WriteLn('En mayusculas es: ', UpperCase(cd));
  WriteLn('En minusculas: ', LowerCase(cd));
  WriteLn('Y escrito al reves: ', ReverseString(cd));
{$IFDEF WINDOWS}
ReadLn;
{$ENDIF}
end.
```

En este ejemplo hemos hecho uso de funciones que no se encuentran en la unidad `system`, por lo que explícitamente hemos tenido que indicar a Lazarus en la cláusula **`uses`** las unidades donde se encuentran aquellas. Las funciones `UpperCase()` y `LowerCase()` se definen en la unidad `sysutils`, mientras que `ReverseString()` lo hace en la unidad `strutils`.

Quizás ahora se esté preguntando cómo sabemos dónde se define una rutina o función para indicarlo en la cláusula **`uses`**. Bueno, si conoce el nombre de la función, es bastante sencillo saber en qué unidad se define. Escriba su nombre en el **Editor de código** de Lazarus, sitúe el cursor en cualquier lugar de la palabra y pulse la tecla **[F1]**. La ayuda del sistema le proporcionará la información sobre la función con un gran número de detalles, lo que incluye su unidad (Figura 4.4).

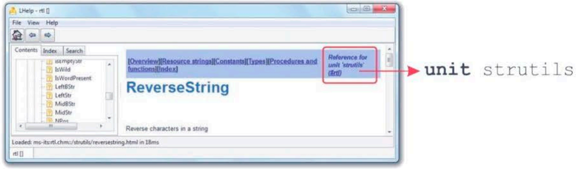



**Figura 4.4.** Ayuda para encontrar la unidad de una función con [F1]

A estas alturas ya se habrá dado cuenta de que en ninguno de los literales que estamos escribiendo con la rutina `WriteLn` ponemos tildes. Como hemos dicho, Lazarus codifica todo el texto en UTF-8, un formato muy extendido en el mundo de Unix y Linux, pero no así en Windows. Por ello, es necesario que para ver los caracteres correctamente en la consola de Windows, solo emplee el alfabeto anglosajón. Si, por el contrario, escribe el código para un sistema tipo Linux, podrá introducir las características tipográficas de nuestro idioma sin ningún problema.

## 4.3 REGISTROS

Un **registro** es una estructura formada por varios elementos constitutivos llamados **campos**. Los campos, ahora, pueden ser de diferentes tipos, lo que incluye números, caracteres, cadenas y arreglos en general, registros, etc.

Los registros se definen en Pascal con la palabra reservada **`record`**. Por ejemplo, podríamos declarar el siguiente registro `TCiudad`, que combina una cadena, `Nombre`; y un entero ordinal, `Poblacion`:

```
type TCiudad = record
  Nombre: string[30];
  Poblacion: longword;
  end;
```

Una vez definido el registro, se pueden declarar varias variables diferentes como registros de este tipo:

```
var Ciudad: TCiudad;
```

Lazarus también permite trabajar con **constantes** de **tipo registro**, que se escriben entre paréntesis con sus respectivos campos separados por punto y coma. Por ejemplo, una declaración válida de una constante tipo registro podría ser la siguiente:

```
const Madrid: TCiudad = (Nombre: 'Madrid'; Poblacion:
      3207247);
```

**IMPORTANTE**
Los registros no pueden contener como campos ficheros y, en la medida de lo posible, evite usar punteros o tipos basados en punteros, como las cadenas de tipo `AnsiString`.

Para acceder a los elementos individuales de un registro, se debe construir un **indicador** de campo. Este es la combinación del nombre de la variable de tipo registro, un punto y el nombre de un campo (`Registro.Campo`); por ejemplo:

```
Ciudad.Nombre
Ciudad.Poblacion
```

Escribamos ahora una pequeña aplicación para ver cómo se trabaja con los registros, y su enorme versatilidad. Abra un nuevo proyecto de consola en Lazarus, de nombre **`notas`**, y guárdelo en una nueva carpeta. Borre el contenido de la cláusula **`uses`** que Lazarus le presenta y teclee el código necesario para que su programa se parezca a esto:

```
program notas;

{$mode objfpc}{$H+}

uses strutils;
type
  TNotas = record
     Nombre: string[20];
     Media: byte;
```

```
end;

const Irene: TNotas = (Nombre: 'Irene'; Media: 63);
      Marta: TNotas = (Nombre: 'Marta'; Media: 61);
      Miriam: TNotas = (Nombre: 'Miriam'; Media: 54);
      David: TNotas = (Nombre: 'David'; Media: 81);

procedure MostrarInfo(aNombre: TNotas);
var longitud: integer;
begin
     longitud:= aNombre.Media div 7;
     WriteLn(aNombre.Nombre:7, aNombre.Media:6, '   ',
       DupeString('*',longitud));
end;
begin
     WriteLn(' Nombre  Media  Media Relativa');
     WriteLn(' ------  -----  --------------');
     MostrarInfo(Irene);
     MostrarInfo(Marta);
     MostrarInfo(Miriam);
     MostrarInfo(David);
     {$IFDEF WINDOWS}
     ReadLn;
     {$ENDIF}
end.
```

Estudiemos un poco el código, pues hay algunos detalles interesantes. En primer lugar, hemos declarado un registro de nombre `TNotas` y cuatro constantes de tipo `registro`, cada una con el nombre y nota media de un alumno. A continuación, hemos escrito un procedimiento (**`procedure`**) para mostrar la información de cada constante. El único dato que necesita este procedimiento es el parámetro `aNombre` de tipo `TNotas`.

Para mostrar los datos por pantalla, hemos hecho uso de una característica interesante de `WriteLn`, la posibilidad de escribir datos en campos de anchura `N`, para lo que se usa el especificador `:N` tras los datos. Así, mostramos en la consola el nombre de los alumnos en un campo de 7 caracteres de anchura, y su media en uno de 6. Para finalizar, usamos la función `DupeString`, que se define en la unidad `strutils`, para formar una cadena de caracteres duplicados (en nuestro caso, asteriscos). Con ellos indicaremos visualmente las medias relativas; lo que hemos hecho con una simple división entera del campo `Media`. Si compila y ejecuta el programa, obtendrá una salida semejante a lo siguiente:

```
end;

const Irene: TNotas = (Nombre: 'Irene'; Media: 63);
      Marta: TNotas = (Nombre: 'Marta'; Media: 61);
      Miriam: TNotas = (Nombre: 'Miriam'; Media: 54);
      David: TNotas = (Nombre: 'David'; Media: 81);

procedure MostrarInfo(aNombre: TNotas);
var longitud: integer;
begin
     longitud:= aNombre.Media div 7;
     WriteLn(aNombre.Nombre:7, aNombre.Media:6, '   ',
       DupeString('*',longitud));
end;
begin
     WriteLn(' Nombre  Media  Media Relativa');
     WriteLn(' ------  -----  --------------');
     MostrarInfo(Irene);
     MostrarInfo(Marta);
     MostrarInfo(Miriam);
     MostrarInfo(David);
     {$IFDEF WINDOWS}
     ReadLn;
     {$ENDIF}
end.
```

Estudiemos un poco el código, pues hay algunos detalles interesantes. En primer lugar, hemos declarado un registro de nombre `TNotas` y cuatro constantes de tipo `registro`, cada una con el nombre y nota media de un alumno. A continuación, hemos escrito un procedimiento (**`procedure`**) para mostrar la información de cada constante. El único dato que necesita este procedimiento es el parámetro `aNombre` de tipo `TNotas`.

Para mostrar los datos por pantalla, hemos hecho uso de una característica interesante de `WriteLn`, la posibilidad de escribir datos en campos de anchura `N`, para lo que se usa el especificador `:N` tras los datos. Así, mostramos en la consola el nombre de los alumnos en un campo de 7 caracteres de anchura, y su media en uno de 6. Para finalizar, usamos la función `DupeString`, que se define en la unidad `strutils`, para formar una cadena de caracteres duplicados (en nuestro caso, asteriscos). Con ellos indicaremos visualmente las medias relativas; lo que hemos hecho con una simple división entera del campo `Media`. Si compila y ejecuta el programa, obtendrá una salida semejante a lo siguiente:

```
Nombre  Media  Media Relativa
------  -----  --------------
 Irene    63   *******
 Marta    61   ******
Miriam    54   *****
 David    81   ***********
```

### 4.3.1 La sentencia *with...do*

Si observa con calma las líneas de código anteriores, se dará cuenta de que la referencia a registros se hace un poco monótona cuando se tienen que indicar varios campos, porque se han de repetir los nombres del registro cada vez que se hace referencia a un campo del mismo. En estos casos, es posible utilizar la sentencia **with... do**, que permite omitir el nombre de los registros en los indicadores de campo.

La sentencia completa consta de la palabra reservada **with**, seguida de una lista de variables de tipo registro y separadas por comas, para terminar con otra palabra reservada, **do**. Enseguida se abrirá el bloque en el que irán las instrucciones sobre las que actúa.

Por ejemplo, el procedimiento `MostrarInfo` que hemos hecho para el ejemplo anterior:

```
procedure MostrarInfo(aNombre: TNotas);
var longitud: integer;
begin
     longitud:= aNombre.Media div 7;
     WriteLn(aNombre.Nombre:7, aNombre.Media:6, '    ',
       DupeString('*',longitud));
end;
```

Podríamos haberlo escrito así con la sentencia **with...do**:

```
procedure MostrarInfo(aNombre: TNotas);
var longitud: integer;
begin
  with aNombre do
    begin
      longitud:= Media div 7;
      WriteLn(Nombre:7, Media:6, '   ', DupeString(
        '*',longitud));
    end;
end;
```

Desde el punto de vista del compilador, ambos procedimientos son idénticos; sin embargo, para el programador o el analista, el seguimiento del flujo de programa puede a veces complicarse bastante al emplear sentencias **with...do**. Así que el consejo es que use aquello que le resulte más fácil.

## 4.4 CONJUNTOS

En general, un **conjunto** es una colección ordenada de datos simples del mismo tipo, a los que llamamos **elementos**. Generalmente sus elementos son de tipo ordinal, a menudo de tipo enumerado o subrango.

Con el fin de utilizar esta nueva estructura de datos, primero se debe definir un tipo conjunto para declarar, a continuación, las variables de tipo conjunto. De hecho, una variable conjunto puede representar cualquier número de elementos del conjunto, incluido ninguno. Esta característica nos ofrece una manera sencilla de determinar si una entidad o evento está dentro de una o más categorías predefinidas.

En Pascal orientado a objetos, los elementos son datos de algún tipo previamente declarado. Son ejemplos de conjuntos los números enteros del 1 al 100, las letras del alfabeto, las vocales o las consonantes, etc.

En Lazarus se declaran los conjuntos con las palabras reservadas **set of**:

```
type TConjunto = set of TTipoBase;
```

Para construir un conjunto, se deben colocar los elementos que lo forman entre corchetes y separados por comas; por ejemplo:

```
type TLetras = set of Char;
const vocales: TLetras = ['a', 'e', 'i', 'o', 'u'];
```

Los conjuntos, en Pascal, están bastante limitados en tamaño, pues no pueden contener más de 256 elementos, con ordinales comprendidos entre 0 y 255. Los elementos en un conjunto no ocupan una posición determinada en él, solo se puede decir si pertenecen o no al conjunto.

Las **operaciones** que se pueden realizar son las que normalmente se utilizan con los conjuntos matemáticos, es decir: unión, intersección, diferencia, igualdad, desigualdad, inclusión y pertenencia. Pasemos ahora a ver la descripción y efecto de cada una de ellas:

- **Unión de conjuntos**. Se expresa con el signo +, y su resultado es el conjunto formado por todos los elementos que pertenecen al menos a uno de los conjuntos dados:

  ['a'..'c'] + ['b'..'d'] → ['a','b','c','d']

- **Intersección**. Se expresa con el signo * y su resultado es el conjunto formado por los elementos comunes a todos los conjuntos dados:

  ['a'..'c'] * ['b'..'f'] → ['b','c']

- **Diferencia de conjuntos**. Se expresa con el signo **–** y su resultado es el conjunto formado por los elementos que pertenecen al primer conjunto y no al segundo:

  ['a'..'c'] * ['b'..'f'] → ['a']

- **Igualdad**. Dos conjuntos son iguales si y solo si sus elementos son los mismos, sin importar el orden. Se expresa con el signo **=** y el resultado será `True` o `False`.

- **Desigualdad**. Dos conjuntos son desiguales si y solo si uno de los elementos es diferente en ambos conjuntos. La desigualdad se expresa con el operador **<>**. Su resultado, como en el caso anterior, será `True` o `False`.

- **Inclusión de conjuntos**. Se emplean los símbolos **<=** para indicar la relación de inclusión "contenido en", y **>=** para la relación de inclusión "contiene a".

  ['a'..'z'] >= ['b','h'] → True

- **Pertenencia**. Se utiliza para saber si un elemento pertenece a un conjunto. Para ello se emplea la palabra reservada **`in`**:

  'c' **in** ['a'..'d'] → True

## 4.5 FICHEROS

Los ficheros son los elementos que emplea un ordenador para almacenar físicamente información en el disco duro o en cualquier otra unidad de almacenamiento. Es normal que un programa genere información que se deba almacenar para ser posteriormente procesada. Por ello, cualquier lenguaje de programación debe ser capaz de manejar ficheros: abrir, insertar elementos, leerlos, borrar ficheros y cerrarlos.

Un **fichero** es una secuencia binaria de un determinado **tipo base**, que en Pascal no puede ser ni de tipo puntero, ni un arreglo dinámico, ni una cadena `ansistring`.

Lazarus permite manejar dos tipos de ficheros, los de tipo binario y los de texto. Un **fichero** de tipo **binario** se basa en un tipo estándar, un arreglo estático o un registro, aunque el compilador, internamente, interprete los ficheros como registros. Los **ficheros** de **texto**, por el contrario, no se basan en un tipo base predefinido, sino en líneas de texto de longitud variable. Pascal no permite accesos aleatorios a líneas individuales de texto, estas deben leerse de forma secuencial, como veremos en el último ejemplo del capítulo.

Por lo que respecta a los ficheros binarios, hay dos modos de declararlos, según sean o no tipados.

Los ficheros **binarios tipados** se declaran con las palabras reservadas **`file of`** más el tipo base, que suele ser `byte` o `record`.

La variable global de sistema **`FileMode`** especifica cómo se abre el fichero: como solo lectura (0), solo escritura (1), o para operaciones de lectoescritura (2), que es su valor por defecto.

En cuanto a los ficheros **no tipados**, se declaran solo con la palabra reservada **`file`**, sin ningún tipo base en la declaración. El compilador los tratará como secuencias de registros con un tamaño fijo de 128 bytes.

Las **operaciones básicas** que se pueden realizar con los ficheros son las siguientes:

- **Asignación**. Consiste en especificar el nombre que va a tener el fichero que vamos a utilizar, tanto en el programa como en el sistema operativo. La sentencia es la siguiente:

```
AssignFile (nombre_interno, 'nombre_externo');
```

- **Lectura**. Cada vez que se quiera leer un dato almacenado en un fichero, será necesario realizar una operación de lectura, que se hace con las siguientes instrucciones:

  - **`Reset`**. Es la instrucción que procesa un fichero para su lectura, de forma que un elemento apunta a la primera posición del fichero. La sentencia es la siguiente:

    ```
    Reset (nombre_interno);
    ```

  - **`Read`**. Con esta instrucción se lee desde el programa un dato contenido en la posición que se indique. Una vez que se ha realizado una lectura del fichero, el apuntador se desplaza al siguiente componente del fichero. Si se vuelve a realizar otra lectura se leerá el dato que está a continuación. Se trata, por tanto, de accesos secuenciales. La sentencia es la siguiente:

    ```
    Read (nombre_interno, dato);
    ```

    Para ficheros binarios no tipados, `Read` debe sustituirse por **`BlockRead`**.

- **Escritura**. No solo es necesario leer ficheros, también lo es poder escribir valores en un fichero propio o ajeno al programa. La instrucción que se emplea en este caso es **`Write`**. Con esta se escribe de forma secuencial información en un fichero previamente abierto. Si, por el contrario, el fichero es nuevo, entonces habrá que usar la orden **`ReWrite`**. Después de ejecutarse una sentencia `write`, el apuntador del fichero señalará a la siguiente posición del fichero. Su sintaxis es:

  ```
  Write (nombre_interno, variable);
  ```

  Para escribir en ficheros binarios no tipados, `Write` debe sustituirse por **`BlockWrite`**.

- **Fin de fichero**. La función **`EOF`** (*End Of File*) predefinida en Pascal devuelve el valor `True` si se ha alcanzado el fin de fichero.

- **Cierre de fichero**. La instrucción **`Close`** insta a que el archivo se cierre y se guarde en disco, liberando de este modo la memoria RAM ocupada por el fichero. Todo fichero abierto por un programa debe ser obligatoriamente cerrado por el mismo. Si por error se deja abierto, puede provocar errores y conflictos en el sistema operativo.

En cuanto a los **ficheros de texto**, no están formados por una secuencia repetida de registros binarios, sino que solo contienen líneas de texto de cualquier longitud.

Se declaran de la siguiente manera:

```
var nombre_interno: TextFile;
```

A continuación vamos a escribir una pequeña aplicación de consola para mostrarle cómo acceder fácilmente al contenido de un fichero de texto. Abra en Lazarus un proyecto y guárdelo con el nombre **fichero_texto.lpr**. Adapte el contenido de la plantilla de código para que se parezca a esto:

```
program fichero_texto;
{$mode objfpc}{$H+}
uses sysutils;
var txtF: TextFile;
  cd: String;

procedure Leer15Lineas;
const Lineas: integer = 0;
var lineasLeidas: integer = 0;
begin
  while not EOF(txtF) and (lineasLeidas < 15) do
    begin
      ReadLn(txtF, cd);
      Inc(lineasLeidas); // Incrementamos el ordinal
      Inc(Lineas);
      cd:= Format('Linea %d: %s',[Lineas, cd]);
      WriteLn(cd);
    end;
end;

begin
  AssignFile(txtF, 'fichero_texto.lps');
  Reset(txtF); // Apunta primera posición fichero
  WriteLn('Las lineas de fichero_texto.lps se
         mostraran de 15 en 15');
  WriteLn;
  try   // Código expuesto a posibles errores
    while not EOF(txtF) do
    begin
      Leer15Lineas;
      WriteLn;
      WriteLn('Pulse [Enter] para continuar');
      ReadLn;
    end;
finally  // Se libera el recurso. Siempre se ejecuta
  CloseFile(txtF);
end;
Write('Fin del fichero. Pulse [Enter] para acabar');
ReadLn;
end.
```

Cuando acabe, compile y ejecute el programa. Verá cómo se le mostrará en la consola el contenido del fichero indicado de 15 en 15 líneas.

Aunque no se han explicado aún todas las sentencias y bloques que aparecen en el programa, su contenido no es difícil de entender. Fíjese, sobre todo, en las instrucciones que hemos escrito para poder trabajar con el contenido del fichero: `AssignFile`, `Reset`, `Read` y `CloseFile`. Así mismo, es interesante comentar el funcionamiento de la función **`Format`**`()`, descrita en la unidad `sysutils`, que nos facilita de forma muy precisa el control del formato de los datos que se muestran en una cadena, al estilo de como se hace en el lenguaje C. El control se realiza a través de los parámetros que se le pasan, que comienzan con el símbolo `%` y son los siguientes:

- `%d`: decimal entero.
- `%e`: notación científica.
- `%m`: moneda.
- `%n`: número en coma flotante.
- `%p`: puntero.
- `%s`: cadena.
- `%u`: decimal sin signo.
- `%x`: hexadecimal.

De modo que en la sentencia,

```
Format ('Linea %d: %s',[Lineas, cd]);
```

`%d` y `%s` se reemplazan con el formato indicado por los valores que contienen las variables `Lineas` y `cd`, respectivamente. Por ejemplo, si se define una variable de nombre `unaCadena` del siguiente modo:

```
unaCadena := Format('El valor hexadecimal de %d es
                %x', [14,14]);
```

Tendrá, desde ese momento, el contenido: “El valor hexadecimal de 14 es E”. Se trata, por tanto, de una manera rápida y útil de introducir variables en el lugar adecuado de una cadena y con el formato correcto.

# 5

# OPERADORES Y EXPRESIONES

Un **operador** es un símbolo especial que indica al compilador que debe efectuar una operación matemática o lógica. En un programa, el tipo de un dato determina las operaciones que se pueden ejecutar con él. Por ejemplo, con los datos de tipo entero se pueden realizar operaciones aritméticas, tales como la suma (+), la resta (−) o la multiplicación (*).

Cuando se combinan uno o más operadores con uno o más operandos se obtiene una expresión. De modo que una **expresión** es una secuencia de operandos y operadores escrita bajo unas determinadas reglas sintácticas. Un operador siempre forma parte de una expresión, en la cual el operador siempre actúa sobre al menos un operando. Por el contrario, un operando sí puede aparecer solo en una expresión. Los operadores que requieren un operando son llamados **operadores unarios**, mientras que los que actúan sobre dos operandos se denominan **operadores binarios**.

La lista de operadores es bastante larga. A los operadores aritméticos tan familiares como `+`, `–`, `*` y `/` se les unen otros menos conocidos, como >> y <>; y otros aún más extraños, como **`mod`**, **`div`**, **`not`** o **`xor`**, entre otros. La mayoría solo son aplicables a un pequeño número de tipos, aunque, como en otros lenguajes de programación orientados a objetos, ampliamos las capacidades del lenguaje con el mecanismo de la **sobrecarga de operadores**. Este mecanismo permite que algunos operadores puedan ser aplicables a distintos tipos, como ocurre con el operador suma (+), que puede operar tanto con números, para sumarlos, como con caracteres y cadenas para concatenarlos.

Los operadores en Lazarus se pueden dividir en las siguientes cuatro categorías:

- De asignación.
- Aritméticos.
- Lógicos y de bit.
- Relacionales.

Como verá a lo largo del capítulo y la obra, en Pascal orientado a objetos los operadores son muy potentes y versátiles, y, por si eso fuera poco, aún nos permite ir más allá y emplear operadores definidos por el usuario. Los operadores de asignación ya los estudiamos en el Capítulo 3, así que comencemos con los operadores aritméticos.

## 5.1 OPERADORES ARITMÉTICOS

Lazarus soporta varios operadores aritméticos para los números enteros y en coma flotante. Se incluyen + (suma), - (resta), * (multiplicación), / (división), ** (exponenciación), **div** (división entera) y **mod** (módulo).

En la tabla siguiente se recogen los operadores que pueden aplicarse a los valores numéricos.

| Operador | Operación | Operandos |
|---|---|---|
| + | Suma/Signo | Enteros, reales |
| - | Resta/Signo | Enteros, reales |
| * | Producto | Enteros, reales |
| / | División real | Enteros, reales |
| div | División entera | Enteros |
| mod | Módulo | Enteros |
| ** | Exponenciación | Enteros, reales |

**Tabla 5.1.** Operadores aritméticos definidos para Lazarus

Como se observa en la tabla, los operadores + y - tienen versiones unarias que se aplican sobre un operando para definir el signo o invertirlo, respectivamente.

El tipo de los datos devueltos por una operación aritmética depende del tipo de sus operandos. Si se suman dos enteros, se obtiene un entero como tipo devuelto con el valor de la suma de los dos enteros.

El lenguaje Pascal orientado a objetos **sobrecarga** la definición del operador + para incluir la concatenación de cadenas o caracteres. El siguiente ejemplo utiliza el operador + para concatenar la cadena "*Hola*" con la cadena " *mundo.*":

```
NuevaCadena :=  'Hola' + ' mundo.'
```

En una expresión aritmética puede aparecer más de un operador aritmético. En ese caso, para poder evaluar correctamente la expresión es necesario seguir un criterio de prioridad de operadores. Cuando aparecen juntos operadores con la misma prioridad, estos se evalúan de izquierda a derecha.

En la tabla 5.2 se refleja el orden de prelación de los operadores aritméticos definidos en Pascal orientado a objetos:

| Operador | Operación |
|---|---|
| * / div mod | Producto, cociente, división entera y módulo |
| + - | Suma y resta |

**Tabla 5.2.** Orden de prelación de los operadores aritméticos

Si se desea modificar la prioridad de los operadores en las expresiones, se debe hacer uso de los paréntesis, del mismo modo que los usamos en el álgebra.

## 5.2 OPERADORES LÓGICOS

Los operadores lógicos son aquellos que actúan sobre una o dos variables de tipo booleano para dar como resultado una cierta condición: `True` o `False`.

En Lazarus trabajamos con cuatro operadores lógicos: **`and`**, **`or`**, **`xor`** y **`not`**. El operador **not** es unario y actúa sobre un único operando booleano negando su valor; de modo que **not** `True` es `False` y **not** `False` es `True`. El resto de operadores son binarios.

En la tabla 5.3 se recoge el resultado de aplicar los tres operadores booleanos binarios a las distintas combinaciones de operandos.

| **Operador** | | **and** | **or** | **xor** |
|---|---|---|---|---|
| Operación | | Conjunción | Disyunción | Disyunción excluyente |
| **A** | **B** | A **and** B | A **or** B | A **xor** B |
| **False** | **False** | False | False | False |
| **False** | **True** | False | True | True |
| **True** | **False** | False | True | True |
| **True** | **True** | True | True | False |

**Tabla 5.3.** Tablas de verdad de las operaciones lógicas binarias

Lazarus también permite trabajar en bits. Incluso aquellos tipos que menos espacio ocupan en memoria son de ocho bits, de modo que el programador podrá modificar bit a bit el contenido de cualquier valor. Existen seis operadores que podrá emplear para este cometido: **not, and, or, xor, shr** y **shl.**

El operador **not** es unario, como ya sabe, de modo que su actuación invertirá los dígitos binarios sobre los que actúe. Cualquier bit 0 se convertirá en 1 y cualquier bit 1 lo hará en 0. El resto de operadores de bit son binarios. Las tablas de verdad de **and, or** y **xor** son idénticas a las mostradas en la tabla 5.3. Basta suponer que False es el bit 0 y True el bit 1.

Tenga en cuenta que trabajar en bits, cuando no pensamos así, puede provocar resultados bastante extraños, aunque correctos. Por ejemplo, si efectuamos a escala de bit la operación 12 **and** 25, el resultado que obtendrá será 8. ¿Cómo es posible? Al convertir a binario ambos operandos y operar en bits con el operador **and**, se obtiene:

```
    12 = 0000 1100
and 25 = 0001 1001
---------------------
     8 = 0000 1000
```

Un resultado poco obvio, pero correcto.

Existen dos operadores adicionales para trabajar en bits, los operadores de **desplazamiento lógico**. En el desplazamiento lógico los bits de un registro se mueven una o más posiciones hacia la derecha o hacia la izquierda, por lo que hablamos de un desplazamiento lógico hacia la **izquierda**, para el que empleamos en Lazarus la

función **shl** o el operador <<; o de un desplazamiento lógico hacia la **derecha**, para el que utilizamos la función **shr** o el operador >>. Los bits que salen del registro por un extremo se pierden y el otro extremo del registro se rellena con un bit cero por cada bit desplazado.

Por ejemplo. Si se tiene un registro de 8 bits de valor `0001 0111`, y se hace un desplazamiento hacia la izquierda de un bit, todos los bits se mueven una posición hacia la izquierda, el bit de la izquierda se pierde y entra un bit cero de relleno por el lado derecho (Figura 5.1). En un desplazamiento de un bit hacia la derecha ocurre algo análogo, el bit de la derecha se pierde y el de la izquierda se rellena con un cero (Figura 5.1):

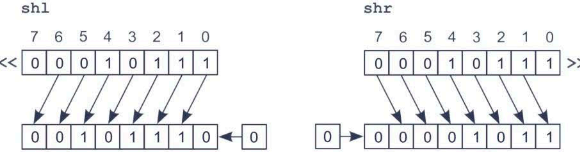



**Figura 5.1.** Desplazamientos lógicos de un bit hacia la izquierda (shl) y derecha (shr)

De modo que el registro original de valor decimal `23` (`0001 0111`) se ha transformado es dos valores numéricos diferentes al aplicar el desplazamiento de un bit:

```
23 << 1 = 0010 1110 = 46    // 1 bit izquierda
23 >> 1 = 0000 1011 = 11    // 1 bit derecha
```

En una expresión lógica pueden aparecer uno o más operadores lógicos, por lo que para poder evaluar correctamente estas expresiones es necesario seguir un criterio de prioridad de operadores. En Pascal orientado a objetos, el orden de prelación de los seis operadores lógicos estudiados queda recogido en la tabla 5.4. Cuando aparezcan dos o más operadores con la misma prioridad, se evaluarán de izquierda a derecha.

| Operador | Operación |
|---|---|
| `not` | Negación |
| `and shl shr` | Conjunción, desplazamiento a la izquierda y desplazamiento a la derecha |
| `or xor` | Disyunción y disyunción excluyente |

**Tabla 5.4.** Orden de prelación de los operadores lógicos en Lazarus

## 5.3 OPERADORES RELACIONALES

Un operador **relacional** se utiliza para comparar los valores de dos expresiones ordinales. Estas deben ser de tipos compatibles, excepto para enteros y reales, que pueden compararse sin problemas. El resultado de la comparación será de tipo booleano.

Las comparaciones entre los valores de tipo numérico son obvias. En lo que respecta a los valores de tipo carácter, sin embargo, su orden viene dado por el valor numérico del código ASCII extendido utilizado por el ordenador para representarlos.

En la tabla siguiente se recogen los operadores relacionales que se pueden emplear en Pascal orientado a objetos:

| Operador | Operación |
|---|---|
| = | Igualdad |
| <> | Desigualdad |
| > | Estrictamente mayor que |
| < | Estrictamente menor que |
| >= | Mayor o igual que |
| <= | Menor o igual que |

**Tabla 5.5.** Operadores relacionales aplicables a los tipos ordinales

Todos los operadores relaciones tienen la misma prioridad en las expresiones en las que aparecen, por lo que se evaluarán siempre de izquierda a derecha.

## 5.4 PRECEDENCIA DE OPERADORES

Cuando en una expresión aparecen varios operadores, el compilador deberá elegir en qué orden los aplica. A este orden se le llama *precedencia*.

Los operadores con mayor precedencia son evaluados antes que los operadores con una precedencia relativa menor.

Cuando en una sentencia aparecen operadores con la misma precedencia:

- Los operadores de asignación se evalúan de derecha a izquierda.
- Los operadores binarios, menos los de asignación, se evalúan de izquierda a derecha.

Se puede indicar explícitamente al compilador de Free Pascal que evalúe en otro orden los operadores mediante el uso de paréntesis. Para hacer que el código sea más fácil de leer y mantener, es preferible ser explícito e indicar con paréntesis qué operadores deben ser evaluados primero.

La siguiente tabla muestra la precedencia asignada a los operadores. Los operadores están listados en orden de precedencia: cuanto más arriba aparezca un operador, mayor es su precedencia. Los operadores en la misma fila tienen la misma prioridad:

| Operador | Operación |
|---|---|
| `@ not` | Puntero y negación |
| `* / div mod and shl shl` | Producto, cociente, división entera, módulo y conjunción |
| `+ - or xor` | Suma, resta, disyunción y disyunción exclusiva |
| `< <= > >= = <>` | Menor que, menor o igual, mayor que, mayor o igual, igual que y distinto |

**Tabla 5.6.** Precedencia de operadores en Lazarus

Por ejemplo, la siguiente expresión produce un resultado diferente dependiendo de si se realiza primero la suma o división:

```
a + b / 25
```

Si no se le indica explícitamente al compilador el orden en que queremos realizar las operaciones, entonces este decide basándose en la precedencia asignada a los operadores. Como el operador de división tiene mayor prioridad que el operador de suma, el compilador evaluará `b / 25` en primer lugar.

Si se quisiera cambiar el orden, habría que indicarlo explícitamente con el uso de paréntesis:

```
(a + b) / 25
```

Ahora el compilador evaluaría primero la suma y después el cociente. Además, los paréntesis se pueden anidar, es decir, es posible escribir unos dentro de otros, priorizándose del más interno al más externo y, después, de izquierda a derecha.

## 5.5 UN PROGRAMA DE EJEMPLO

Abra un nuevo proyecto en Lazarus en modo consola y dele el nombre **operadores**. Borre la cláusula **uses** y copie el código fuente siguiente:

```
program operadores;
{$mode objfpc}{$H+}
var a: byte = 25;
    b: byte = 5;
begin
 writeln ('El valor inicial de a es ', a,' y el de b '
         , b, sLineBreak);
 writeln ('El cociente entero de a/b es ', a div b, '
         y el resto ', a mod b, sLineBreak);
 a:= a shr 1;
 b:= b shl 3;
 writeln ('Despues de a >> 1, a es ', a,'; tras b << 3
         , b vale ', b, sLineBreak);
 writeln ('La negacion de a es ', not a, ' y la de b '
         , not b, sLineBreak);
 writeln ('a xor b = ', a xor b,', a and b = ', a and
         b, ', a or b = ', a or b, sLineBreak);
 writeln ('a > b es ', a>b,', a < b es ', a <b, ', a
         <> b es ', a <>b, ', a = b es ', a = b);
 {$IFDEF WINDOWS}
 ReadLn;
 {$ENDIF}
end.
```

Lo interesante de este programa sería que antes de ejecutarlo pudiera predecir los valores que se mostrarán en la salida estándar (la pantalla). Si no puede, no se preocupe. Intente comprender el funcionamiento de los operadores. Ese es el único objetivo que se persigue en este capítulo.

# 6

# SENTENCIAS DE CONTROL

La programación estructurada se basa en la utilización de un reducido número de estructuras, lo que permite que un programa sea suficientemente legible. Además, se consigue reducir considerablemente el número de errores y se facilita en gran medida la detección y solución de estos. La característica fundamental del paradigma de la programación estructurada es que todas las estructuras tienen un único punto de entrada y un único punto de salida, lo que permite descomponer fácilmente un problema en partes más pequeñas, reducir la complejidad y facilitar su programación.

Hasta este momento solo se ha usado el flujo secuencial. Cada una de las sentencias que se utiliza en Pascal está separada casi siempre de la siguiente por un punto y coma. No obstante, en algunos casos, nos interesará agrupar en un bloque una serie de sentencias, como veremos al explicar las estructuras de selección y de iteración. El bloque de sentencias se define por las palabras reservadas **`begin`**, para marcar el inicio del mismo, y **`end`** para indicar el final.

Los pocos programas que hemos hecho hasta este punto se ejecutan de modo secuencial, es decir, una sentencia después de otra. La ejecución comienza con la primera expresión del programa y prosigue hasta la última sentencia, cada una de las cuales se ejecuta una sola vez. Esta forma de programación es adecuada para programas sencillos; sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se ejecutan y en qué momento. Para poder modificar el flujo secuencial de la ejecución, Pascal dispone de varias sentencias, de las cuales las más utilizadas se pueden agrupar en tres familias: las **estructuras** de **selección** o condicionales, las iterativas o **repetitivas** y las de **excepción**.

# 6

# SENTENCIAS DE CONTROL

La programación estructurada se basa en la utilización de un reducido número de estructuras, lo que permite que un programa sea suficientemente legible. Además, se consigue reducir considerablemente el número de errores y se facilita en gran medida la detección y solución de estos. La característica fundamental del paradigma de la programación estructurada es que todas las estructuras tienen un único punto de entrada y un único punto de salida, lo que permite descomponer fácilmente un problema en partes más pequeñas, reducir la complejidad y facilitar su programación.

Hasta este momento solo se ha usado el flujo secuencial. Cada una de las sentencias que se utiliza en Pascal está separada casi siempre de la siguiente por un punto y coma. No obstante, en algunos casos, nos interesará agrupar en un bloque una serie de sentencias, como veremos al explicar las estructuras de selección y de iteración. El bloque de sentencias se define por las palabras reservadas **`begin`**, para marcar el inicio del mismo, y **`end`** para indicar el final.

Los pocos programas que hemos hecho hasta este punto se ejecutan de modo secuencial, es decir, una sentencia después de otra. La ejecución comienza con la primera expresión del programa y prosigue hasta la última sentencia, cada una de las cuales se ejecuta una sola vez. Esta forma de programación es adecuada para programas sencillos; sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se ejecutan y en qué momento. Para poder modificar el flujo secuencial de la ejecución, Pascal dispone de varias sentencias, de las cuales las más utilizadas se pueden agrupar en tres familias: las **estructuras** de **selección** o condicionales, las iterativas o **repetitivas** y las de **excepción**.

## 6.1 ESTRUCTURAS DE SELECCIÓN

Las estructuras de selección o condicionales controlan si se ejecuta una expresión o secuencia de expresiones, en función del cumplimiento o no de una condición o expresión lógica. Pascal orientado a objetos tiene dos estructuras de control para la selección, `if` y `case of`.

### 6.1.1 Sentencia *if*

El orden en el que se ejecutan las expresiones en un programa se denomina en programación **control de flujo**. La CPU, mientras no se le diga lo contrario, procesa las sentencias en el orden en que se indican en el código fuente del programa (Figura 6.1 A). Para que el microprocesador pueda elegir entre diferentes rutas de ejecución, deberemos usar una **estructura de control** que le fuerce a seleccionar una u otra alternativa en función del resultado de la evaluación de una cierta condición (Figura 6.1 B).

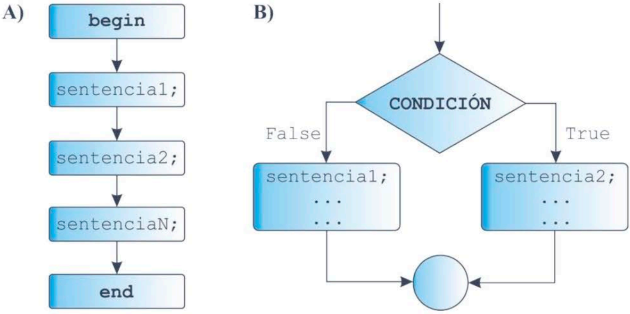



**Figura 6.1.** Flujo de programa. A) Secuencial. B) Estructura de selección

Como se observa en la figura 6.1 B, el flujo se interrumpe por una condición. Si la evaluación resulta `True`, se ejecutará `sentencia2`, mientras que si el resultado es `False`, lo hará `sentencia1`. La sintaxis de esta estructura condicional en Lazarus sería:

```
if (expresion booleana)then    // si … entonces
  sentencia2
else                           // en caso contrario
  sentencia1;
```

El ordenador elige solo una de las dos posibles alternativas, en función de la expresión booleana que se analiza. La siguiente es una estructura de selección válida:

```
if Hoy in [Lunes..Viernes] then
  WriteLn ('Debes trabajar duro')
else
  WriteLn ('Haz una escapada.');
```

**IMPORTANTE**
Observe que en una estructura de selección **if** no aparece el punto y coma final en las sentencias hasta la última de ellas.

Esta estructura condicional admite **anidaciones**; es decir, pueden incluirse unas sentencias **if** dentro de otras. Por ejemplo, sería válido:

```
if dia = 1 then
   writeln('Lunes')
else if dia = 2 then
     writeln('Martes')
     else if dia = 3 then
          writeln('Miercoles')
          else if dia = 4 then
               writeln('Jueves')
               else if dia = 5 then
                    writeln('Viernes');
```

Sin embargo, aunque correcto, el resultado es bastante farragoso, sobre todo a medida que aumenta el número de instrucciones anidadas. El sangrado con el que escribimos el código facilita la legibilidad, pero, aun así, el flujo de programa puede resultar bastante difícil de seguir, y el código de mantener.

En los casos en los que se necesite probar varias condiciones, es mejor emplear la estructura de selección **case of**.

### 6.1.2 Estructura *case of*

La sentencia **case** es una sentencia de Pascal que se utiliza para seleccionar una de entre múltiples alternativas. Es especialmente útil cuando la selección se basa en el valor de una variable o expresión de un tipo simple denominada expresión de control o **selector**.

La expresión del **selector** debe ser un tipo **ordinal**, excepto booleano, o **cadena**. El tipo de cada etiqueta debe ser el mismo que el de la expresión de control.

Cuando se ejecuta la sentencia **case** se evalúa el selector. Si el valor de la expresión es igual al de una de las etiquetas, entonces se transfiere el flujo de control a las sentencias asociadas con la etiqueta correspondiente y se ejecuta la instrucción o bloque de instrucciones correspondiente a dicha etiqueta. Si el valor del selector no está listado en ninguna etiqueta, no se ejecutará ninguna de las opciones, a menos que se especifique la acción por defecto **else**/**otherwise**. Al igual que ocurre con la sentencia **if**, el uso de **else** en estas estructuras es opcional, aunque se recomienda su uso. Si no se encuentra la opción por defecto y el valor del selector no se recoge en ninguna etiqueta, la ejecución continúa con la siguiente sentencia que sigue a la estructura de selección **case of**.

La sintaxis de esta estructura de selección es la siguiente:

```
case {ordinal o cadena} of        // selector
  etiqueta1: intruccion1;
  etiqueta2: instruccion2;
  . . .
  else/otherwise instruccionN;  // acción por defecto
end;
```

El tipo de dato de las etiquetas o casos debe ser el mismo que el del selector y puede contener un rango de valores, contiguos o no. Sin embargo, un valor dado no podrá repetirse en ninguna etiqueta porque el compilador mostrará un error.

A diferencia de la estructura **if**, en **case** sí es obligatorio que cada etiqueta se separe de la siguiente por un punto y coma.

En el siguiente ejemplo emplearemos una estructura de selección **case of** para determinar si el carácter leído por teclado es una vocal, una consonante, un número u otro carácter no alfanumérico.

Abra un nuevo proyecto de consola en Lazarus, guárdelo como **teclado** y adapte el código fuente para que se parezca a este:

```
program teclado;
{$mode objfpc}{$H+}
procedure TeclaPulsada(Tecla: Char);
begin
     case upCase(Tecla) of
       '0'..'9': WriteLn('La tecla ',Tecla,' es un
            numero');
       'A','E','I','O','U': WriteLn('La tecla ',
            Tecla,' es una vocal');
       'B'..'D','F'..'H','J'..'N','P'..'T','V'..'Z':
         WriteLn('La tecla ',Tecla,' es una consonan
            te');
     else WriteLn('La tecla ',Tecla,' es un simbolo');
     end;
end;

var cd: string;
begin
     WriteLn('Pulse una tecla y despues [Enter], (o
       solo [Enter] para finalizar)');
     repeat
       ReadLn(cd);
       if cd <> '' then TeclaPulsada (cd[1]);
     until cd = '';
end.
```

Dese cuenta de que hemos comprobado qué tecla ha pulsado el usuario convirtiéndola previamente a mayúscula con la función `upCase(Tecla)`. De este modo evitamos incluir en la estructura de selección **case** de la rutina `TeclaPulsada` las comprobaciones para las letras minúsculas.

## 6.2 ESTRUCTURAS REPETITIVAS

Tan importantes como las estructuras condicionales son las sentencias iterativas en el control de flujo de un programa. Pascal, como la mayoría de los lenguajes de programación, proporciona sentencias repetitivas que permiten realizar una tarea una y otra vez hasta que se cumpla una determinada condición. En programación, a estas sentencias también se les llama **bucles**; y a las instrucciones que se repiten en ellos, **cuerpo del bucle**.

En Lazarus podemos trabajar con tres tipos de estructuras repetitivas: los bucles **for**, **while do** y **repeat until**.

### 6.2.1 El bucle *for*

La instrucción iterativa **for** se utiliza para crear bucles con un número predeterminado de repeticiones. Este bucle es uno de los más usados para repetir una secuencia de instrucciones, sobre todo cuando se conoce la cantidad exacta de veces que se quiere que se ejecuten. En Pascal, su sintaxis es la siguiente:

```
for contador:= inicio to fin do
  begin
  {acciones que deben ejecutarse}
  end;
```

El funcionamiento del bucle **for** es el siguiente: primero se comprueba si la variable `contador` rebasa el límite marcado como `fin`. Si es así, entonces no se ejecuta el cuerpo del bucle. En caso contrario, se le asigna a `contador` el valor `inicio` y se ejecutan las instrucciones que forman el cuerpo una vez. A continuación, se incrementa o disminuye una unidad el valor de `contador`. Si este nuevo valor está comprendido entre el valor inicial y el valor final, entonces se vuelve a ejecutar el cuerpo, y así sucesivamente hasta que el contador alcance el valor final (Figura 6.2).

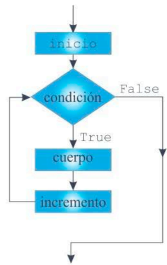



**Figura 6.2.** Diagrama de flujo de un bucle for

La sentencia **`for`** admite en Pascal dos variantes: la instrucción ascendente (**`for to do`**) y la instrucción descendente (**`for downto do`**). La instrucción descendente tiene un comportamiento análogo, salvo que la variable `contador` se disminuye de uno en uno cada vez que se ejecuta el cuerpo del bucle.

Es conveniente recordar que la variable de control, a la que hemos llamado `contador`, puede ser de cualquier tipo ordinal. Durante la ejecución del bucle, el compilador no le dejará modificar el valor de esta variable.

Suponga que necesita escribir una función que devuelva el **factorial** de un número natural *n*, es decir, el producto de todos los números naturales hasta *n*. Esta función podría codificarla mediante una sentencia **`for`** que manejara un contador descendente desde *n* hasta 1, como muestra el siguiente ejemplo:

```
var contador: integer;
    numero: integer = 7;
    factorial: integer = 1;

begin
  for contador:= numero downto 1 do
  begin
      factorial*= contador;
  end;
  WriteLn (numero,'! = ', factorial);
  ReadLn;
end.
```

¿Sencillo? Ahora modifique ligeramente el código para que sea el usuario quien pueda introducir por teclado el número cuyo factorial desea hallar.

Recordemos ahora las características fundamentales que deberá tener siempre presentes cuando diseñe un bucle **`for`**:

1. Las expresiones que definen los límites inicial y final se evalúan una sola vez antes de la primera iteración.

2. El bucle se repite un número conocido de veces.

3. El valor de la variable de control se comprueba antes de ejecutar el bucle.

4. El incremento o disminución del índice del bucle es automático, por lo que no se debe incluir ninguna instrucción para efectuarlo.

5. El bucle **for** termina cuando el valor de la variable de control sale fuera del intervalo de valores establecido.

Pascal, como la inmensa mayoría de los lenguajes de programación, permite **anidar bucles**, lo que resulta de gran utilidad siempre que se desee mostrar o calcular datos en forma de tabla o matriz. La regla es sencilla: el bucle interno debe completarse antes de que se ejecute la siguiente sentencia **for** del bucle externo. Observe en el siguiente ejemplo cómo se hallan los números primos comprendidos entre los enteros 2 y 50 mediante el uso de dos bucles **for** anidados:

```
var
   i, j:integer;
begin
   for i := 2 to 50 do
   begin
     for j := 2 to i do
       if (i mod j)=0  then
          break; // si se halla el factor, no es primo
     if(j = i) then
        writeln(i , ' es primo' );
   end;
   readln;
end.
```

**NOTA**

Si en un bucle **for** no existen las palabras reservadas **begin** y **end**, el compilador asume que la primera sentencia tras la de **for** constituye el cuerpo del bucle.

### 6.2.2 La sentencia *for in*

Desde la versión 2.4.2 de Free Pascal, los programadores disponemos de un nuevo bucle, la construcción **for in**. Generalmente, se emplea para recorrer datos finitos en una expresión numerable. Su sintaxis es la siguiente:

```
for variable in dato do
  {cuerpo}
```

Donde `variable` es el identificador del bucle y `dato` una expresión numerable, es decir, formada por un número fijo de elementos: un arreglo, un conjunto o un tipo enumerado. La variable de control recorrerá cada uno de los elementos de `dato` para ejecutar con cada uno el cuerpo del bucle.

De los cinco tipos de expresiones que puede adoptar la expresión enumerada, nos centraremos en tres:

- **Tipo enumerado**. La variable de control recorre los elementos definidos en el tipo enumerado. El identificador del bucle debe ser de tipo enumerado.
- **Conjunto de valores**. La variable de control recorre los elementos del conjunto. El identificador del bucle debe ser del tipo base del conjunto.
- **Arreglo**. La variable de control recorre los elementos del arreglo. El identificador del bucle debe ser del mismo tipo que cada elemento del arreglo.

Veamos cómo trabajar con cada una de estas expresiones. La situación más simple es usar el bucle con un **tipo enumerado**:

```
type
 TDiasSemana = (lunes, martes, miercoles, jueves,
  viernes, sabado, domingo);
var
 dia : TDiasSemana;
begin
for dia in TDiasSemana do
  writeln(dia);
end.
```

Esto imprimirá en pantalla los siete días de la semana, uno en cada línea.

El segundo caso de utilización de este bucle es cuando los datos de la expresión forman un **conjunto**:

```
type
 TDiasSemana = (lunes, martes, miercoles, jueves,
  viernes,sabado,domingo);
var
 SemanaLaboral: set of TDiasSemana = [lunes, martes,
  miercoles, jueves, viernes];
 dia : TDiasSemana;
begin
for dia in SemanaLaboral do
    writeln(dia);
end.
```

Esto imprimirá en pantalla los cinco días de la semana laboral, uno en cada línea. Observe cómo la variable `dia` es del mismo tipo que el tipo base del conjunto.

La tercera posibilidad es que la expresión numerable sea un **arreglo**:

```
var
 Semana: array[1..7] of String = ('lunes', 'martes',
'miercoles', 'jueves', 'viernes','sabado','domingo');
cd: String;
begin
for cd in Semana do
    WriteLn(cd);
end.
```

Esto también mostrará en pantalla los siete días de la semana. Fíjese cómo la variable `cd` es de tipo **`String`**, igual que cada uno de los elementos del arreglo.

### 6.2.3 El bucle *while*

En muchas situaciones no se sabe de antemano cuántas veces deberá ejecutarse el cuerpo de un bucle. En estos casos ya no podemos emplear la sentencia **`for`**, sino el bucle **`while`**. Su sintaxis es la siguiente:

```
while condicion do
  begin
  {cuerpo del bucle}
  end;
```

Y su diagrama de flujo el mostrado en la figura 6.3. La ejecución de una estructura **`while`** comienza con la comprobación de la condición; si esta es falsa, entonces se salta el cuerpo del bucle que sigue a la palabra reservada **`do`**. Si la condición es verdadera, se ejecuta el cuerpo, se vuelve a comprobar la condición al finalizar, y así sucesivamente hasta salir del bucle.

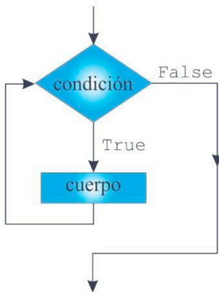



**Figura 6.3.** Diagrama de flujo de un bucle while

Para una correcta utilización de la instrucción **`while`** es necesario que la instrucción modifique las variables que aparecen en la condición, ya que en caso contrario, si la condición es verdadera, siempre permanecerá así y el bucle no terminaría nunca. Una situación en que se puede producir este error surge cuando el cuerpo del bucle es una secuencia de instrucciones y se olvida utilizar los delimitadores **`begin`** y **`end`**.

Los bucles **`while`** se pueden anidar simplemente escribiendo la instrucción **`while`** interior como una sentencia más dentro del cuerpo de otro bucle **`while`**. Si el bucle exterior no llega a ejecutarse, por ser falsa su condición, tampoco lo hará el bucle interior. Si, por el contrario, el bucle exterior se ejecutara, entonces se evaluaría la condición del bucle interior y, si también fuera verdadera, se ejecutarán sus instrucciones interiores hasta que su condición se vuelva falsa, tras lo cual el control volvería a la estructura **`while`** exterior.

En el siguiente ejemplo emplearemos una estructura iterativa **`while`** para determinar si un número natural leído por teclado es o no un número primo.

Abra un nuevo proyecto de consola en Lazarus, guárdelo como **`primos`** y adapte el código fuente para que se parezca a este:

```
program primos;
{$mode objfpc}{$H+}
var
  numero: integer;
  divisor: integer = 2;
  primo: boolean = True;

begin
  Write('Introduzca un numero natural: ');
  ReadLn(numero);
  while (divisor <= trunc(sqrt(numero))) and primo do
   begin
     if numero mod divisor = 0 then
       primo:= false;
     divisor += 1;
   end;
  if primo = true then
    writeln(numero,' es PRIMO.')
  else
    writeln(numero,' NO es primo.');
{$IFDEF WINDOWS}
 ReadLn;
{$ENDIF}
end.
```

Recordemos las propiedades principales que debe tener presentes sobre la estructura iterativa **while**:

1. La condición se comprueba al principio del bucle, antes de ejecutar el cuerpo.
2. El bucle termina cuando la condición deja de cumplirse.
3. Como consecuencia de los puntos anteriores, la instrucción se ejecuta cero o más veces.

### 6.2.4 La estructura *repeat*

La sentencia **repeat** es similar al bucle **while**, salvo que la condición se evalúa al final del cuerpo del bucle, por lo que al menos siempre se ejecuta una vez. Su sintaxis es la siguiente:

```
repeat
  {cuerpo}
until (condicion);  // finaliza cuando se cumpla
```

Las palabras reservadas **`repeat`** y **`until`** delimitan el cuerpo del bucle, por lo que no es necesario incluir los delimitadores **`begin`** y **`end`**. Su diagrama de flujo es el mostrado en la figura siguiente:

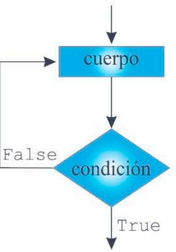



**Figura 6.4.** Diagrama de flujo de un bucle repeat

La función de una estructura **`repeat`** es, por lo tanto, repetir las instrucciones indicadas en el cuerpo del bucle hasta que se verifique la condición que aparece tras **`until`**. A continuación, se pasa a la siguiente instrucción externa al bucle.

Como ejemplo de utilización de esta estructura iterativa, estudie el siguiente fragmento de código con el que pueden sumarse los *n* primeros números naturales.

```
ReadLn(n); // Supuesto que n >= 1
suma:= 0;
contador:= 0;
repeat
  contador += 1;
  suma += contador
until contador = n
```

Observe que la condición $n >= 1$ es imprescindible para que el resultado final sea el esperado. En general, siempre es conveniente comprobar el comportamiento del bucle en valores extremos; en este ejemplo, para $n = 0$ se generaría un bucle infinito, lo cual se evitaría sustituyendo la condición `contador = n` por `contador`

`>= n`. En este caso, dada la característica del bucle **`repeat`**, las instrucciones interiores se ejecutarán al menos una vez, por lo que la suma valdrá al menos 1.

Las propiedades principales de la instrucción **`repeat`** son las siguientes:

1. El bucle admite una lista de instrucciones interiores, no siendo necesario utilizar los delimitadores **`begin end`**.
2. La condición se comprueba después de ejecutar la lista de instrucciones, por lo que esta se ejecuta al menos una vez.
3. El bucle termina cuando se cumple la condición.
4. Como consecuencia de los puntos anteriores, el cuerpo de instrucciones siempre se ejecuta una o más veces.

## 6.3 SENTENCIAS DE CONTROL EN LOS BUCLES

Dentro de los bucles podemos añadir ciertas sentencias de control que permitan modificar la normal ejecución de aquellos. Pascal soporta tres instrucciones básicas para este fin: **`break`**, **`continue`** y **`goto`**; aunque solo hablaremos de las dos primeras.

### 6.3.1 *Break*

La sentencia **`break`** se usa, principalmente, en dos situaciones:

1. Cuando se quiera terminar un bucle y pasar el control del programa a la primera sentencia tras el mismo.
2. Para finalizar una estructura de selección múltiple **`case of`** como sentencia asociada a una de las etiquetas.

En caso de usar bucles anidados, una instrucción **`break`** en el bucle más interno detendrá su ejecución y se transferirá el flujo a la siguiente línea de código tras el bloque.

La sintaxis de una sentencia **break** es la siguiente:

```
break;
```

y su diagrama de flujo:

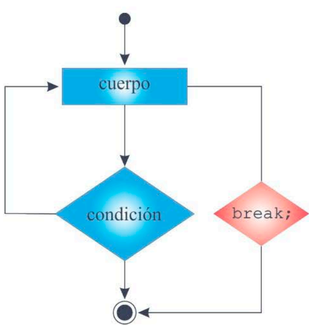



**Figura 6.5.** Diagrama de flujo de un bucle con una instrucción break

Veamos un ejemplo de su aplicación que entenderá perfectamente. Abra un nuevo proyecto de consola en Lazarus y copie el siguiente código fuente:

```
program ejemploBreak;
var
 a: integer;
begin
 a := 5;
 while a < 20 do
  begin
   writeln('El valor de a es: ', a);
   a += 1;
   if( a > 10) then
    (* termina el bucle *)
    break;
  end;
 readln;
end.
```

Cuando ejecute el programa, solo se mostrarán en pantalla los valores de `a` comprendidos entre 5 y 10, aun cuando el bucle **`while`** se ha construido para repetir la acción hasta que `a = 19`. ¿Por qué, entonces, no se muestran todos? La respuesta es que la sentencia `break` ha interrumpido el desarrollo normal del bucle en cuanto la variable `a` ha alcanzado el valor `11`.

### 6.3.2 *Continue*

La sentencia **`continue`** trabaja en un bucle de un modo análogo a como lo hace **`break`**, salvo que en vez de forzar la terminación del mismo, obliga a que el flujo de programa pase a la siguiente iteración. La sentencia **`continue`** salta cualquier sentencia posterior en el bucle y devuelve el flujo al comienzo de la siguiente iteración del bucle actual, hecho que deberá tener en cuenta cuando trabaje con bucles anidados.

La sintaxis de una sentencia **`continue`** es la siguiente:

```
continue;
```

y su diagrama de flujo:

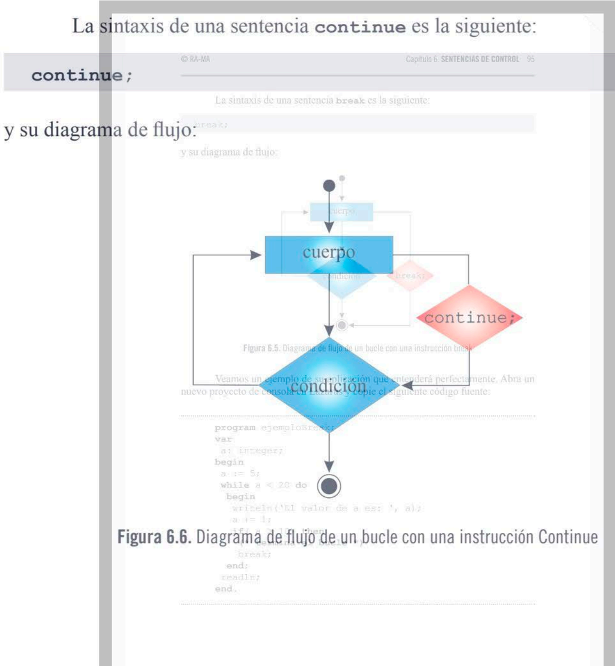



**Figura 6.6.** Diagrama de flujo de un bucle con una instrucción Continue

```
program ejemploContinue;
uses sysutils;
var
  i: integer;
  s : string;

begin
  s := '';
   for i := 1 to 9 do
    begin
      // Saltamos dos valores
     if (i = 3) or (i = 7) then Continue;
     s := s + IntToStr(i);
     s := s + ' ';
    end;
  WriteLn ('s = '+s);
  ReadLn;
end.
```

Cuando ejecute el programa, se le mostrará en la consola la cadena `s = 1 2 4 5 6 8 9`. Los valores `3` y `7` no aparecerán porque la sentencia `Continue` ha llevado para estos valores el flujo del programa al comienzo de la siguiente iteración del bucle **`for`**, que se corresponde con los valores `4` y `8` de la variable `i`.

## 6.4 LAS EXCEPCIONES

Las **excepciones** son eventos que ocurren por un error del programa en tiempo de ejecución, y se generan para indicarle al programador que ha ocurrido un error que impide la normal ejecución del programa.

Las excepciones hacen que los programas sean más robustos, ya que proporcionan un modo de notificar y gestionar errores y otras situaciones inesperadas. Esto permite que los programas sean más fáciles de leer, escribir y depurar. El manejo de las excepciones permite independizar el código de uso de un recurso del manejo de los errores, lo cual no retrasará la ejecución del programa porque el código de acceso al recurso se ejecuta siempre, pero el código de manejo de errores se ejecuta solo cuando estos suceden.

El **soporte de excepciones** es otra de las características importantes de Lazarus. Cuando algo no va bien en tiempo de ejecución, las bibliotecas de Lazarus crean excepciones. Desde el punto de vista del código en el que se crea, la excepción se

pasa a su código de llamada, y así sucesivamente. Si ninguna parte del código controla la excepción, la LCL se encarga de ella y muestra un mensaje estándar de error.

La gestión de excepciones no supone un reemplazo del adecuado control de flujo de un programa. Por eso se recomienda mantener el uso de sentencias **`if`** para comprobar la entrada del usuario y otras posibles condiciones de error.

### 6.4.1 Flujo de programa

La potencia de las excepciones en Lazarus tiene que ver con el hecho de que se pasan de una rutina hasta un manipulador o manejador global, en lugar de continuar la ruta estándar de ejecución del programa. Así que el problema consiste en cómo ejecutar código incluso aunque se lance una excepción.

Cuando Lazarus lanza una excepción, crea una instancia de una clase especial denominada **`Exception`** o algún descendiente, el flujo del programa se altera y únicamente se ejecuta el código de respuesta a la excepción. Este código puede tratar el error o no, en cuyo caso se genera un evento **`OnException`** en la aplicación. Si la excepción no se atiende, termina el programa.

Todo el mecanismo de gestión de excepciones se basa en el uso de cuatro palabras reservadas:

- **`try:`** delimita el comienzo de un bloque de código protegido.
- **`except:`** delimita el final del bloque de código protegido e introduce sentencias de control de excepciones.
- **`finally:`** especifica bloques de código que han de ejecutarse siempre, incluso cuando se den excepciones. En general, se emplea este bloque para realizar operaciones de limpieza.
- **`raise:`** es la sentencia que se utiliza para generar la excepción. La mayoría de las excepciones en Lazarus las genera el sistema, pero también se pueden crear excepciones propias en el código cuando se descubren datos incoherentes en tiempo de ejecución.

Lazarus posee dos tipos de manejadores de excepciones: **`try/except`** y **`try/finally`**. El primero de ellos es solo para manejar errores, mientras que el segundo permite liberar recursos.

El código que forma el bloque que va a quedar protegido por el manejador de excepciones es el que se coloca entre **`try`** y **`except`** o entre **`try`** y **`finally`**.

La diferencia entre ambos manejadores es que el primero solo ejecuta el bloque entre **`except`** y el **`end`** si ocurre un error en el bloque protegido, mientras que el segundo ejecuta el bloque entre el **`finally`** y el **`end`** siempre, tanto si ocurre un error como si no (Figura 6.7).

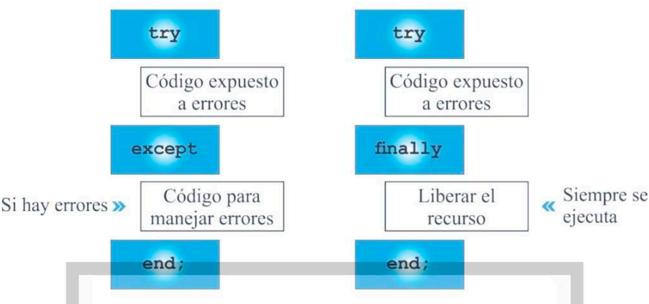



**Figura 6.7.** Estructura de los dos manejadores de excepciones en Lazarus

Un bloque **`try`** puede ir seguido de una sentencia **`except`** o **`finally`**, pero no por ambas al mismo tiempo. La solución más habitual para controlar también la excepción consiste en usar dos bloques **`try` anidados**. Por ejemplo, cada vez que se realiza una división, hay que tener en cuenta el error que puede producirse si el divisor es cero. Según se trate de una división entera o en coma flotante, se dispara una excepción **`EDivByZero`** o **`EZeroDivide`**, respectivamente.

Vea en el siguiente fragmento de código cómo puede gestionarse el error al realizar una división entera por 0 (Figura 6.8):

```
try
  try    // Error al dividir por 0…
    Resultado := J div I;  // I es 0
  finally
    Screen.Cursor := crDefault;
  end;
except
   on E: EDivByZero do
   begin   // Excepción con nuevo mensaje
     raise Exception.Create ('Error en División');
   end;
 end;
```

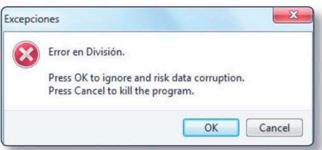



**Figura 6.8.** Excepción lanzada en una división por cero

## 6.4.2 Clases de excepciones

En Lazarus las excepciones se pueden dividir en las siguientes categorías:

- **Conversión de tipo.** Se producen cuando se intenta convertir un tipo de dato en otro. Lazarus lanza la excepción EConvertError.
- **Tipo forzado.** Se producen cuando se intenta forzar el reconocimiento de una excepción de un tipo como si fuera de otro empleando el operador as. Si no son compatibles, se dispara la excepción EInvalidCast.
- **Aritmética entera.** Se producen al hacer operaciones con expresiones de tipo entero:
  - **EDivByZero.** División por cero.
  - **ERangeError.** Número fuera del rango disponible según el tipo de dato.
  - **EIntOverflow.** La capacidad aritmética queda desbordada por números demasiado grandes.
- **Aritmética de punto flotante.** Se producen al hacer operaciones con expresiones de tipo real:
  - **EInvalidOp.** El procesador encontró una instrucción inválida.
  - **EZeroDivide.** División por cero.
  - **EOverFlow.** Capacidad aritmética desbordada por números demasiado grandes.

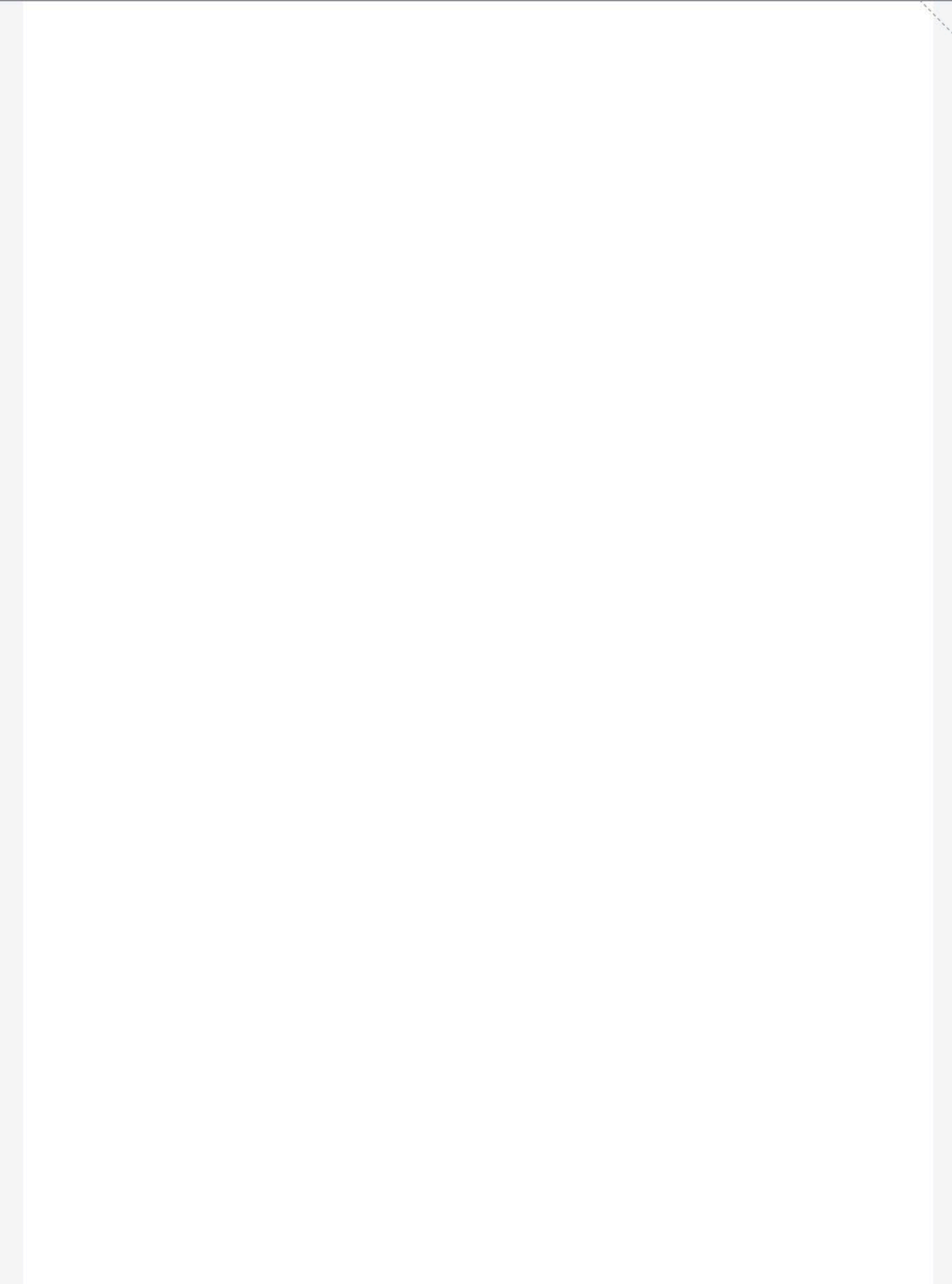
- **Falta de memoria.** Se produce cuando aparece un problema al acceder o reservar memoria:

  - **EOutOfMemory.** No existe suficiente memoria disponible para completar la operación.
  - **EInvalidPointer.** La aplicación intenta disponer de la información referenciada por un puntero que indica una dirección inválida, generalmente, porque la memoria ya ha sido liberada.

- **Entrada/Salida.** Se produce cuando hay un error al acceder a un dispositivo de entrada/salida o a algún archivo. Lazarus define la clase genérica EInOutError.

**TRUCO**
Demasiados bloques `try/except` para controlar las excepciones en el código indicarán o causarán, probablemente, errores en el flujo del programa. Por ello, es más importante utilizar los bloques `try/ finally` que controlar las excepciones.

El manejo de excepciones es bastante complejo para alguien que comienza a programar, pero, por fortuna, es muy probable que no necesite considerarlas durante mucho tiempo. Además, como manejar excepciones toma tiempo y consume recursos, solo debería usarlas en condiciones excepcionales, como cuando un programa no encuentra un determinado fichero.



# 7

# FUNCIONES Y PROCEDIMIENTOS

Para la construcción de programas de cierta envergadura es necesario disponer de herramientas que permitan organizar el código. Por una parte, las técnicas de la programación estructurada hacen posible relacionar las acciones que vayan a realizarse mediante constructores de secuencia, selección e iteración, tal y como se vio en los capítulos anteriores. Por otra parte, siguiendo el paradigma de la programación modular, Pascal facilita la división de un programa en módulos o subprogramas con el fin de hacerlo más manejable y legible para el programador. Estos subprogramas pueden ser invocados desde diferentes puntos del programa principal tan solo escribiendo sus nombres. Así se extiende el juego de instrucciones básicas con otras nuevas a la medida del problema que se está resolviendo. Una elección adecuada de subprogramas facilita en gran medida el paso de los algoritmos a los programas, especialmente cuando se sigue un método de análisis descendente.

Si bien un **módulo** puede entenderse como una parte de un programa en cualquiera de sus formas y variados contextos, en la práctica se los suele tomar como sinónimos de **procedimientos** y **funciones**. Procedimientos y funciones, de forma genérica, se conocen en programación como **rutinas**.

Las rutinas pueden tener declaraciones **locales** tanto de variables como de constantes y poseer procedimientos o funciones anidados que son **privados** en la rutina. Los términos *local* y *privado* deben entenderse aquí en el sentido de que las variables declaradas solo pueden usarse dentro de ese procedimiento o función, pues no serán visibles para el resto del programa. Este es el significado de *privado* que de forma general maneja Pascal. Una rutina se comporta, desde este punto de vista, como un **agujero negro** para el resto del programa. Por lo único que este reconocerá a las funciones y procedimientos será por sus nombres y listas de parámetros.

## 7.1 FUNCIONES

Una **función** es un grupo de sentencias que devuelve un valor tras su ejecución. La declaración de la función le indicará al compilador cuál es su nombre, los parámetros que necesita para su ejecución y el tipo de valor que devuelve.

Las librerías de Lazarus proveen miles de funciones listas para usarse. El reto para cualquier programador es conseguir encontrar la adecuada para su programa; como esto no es siempre posible, no queda más remedio que programar la función que se necesite en cada momento. Ahora bien, una recomendación esencial es que si Lazarus ya posee una función para una determinada tarea, no programe otra, úsela. Las funciones de sus librerías están completamente optimizadas para la labor que realizan, son rápidas y han sido probadas por decenas de miles de programadores.

### 7.1.1 Declaración y definición de una función

En Pascal, una función se define utilizando la palabra reservada **`function`**. La declaración le indica al compilador no solo el nombre de la función, sino también cómo ha de ser invocada en el código fuente.

La estructura general de la definición de una función en Lazarus es la siguiente:

```
function NombreFuncion(argumento(s): tipo1; argumento (s) :
   tipo2; ...): tipo;
<declaraciones locales>
begin
 ...
 <instrucciones>
 ...
 Result:= expresión; //dato devuelto por la función
end;
```

Observe cómo en la declaración de la función se escribe, entre paréntesis, una lista con los argumentos que recibe la función y el tipo de dato de cada uno de ellos. La declaración finaliza con el tipo que devuelve la función, que será el de la asignación encabezada por `Result`. Una alternativa que utilizan muchos programadores para devolver el resultado es emplear el propio nombre de la función. Veamos un ejemplo muy sencillo. Definamos una función para hallar el volumen de una esfera cuyo radio conocemos:

```
function VolumenEsfera(aRadio: double): double;
begin
  Result:= 4/3 * Pi * (aRadio ** 3);
end;
```

Observe que la función solo admite un parámetro, el radio de la esfera, `aRadio`, definido como de tipo `double`. La función también se ha definido del mismo tipo, pues devuelve el volumen de la esfera, que es un número real de tipo `double`. Así mismo, es una convención entre los programadores anteponer una letra `a` al nombre de los parámetros recibidos por las funciones para distinguirlos rápidamente en el código fuente de aquellas variables que puedan definirse y emplearse en el cuerpo de una función.

**IMPORTANTE**
El operador potencia `**` está definido en Lazarus en la librería `Math`, por lo que no se olvide de añadirla en la cláusula `Uses`.

Recuerde que podría haber sustituido la variable `Result` en la función por el nombre de la función, `VolumenEsfera`. Ambas formas son completamente válidas en Pascal.

En la definición de cualquier función aparecerá, por tanto, una **cabecera**, las **declaraciones locales** y el **cuerpo**. Veamos con más detenimiento cada una de ellas:

- **Cabecera**. La cabecera consiste en la palabra clave **`function`** seguida del nombre con el que el compilador conocerá a la función. Además, aparecerán otros dos elementos:

  - **Argumentos**. Los argumentos de una función son el nexo común entre el programa y la propia función. También se denominan *parámetros*. Cuando se llame en el código a una función, se le debe pasar el parámetro o parámetros necesarios para su correcta ejecución. La lista de parámetros se refiere al tipo, número y orden de los parámetros.

  - **Tipo devuelto**. Como todas las funciones devuelven valores, estas deben tener asignado un tipo. El tipo de una función es el del dato del valor que devuelve.

- **Declaraciones locales**. Las declaraciones se refieren a las etiquetas constantes, variables, funciones y procedimientos que serán de aplicación en el cuerpo de la función.

- **Cuerpo de la función**. El cuerpo de una función contiene el conjunto de instrucciones que definen lo que hace la función. Siempre irá entre las palabras reservadas **`begin`** y **`end`**. La función acabará con una asignación que devolverá un valor cuando la función haya finalizado de ejecutarse. Es habitual asignar ese valor al nombre mismo de la función, aunque de forma predefinida Free Pascal provee la variable `Result`. Se comporta, de hecho, como una variable oculta que no está definida hasta que el flujo del programa pasa a la función.

### 7.1.2 Parámetros de referencia

Los parámetros pueden pasarse a las funciones o procedimientos de cuatro formas: como **valor**, **variable**, **constante** o como un parámetro **externo**. Se reconocen por el uso de las palabras reservadas **`var`**, **`const`** y **`out`** que, en caso de emplearse en una rutina, aparecerán justo antes del nombre del parámetro correspondiente. Veamos con algo más de detalle cada uno de ellos:

- **Parámetros como valor**. Son los parámetros que por defecto el compilador asume que se le pasarán a la rutina, a menos que se especifique lo contrario. Estos parámetros transfieren una copia de su valor actual a la pila y la rutina usa y maneja la copia, no el valor original. Por ejemplo, en la declaración de la función `VolumenEsfera`:

  ```
  function VolumenEsfera(aRadio: double): double;
  ```

  El único parámetro que necesita es de tipo `valor`. De hecho, podríamos pasárselo como un número literal. Tras la llamada a la función o rutina, los parámetros por valor permanecerán inalterables.

- **Parámetros como variables**. Estos se pasan a las rutinas por referencia, es decir, no se realiza una copia del valor en la pila, sino que la rutina actúa sobre el valor original del parámetro. Esto permite al procedimiento o función cambiar el valor del argumento, como veremos en un ejemplo al finalizar el capítulo. Se identifican porque se antepone al nombre del parámetro la palabra reservada **`var`**.

- **Parámetros como constantes**. Son como los parámetros de valor, salvo que no se les puede asignar un valor en el cuerpo de la rutina. Ya que no puede asignar un nuevo valor a un parámetro constante dentro de una rutina, el compilador puede optimizar la transmisión de parámetros, pero el comportamiento seguirá siendo similar a los parámetros de valor. Se reconocen porque el nombre del parámetro va precedido de la palabra reservada `const`.

- **Parámetros externos**. Un parámetro externo no tiene valor inicial y se identifica porque se antepone la palabra reservada `out` al nombre del parámetro. Se usan para devolver un valor desde una rutina, como ocurre con la variable `Result` de una función. Excepto por no tener valor inicial, los parámetros externos se comportan como los parámetros por referencia, aunque, en general, es mejor ceñirse a estos últimos por ser mucho más eficientes.

### 7.1.3 Argumentos por defecto

Pascal permite declarar parámetros con un determinado valor por defecto para los tipos básicos de datos, incluidas las cadenas. Esta asignación se hace en la misma declaración de la rutina. Si cuando se invoca a la función o procedimiento no se le suministra ese parámetro, entonces el compilador asumirá que su valor es el declarado por defecto.

Esto significa que podemos declarar funciones en apariencia sobrecargadas, por ejemplo:

```
function MostrarMsj (cd1:string; cd2:string='
Mundo'):string;
```

De modo que ahora podemos llamar a la función con uno o dos parámetros:

```
MostrarMsj ('Hola');
MostrarMsj ('Hola', ' Marta');
```

El resultado de la primera llamada sería `Hola Mundo` y el de la segunda, `Hola Marta.`

### 7.1.4 Sobrecarga de funciones

La idea de la sobrecarga de funciones es sencilla. El compilador permite definir dos funciones o procedimientos usando un mismo nombre, suponiendo que los parámetros sean distintos. Cuando el compilador compruebe los parámetros, determinará automáticamente a cuál de las versiones de la rutina nos estamos refiriendo.

Considere esta sucesión de funciones extraída de la unidad `Math` de la RTL de Lazarus:

```
function Max(a,b: Integer): Integer; overload;
function Max(a,b: Int64): Int64; overload;
function Max(a,b: Extended): Extended; overload;
```

Todas ellas se llaman igual y poseen dos parámetros también, pero son de distinto tipo. Cuando efectúe una llamada a `Max (5, 27)`, el compilador determina automáticamente que se está invocando a la primera función del grupo, con lo que el valor de salida será el entero `27`.

Las reglas básicas que debe considerar cuando programe funciones sobrecargadas son dos:

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

## 7.2 PROCEDIMIENTOS

Los procedimientos, en Pascal, son subprogramas que, en vez de devolver un único valor, como las funciones, permiten obtener un conjunto de resultados.

### 7.2.1 Definición de un procedimiento

Los procedimientos se definen con la palabra reservada **procedure**. Su estructura general es la siguiente:

```
procedure NombreProc (argumento(s): tipo1; argument
   to(s): tipo2, ... );
<declaraciones locales>
begin
 <cuerpo>
end;
```

Como las funciones, un procedimiento consta de una **cabecera**, las **declaraciones locales** y el **cuerpo** de instrucciones. En la cabecera se recoge el nombre del procedimiento y los parámetros que, como en el caso de las funciones, pueden pasarse por valor o referencia.

## 7.2.2 Declaración

La declaración de un procedimiento le indica al compilador cómo se llama al mismo en el programa principal. Posee la siguiente sintaxis:

```
procedure NombreProc (argumento(s): tipo1; ... );
```

Es importante observar que el nombre de un procedimiento no está asociado a ningún tipo especial de dato, a diferencia de lo que ocurre con las funciones.

Para llamar al procedimiento, junto con su nombre, se le pasarán los parámetros o argumentos necesarios. Cuando el programa general invoca el procedimiento, el control se transfiere a este último y, cuando se alcance la sentencia **end** final, el control se devuelve de nuevo al programa principal.

## 7.2.3 Argumentos

Los argumentos de un procedimiento, como en el caso de las funciones, pueden pasarse por valor o referencia y, como en aquellas, se comportan del mismo modo.

Veamos ahora un ejemplo para demostrar cómo pasar parámetros por valor o por referencia, lleva a resultados distintos. Recuerde que cuando los argumentos se pasan por valor, que es la forma que Lazarus usa por defecto, salvo que se indique lo contrario, el compilador trabaja con una copia de los mismos, por lo que los valores permanecerán sin modificación alguna tras la ejecución de la función o parámetro.

Abra un nuevo proyecto en modo consola en Lazarus con el nombre **llamadaporvalor** y copie el siguiente código:

```
program llamadaporvalor;
var
  a, b: integer;
(*definicion del procedimiento*)
procedure intercambio(x, y: integer);
var
  temp: integer;
begin
  temp := x;
  x:= y;
  y := temp;
end;
begin
  a := 10;
  b := 20;
  writeln('Antes del intercambio, a es : ', a);
  writeln('Antes del intercambio, b es : ', b);
(*llamada por valor al procedimiento*)
  intercambio(a, b);
  writeln('Tras el intercambio, a es : ', a );
  writeln('Tras el intercambio, b es : ', b );
  readln;
end.
```

Cuando ejecute el programa verá la siguiente salida:

```
Antes del intercambio, a es: 10
Antes del intercambio, b es: 20
Tras el intercambio, a es: 10
Tras el intercambio, b es: 20
```

¿Qué ha pasado? Pues nada. Los valores de las variables son exactamente iguales antes que después de intercambiarlas. La explicación es sencilla. Como se han pasado las variables `a` y `b` por **valor**, estas se han copiado a la pila y son los valores que ha manejado el procedimiento. Como no se han podido modificar entonces los valores originales, ambas variables valdrán lo mismo antes y después de ejecutarse la rutina.

Veamos lo que ocurre si ahora le pasamos a la misma rutina los argumentos por **referencia**. Para ello, lo único que debe hacer es añadir la palabra reservada `var` en la declaración de la rutina, de la siguiente forma:

```
procedure intercambio(var x, y: integer);
```

Cuando compile y ejecute el programa verá esta salida:

```
Antes del intercambio, a es: 10
Antes del intercambio, b es: 20
Tras el intercambio, a es: 20
Tras el intercambio, b es: 10
```

Ahora sí se han intercambiado los valores de sendas variables. El motivo es que al pasar los parámetros por referencia, la rutina actúa sobre los valores originales, lo que ha permitido al procedimiento intercambiar sus valores.

### 7.2.4 La rutina *Exit*

El procedimiento `Exit` se encuentra definido en la unidad `System` de la librería RTL de Lazarus. Su empleo permite abandonar en cualquier momento y lugar una rutina. En el momento en que se lee la instrucción, el compilador de Free Pascal salta directamente a la sentencia `end` final de la función o procedimiento en la que se encuentra y se devuelve el control a la rutina que le llamó. Si se invoca, por el contrario, en el cuerpo principal del programa, `Exit` finaliza su ejecución. La rutina de salida, declarada como,

```
procedure Exit (const X: TAnyType);
```

permite que si se escribe en una función, pueda tomar un parámetro único de un tipo apropiado, que será devuelto como valor de la función.

Veamos un ejemplo de su uso con un programa que contará cuántos números existen en una cadena cualquiera que el usuario introduzca por teclado.

Abra un nuevo proyecto en modo consola, guárdelo con el nombre `cuentanumeros` y escriba el código que le mostramos a continuación:

```
program cuentanumeros;

{$mode objfpc}{$H+}

function CuentaNumeros(const cd: string): integer;
var
   c: Char;
begin
if (cd = '') then Exit(0) // devuelve 0 como resultado
else
    begin
    Result:= 0;
    for c in cd do
        if c in ['0'..'9'] then Inc(Result);
    end;
end;

var cd: string;
begin
     Write('Escriba un texto con algun numero: ');
     ReadLn(cd);
     WriteLn('El texto "',cd,'" tiene ',CuentaNumeros
       (cd),' numeros.');

     {$IFDEF WINDOWS}
     ReadLn;
     {$ENDIF}
end.
```

¿Cómo funciona la rutina `Exit(0)` en el programa? La función `CuentaNumeros` recibe la cadena introducida por el usuario. Si es una cadena vacía `(cd = '')`, entonces, devuelve como resultado 0. Evidentemente, no hay ningún número en una cadena vacía.

## 7.3 RUTINAS RECURSIVAS

Como hemos estudiado a lo largo del capítulo, cualquier programa o subprograma puede llamar a cualquier otro, incluido a sí mismo. Es lo que llamamos en programación **recursividad**. Para ilustrar este concepto, vamos a hallar la sucesión de Fibonacci, un ejemplo muy característico de sucesión recurrente.

La sucesión de Fibonacci, de hecho, se define matemáticamente mediante la siguiente relación de recurrencia:

$$f_n = f_{n-1} + f_{n-2}$$

Es decir, cada número natural *n* de la sucesión es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

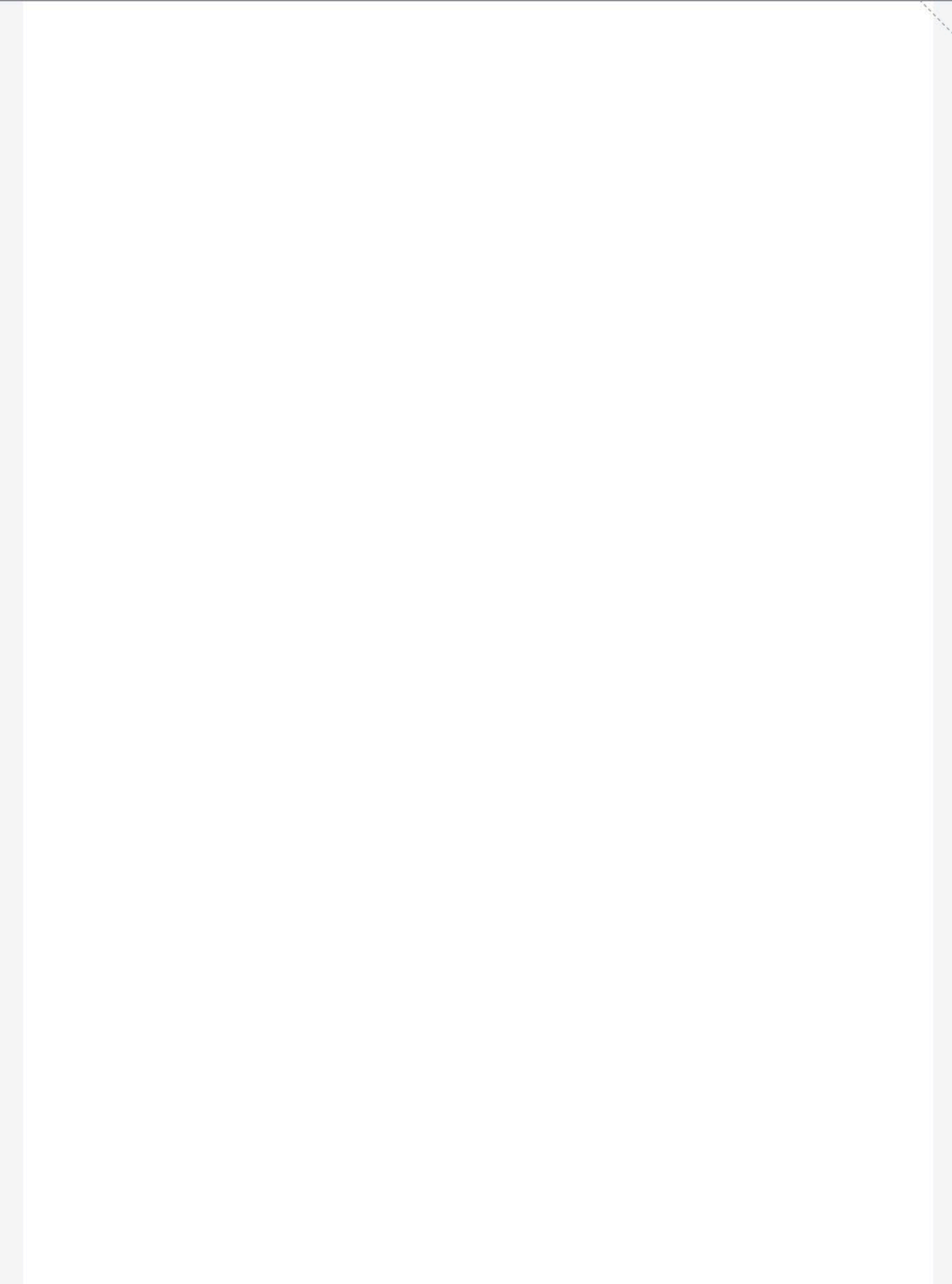
Veamos cómo implementar el algoritmo con el siguiente programa, que calcula los 12 primeros términos de la sucesión mediante una función recursiva que se llamará a sí misma el número necesario de veces:

```
program fibonacci;
var
i: integer;
function fibonacci(n: integer): integer;
begin
  if n = 1 then
    result := 0
 else if n = 2 then
    result := 1
 else
    result := fibonacci(n-1) + fibonacci(n-2);
end;

begin
  for i:= 1 to 12 do  // se hallan los 12 términos
  write(fibonacci (i), ' ');
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Cuando ejecute el programa, verá por pantalla la sucesión formada por los 12 primeros términos de la sucesión:

```
0 1 1 2 3 5 8 13 21 34 55 89
```



# 8

# PROGRAMACIÓN ORIENTADA A OBJETOS

Antes de estudiar los principios básicos necesarios para programar aplicaciones en modo gráfico, es necesario comprender qué son y cómo se trabaja con un tipo especial de datos que en programación orientada a objetos (**POO**) denominamos **clases**.

El entorno de desarrollo Lazarus se basa en una extensión orientada a objetos del lenguaje de programación Pascal conocida como **Object Pascal**, y, como todos los lenguajes que soportan POO, se basa en tres conceptos básicos: la **encapsulación**, normalmente implementada mediante clases, la **herencia** y el **polimorfismo**.

Como ha visto a lo largo de los capítulos anteriores, la sintaxis del lenguaje Pascal es bastante explícita, más legible que la del lenguaje C, por ejemplo. Así pues, su extensión orientada a objetos sigue la misma senda y ofrece, además, la misma potencia que otros lenguajes orientados a objetos, desde Java a C#. Sin embargo, debe conocer que el proyecto Lazarus no está acabado, por lo que el núcleo del lenguaje seguirá sufriendo continuos cambios y mejoras.

## 8.1 CLASES Y OBJETOS

Lazarus se basa en los conceptos de la orientación a objetos y, de forma particular, en la definición de nuevos tipos de clase. El uso de la POO está forzado por el entorno de desarrollo visual, ya que para cada formulario nuevo definido en tiempo de diseño, Lazarus define automáticamente una clase nueva. Además, cada componente situado en un formulario es un objeto de un tipo de clase disponible en la biblioteca del sistema o añadido a ella.

Una **clase** es una construcción definida por el programador que se utiliza como un modelo o plantilla para crear objetos de ese tipo. Esta plantilla describirá el estado y contendrá el comportamiento de todos los objetos creados a partir de esa clase. Las clases son un ente abstracto; los **objetos**, por el contrario, son entidades reales, concreciones de esa clase (Figura 8.1). Un objeto creado a partir de una determinada clase se denomina **instancia** de esa clase.

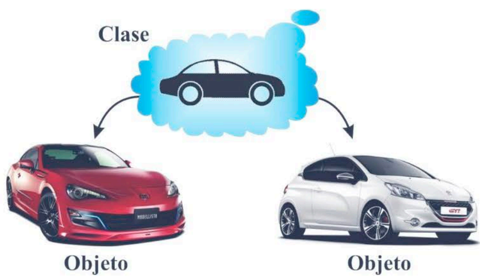



**Figura 8.1.** Una clase es un modelo para crear objetos de ese tipo

En Lazarus, los atributos de un objeto se definen de un modo similar a un registro. La diferencia principal es que un registro solo consta de campos, mientras que en una clase se definen tanto campos como procedimientos y funciones. Los **procedimientos** y las **funciones** que se definan en una **clase** se conocen genéricamente como **métodos**, para distinguirlos de aquellos declarados fuera, que llamamos **rutinas**.

El primer paso para trabajar con objetos en un entorno POO es definir una clase. Su sintaxis es similar a la declaración de un tipo registro. El formato general para declarar un tipo clase es:

```
type TNombreClase = class
                      <declaración de campos>
                      <declaración de métodos>
                    end;
```

Donde `TNombreClase` es el identificador del nombre de la clase definida, `<declaración de campos>` tiene el mismo formato que la definición usual de variables y `<declaración de métodos>` es una cabecera de los procedimientos

y funciones necesarios. Por ejemplo, partiendo de nuestra idea de lo que es un coche, podríamos definir una clase `TCoche`, del siguiente modo:

```
type TCoche = class
            Marca: string;
            Modelo: string;
            Color: string;
            Matricula: string;
            procedure Arrancar;
            function verMatricula: string;
     end;
```

**NOTA**
La convención en Lazarus, al igual que en Delphi, es usar la letra `T`, de *Tipo*, como prefijo para el nombre de cada clase que se escribe.

Un **objeto** es una variable que tiene una clase como su tipo y se especifica con una declaración de la forma:

```
var Coche1: TCoche;
```

Se pueden declarar dos o más variables del mismo tipo (Figura 8.2):

```
var Coche1, Coche2: TCoche;
```

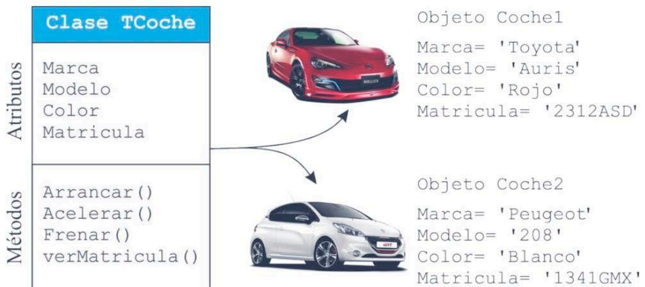
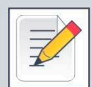



**Figura 8.2.** Dos objetos de una misma clase tienen los mismos atributos

Coche1 y Coche2 poseen los mismos atributos (campos) y tienen acceso a los mismos métodos (funciones y procedimientos). La declaración de estas dos variables creará dos áreas de memoria diferentes que almacenarán los valores de cada uno de los campos. Una vez creadas estas instancias de la clase TCoche, ¿cómo se comunica el programa con los objetos creados? Se hace de modo semejante a los registros:

```
NombreObjeto.NombreCampo
NombreObjeto.NombreMétodo
```

Por ejemplo, para acceder al campo Matricula del objeto Coche1, escribimos Coche1.Matricula, y para invocar el método VerMatricula de la instancia Coche2, escribimos Coche2.VerMatricula.



**IMPORTANTE**
Los métodos se definen en dos partes del programa: las cabeceras se especifican en la declaración del tipo de clase y la implementación (cabecera más cuerpo), fuera de la definición del tipo.

El acceso a los campos es también posible con la construcción **with do** que estudió con los registros:

```
with Coche1 do
  begin
    <atributos>
    <métodos>
  end;
```

Sin embargo, aunque válido en Pascal orientado a objetos, si intentamos ejecutar el código sin hacer nada más, se produciría un error de compilación, ya que se lanzaría una excepción de la clase **External**, consecuencia de un intento de acceso a una zona de memoria no asignada (Figura 8.3).

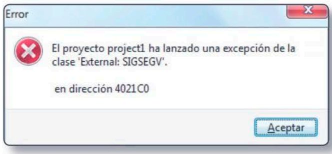



**Figura 8.3.** Error de compilación al acceder a una zona de memoria no asignada

La gestión de memoria de las clases es bastante diferente a la de los registros. La memoria que una variable tipo `registro` necesita es directamente reservada por el compilador en el momento de la declaración. Generalmente esa memoria se reserva en la pila, una región de memoria de acceso tipo LIFO (*Last In First Out*) gestionada por el compilador. De este modo, el programador no tiene que preocuparse de la gestión de memoria; sin embargo, con las clases no se da una gestión automática.

## 8.2 GESTIÓN DE MEMORIA

En Lazarus, como en la mayoría de los lenguajes POO, una variable de tipo `clase` no proporciona almacenamiento para el objeto, sino solo un puntero al objeto en memoria.

Así pues, nuestras variables `Coche1` y `Coche2` son punteros y, tras su declaración, sus valores se desconocen. Esto significa que aún no pueden usarse. Todas las clases necesitan tener asignada memoria en el área de memoria dinámica, que el compilador, con excepciones, no puede gestionar por sí mismo de forma automática.

Cada clase hereda siempre de la clase ancestro `TObject` dos métodos de gestión de memoria para reservarla cuando se crea y limpiarla cuando la clase se destruye. Estos métodos se llaman `Create` y `Destroy`.

Así que, antes de poder usar nuestras variables `Coche1` o `Coche2`, debemos reservar suficiente memoria:

```
begin
  Coche1 := TCoche.Create;
end;
```

La llamada a `Create` invoca a un constructor predefinido disponible para cada clase, a no ser que esta lo vuelva a definir.

**TRUCO**
Si mientras el cursor se encuentra sobre la definición de la clase, pulsa **[Ctrl]** + **[May]** + **[C]**, la función **Completar clase** del Editor de Lazarus añadirá automáticamente los cuerpos de los métodos, los métodos de acceso a las propiedades y variables y las variables privadas.

Una vez que el programa haya terminado de usar la instancia de clase a la que Coche1 o Coche2 apuntan, deberá liberar la memoria. Esto se hará con el destructor Free. En nuestro caso:

```
Coche1.Free;  // Se libera la memoria
```

## 8.3 ENCAPSULADO

Una clase puede contener cualquier cantidad de datos y cualquier número de métodos, sin embargo, una buena técnica en programación orientada a objetos consiste en ocultar los datos o encapsularlos dentro de la clase que los usa. Cuando se accede a una fecha, por ejemplo, no tiene demasiado sentido modificar tan solo el valor del día directamente. De hecho, si solo cambiamos este valor, podría resultar una fecha incorrecta, como el 31 de abril, por ejemplo. Para limitar este riesgo, empleamos los métodos para acceder a la representación interna de un objeto, pues los métodos pueden verificar si la fecha es válida. El encapsulado es importante pues permite que el programador que escribe la clase modifique la representación interna en cualquier versión posterior. La encapsulación, por tanto, garantiza la integridad de los datos que contiene un objeto al aislarlo del exterior.

Lazarus soporta el encapsulado clásico basado en módulos que usa la estructura de unidades, pero implementa también el encapsulado basado en clases. Todo identificador que se declare en la sección de interfaz de una unidad resultará visible a otras unidades del programa, siempre que se emplee una sentencia **uses** que se refiere a la unidad que define el identificador, como veremos en breve.

Observe el siguiente ejemplo. Los tres procedimientos declarados están encapsulados dentro de la clase TLibro:

```
type TLibro = class
            Autor, Titulo, ISBN: string;
            Precio: Currency;
           {Métodos asociados a Libro}
            procedure Inicio (A,T,I:string; PVP:Currency);
            procedure Leer;
            procedure Escribir;
           end;
```

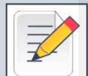
El programador nunca necesita acceder directamente a los atributos de un objeto de esta clase. En este caso, los objetos tendrán solo cuatro campos: Autor, Titulo, ISBN y Precio. Para acceder a ellos se han definido tres procedimientos, cada uno de los cuales puede alterar o informar el valor del campo.

### 8.3.1 Niveles de acceso a los campos

En el caso de un encapsulado basado en clases, Lazarus presenta **cinco niveles** o especificaciones de acceso, de los que solo estudiaremos los tres más importantes: **público**, **protegido** y **privado**:

- **La directiva `public`**. Denota campos y métodos a los que se puede acceder libremente desde cualquier parte del programa, así como en la unidad en la que se definen.
- **La directiva `protected`**. Se emplea para indicar campos y métodos con visibilidad limitada. Solo podrá acceder a los elementos la clase actual y sus clases heredadas.
- **La directiva `private`**. Denota campos y métodos de clase no accesibles fuera de la unidad que declara la clase.

Como norma general, los **campos** de una clase deberían ser **privados** y los **métodos**, **públicos**. Evidentemente, no siempre ha de ser así, es algo que el programador debe valorar con cada aplicación.



**IMPORTANTE**
Si dos clases se encuentran en la misma unidad o módulo, sus campos privados no estarán protegidos.

Veamos como ejemplo una nueva versión de la clase `TCoche`:

```
type
  TCoche = class
  private
    { declaraciones privadas }
      Marca, Modelo, Color: String;
  public
    { declaraciones públicas }
      procedure Arrancar;
      function VerMatricula: String;
  end;
```

Observe cómo hemos declarado como privados sus tres atributos y como públicos el procedimiento y la función (métodos).

Llegados a este punto, vamos a escribir nuestro primer programa empleando la extensión orientada a objetos del lenguaje Pascal. Abra un nuevo proyecto de Lazarus en modo consola, llámelo `coches` y ajuste la plantilla de código que le aparece en pantalla para que se parezca a lo siguiente:

```
program coches;
{$mode objfpc}{$H+}
type TCoche = class
     private
       Marca, Modelo, Color: string;
       Matricula: string;
     public
       procedure Iniciar;
       procedure MostrarInfo;
  end;

var Coche1: TCoche;
procedure TCoche.MostrarInfo;
begin
   WriteLn('Coche1', sLineBreak);
   WriteLn('Marca: ', Coche1.Marca, sLineBreak);
   WriteLn('Modelo: ', Coche1.Modelo, sLineBreak);
   WriteLn('Color: ', Coche1.Color, sLineBreak);
   WriteLn('Matricula: ', Coche1.Matricula,
          sLineBreak);
end;

procedure TCoche.Iniciar;
begin
    with Coche1 do
    begin
       Marca:= 'Toyota';
       Modelo:= 'Auris';
       Color:= 'Rojo';
       Matricula:= '2312ASD';
    end;
end;

begin
    Coche1:= TCoche.Create;  // Se crea la instancia
```

```
      Coche1.Iniciar;
      Coche1.MostrarInfo;
      Coche1.Free;    // Se libera la memoria
     {$IFDEF WINDOWS}
      ReadLn;
     {$ENDIF}
  end.
```

Una vez que lo ejecute, verá cómo por pantalla le aparecen los datos (`Marca, Modelo, Color y Matricula`) que ha definido para el objeto Coche1 en el procedimiento `Iniciar`.

Aunque el código puede optimizarse más, se ha escrito siguiendo el orden lógico que cualquier programador novato utilizaría. Lo importante de este primer programa es que nos permite pensar en los objetos como eje fundamental de su desarrollo.

Hay varios puntos que destacar en el ejemplo. Observe, en primer lugar, que para poder emplear las técnicas de la POO debe estar presente la directiva de compilación `{$mode objfpc}`, que le indica al compilador de Free Pascal que vamos a usar Pascal orientado a objetos como dialecto de Pascal. Por otro lado, fíjese en que hemos seguido la norma general de declarar los campos de la clase como privados (`Marca, Modelo, Color y Matricula`) y los métodos como públicos (`Iniciar y MostrarInfo`). El cuerpo principal del programa sigue la secuencia:

```
Iniciar;
MostrarInfo;
Free;
```

para el objeto `Coche1` de la clase `TCoche.` La memoria se reserva por el constructor Create y se libera por la llamada al destructor a través de Free.

La línea básica de trabajo en POO consiste en crear la instancia (objeto) con el constructor `Create`, usar el objeto para el fin elegido y liberarlo de la memoria cuando no sea necesario con una llamada a `Free`.

`TCoche` es básicamente una clase que contiene datos fijos relativos a cada vehículo, por lo que tendría sentido protegerlos a través de un encapsulado con propiedades, como veremos a continuación.

Lógicamente, estos datos relativos a las características del vehículo se obtendrían en un programa real mediante una consulta a una base de datos y no por asignación directa, como hemos hecho en este.

Sería ahora un buen momento para modificar el código y que el programa le mostrara los datos relativos a los dos vehículos presentados en la figura 8.2.

### 8.3.2 Encapsulado con propiedades

Las propiedades constituyen una forma muy curiosa de orientación a objetos. Básicamente, consiste en definir campos virtuales que ocultan completamente los datos de implementación, lo que permite modificar la clase sin que afecte al código que la emplea.

Desde la perspectiva del programador que define la clase, las propiedades actúan de manera similar a los atributos, pero con la diferencia de que el valor asignado a este atributo se asocia a un método (función o procedimiento).

La palabra reservada **property** se usa para definir la propiedad, tras la que se coloca su identificador, con el tipo de dato correspondiente. A continuación, se especifican los métodos asociados, que son una función para la lectura de los valores de la propiedad y un procedimiento para la escritura.

Por ejemplo, aquí tenemos la definición de una propiedad `Dia` para una clase de fecha:

```
property Dia: Integer read FDia write EstableceDia;
```

Para acceder al valor de la propiedad `Dia`, el programa lee el valor del campo privado `FDia`, mientras que para cambiar el valor de la propiedad llama al método `EstableceDia`, definido, lógicamente, dentro de la clase.

Las palabras reservadas **read** y **write** sirven para indicar qué método se usará para lectura o escritura de la propiedad. Si se omite la cláusula **write** en la propiedad, esta será solo de lectura.

**NOTA**
La convención en Lazarus es usar la letra `F`, de *Field* (campo), como prefijo para el nombre de cada campo privado que se declare en la clase.

La función que se usa para la lectura de una propiedad no debe tener parámetros y tiene que devolver un valor del mismo tipo de dato que el de la propiedad, mientras que el procedimiento para la escritura debe tener solo un parámetro, también del mismo tipo que el de la propiedad.

Generalmente, las propiedades siempre se crean en la sección pública de una clase (**`public`**), aunque sus métodos se deben implementar en la sección privada de aquella. Esto significa que hay que usar la propiedad para tener acceso a los métodos o datos.

Las propiedades dentro de las clases se declaran de una de estas dos formas:

1. Mediante acceso directo al campo privado, como en este ejemplo:

```
TClase = class
    Fcampo: TTipo;
    property Nombre: TTipo read FCampo write FCampo;
  end;
```

2. Mediante una función de lectura y otra de escritura, como en el siguiente ejemplo:

```
TClase = class
    Fcampo: TTipo;
    function GetData: TTipo;
    procedure SetData (var aDato: TTipo);
    property Nombre: TTipo read GetData write SetData;
end;
```

Como ejemplo práctico de encapsulado con propiedades, vamos a reescribir el programa `coches` con propiedades de solo lectura para manejar aquellos datos que no se van a modificar en el transcurso de la ejecución del programa. Abra un nuevo proyecto en Lazarus y guárdelo con el nombre `property_coches`. A continuación, modifique el código para que se parezca a esto:

```
program property_coches;
{$mode objfpc}{$H+}
type TCoche = class
   private
     FMarca, FModelo, FColor: string;
     FMatricula: string;
   public
```

```
    constructor Create(aMarca, aModelo, aColor,
                       aMatricula: string);
    destructor Destroy; override;
    procedure MostrarInfo;
    property Marca: string read FMarca;
    property Modelo: string read FModelo;
    property Color: string read FColor;
    property Matricula: string read FMatricula;
  end;

constructor TCoche.Create(aMarca, aModelo, aColor,
                          aMatricula: string);
begin
  inherited Create;
  FMarca:= aMarca;
  FModelo:= aModelo;
  FColor:= aColor;
  FMatricula:= aMatricula;
end;

destructor TCoche.Destroy;
begin
  inherited Destroy;
end;

procedure TCoche.MostrarInfo;
begin
   WriteLn('Coche1', sLineBreak);
   WriteLn('Marca: ', Marca, sLineBreak);
   WriteLn('Modelo: ', Modelo, sLineBreak);
   WriteLn('Color: ', Color, sLineBreak);
   WriteLn('Matricula: ', Matricula, sLineBreak);
end;

var Coche1: TCoche;
begin
  Coche1:= TCoche.Create('Toyota','Auris', 'Rojo',
                         '2312ASD');
  Coche1.MostrarInfo;
  Coche1.Free;
{$IFDEF WINDOWS}
  ReadLn;
{$ENDIF}
end.
```

Como los datos utilizados por el programa son solo de consulta, tiene mucho sentido protegerlos con un encapsulado con propiedades omitiendo la cláusula **`write`.** Observe que hemos declarado los campos como datos privados y los métodos y propiedades, así como el constructor y el destructor, en la sección pública de la clase `TCoche`.

En esta implementación hemos iniciado los datos a través del constructor mediante una lista de parámetros. En la definición del mismo llamamos primero al constructor heredado para gestionar correctamente la memoria y después lo completamos con los parámetros adecuados.

En la superclase `TObject` el destructor `Destroy` se declara como virtual. Esto significa que se puede volver a declarar un destructor con el mismo nombre en una clase descendiente con la palabra clave **`override`**. El compilador generará el código necesario para asegurar que se llame al destructor adecuado en tiempo de ejecución. Este método de usar un mismo método para realizar diferentes tareas en distintos niveles dentro de la jerarquía de clases es lo que se conoce como **polimorfismo**, que veremos en el capítulo siguiente.

## 8.4 DEFINICIÓN DE OBJETOS CON UNIDADES

La definición de un objeto dentro de una o más unidades proporciona un método adecuado para utilizarlos en diferentes programas. Cuando se define un objeto en una unidad, la sección privada del objeto está oculta al programa que utiliza la unidad.

Una **unidad**, en Lazarus, es como una biblioteca externa de funciones y procedimientos que se puede utilizar en cualquier programa haciendo mención de su nombre. Las unidades permiten lograr dos objetivos:

- La programación se hace más modular, lo que facilita la legibilidad y mantenimiento de los programas.
- Asegura la unicidad de métodos y funciones al evitar que coexistan distintas versiones de los mismos métodos.

Una unidad se define con la palabra reservada `unit` y tiene dos partes: una **pública,** a la que se podrá acceder sin problemas, y una **privada**, que implementa el desarrollo de la parte pública y a la que no se puede acceder desde el resto de programas. La parte pública se detalla con la palabra reservada **`interface`** y la privada con **implementation**.

Debajo de **`interface`** se declaran los nombres de los métodos que se quiera exportar, así como las variables o clases que se quiera crear. En la sección **`implementation`** se definen los métodos.

La estructura general para declarar una unidad es:

```
unit Nombre
interface // parte pública
  <declaración de métodos, variables y clases>
implementation // parte privada
  <definición de métodos>
end.
```

Una unidad es un módulo auxiliar, luego no puede ejecutarse sino a través de otro programa que la use. Veamos un ejemplo.

En Lazarus, abra un nuevo proyecto en modo consola. A continuación, vaya al menú **Archivo | Nueva unidad**. Verá que en el **Editor de código fuente** le aparecen dos pestañas: `project1.lpr` y `Unit1` (Figura 8.4).

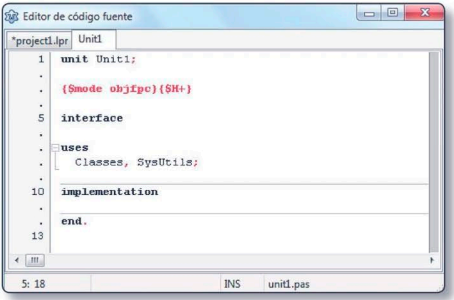



**Figura 8.4.** Editor de código fuente para un proyecto con una unidad

En el menú **Proyecto**, elija **Guardar proyecto como** y salve el archivo `project1.lpr` como `elipses.lpr` y `unit1.pas` como `elipse.pas`. A continuación, sitúese en la sección **`interface`** y modifique el código de la unidad para que se parezca a lo siguiente:

```
unit elipse;
{$mode objfpc}{$H+}
interface
uses SysUtils;     // para FormatFloat
type TElipse = class
  procedure Iniciar (a_, b_: Real);
  function Area: Real;
  function Perimetro: Real;
  function Excentricidad: Real;
  procedure Resultado;
end;
var a, b: Real;
implementation
procedure TElipse.Iniciar(a_, b_: Real);
begin
  a:= a_;  // semieje mayor
  b:= b_;  // semieje menor
end;

function TElipse.Area: Real;
begin
  Result:= Pi * a * b;
end;

function TElipse.Excentricidad: Real;
begin
  Result:= sqrt(a*a-b*b)/a;
end;

function TElipse.Perimetro: Real;
begin
  Result:= Pi * (3*(a+b)-sqrt((3*a+b)*(a+3*b)));
end;

procedure TElipse.Resultado;
begin
 Writeln ('Elipse con semiejes a: ',FormatFloat
   ('0.00', a),' y ', 'b: ',FormatFloat('0.00', b));
 WriteLn();
 WriteLn('Area: ', FormatFloat('0.00', Area));
 WriteLn('Perimetro: ', FormatFloat('0.00', Perimtro));
 WriteLn('Excentricidad: ', FormatFloat('0.00', Excentricidad));
end;
end.
```

Una vez implementada la unidad, vaya a la ficha `elipses.lpr` para escribir el programa principal.

### 8.4.1 Uso de las unidades

Para poder emplear las unidades en otros programas se hace uso de la palabra reservada **uses** al inicio del programa

Las unidades se ejecutan a través del programa que las utiliza, aunque sí se pueden compilar independientemente. Una vez que una unidad ha sido compilada no necesita compilarse más veces.

En aquellos casos en que la unidad se encuentre en un directorio diferente a aquel en el que se halla el programa que la usa, se puede indicar a través de una directiva del compilador en qué carpeta se encuentra la unidad. La directiva que se emplea es **{$UNITPATH ruta}**.

Aprovechando la unidad `elipse.pas` que acabamos de construir, vamos a escribir un programa que use esta unidad para hallar el área, el perímetro y la excentricidad de una elipse de semiejes conocidos.

En la ficha `elipses.lpr` escriba el siguiente código:

```
program elipses;
{$mode objfpc}{$H+}
uses elipse;  // se utiliza la unidad
var MiElipse: TElipse;
begin
  MiElipse.Iniciar (4,3); // se le pasan los semiejes
  MiElipse.Resultado;
  {$IFDEF WINDOWS}
    ReadLn;
  {$ENDIF}
end.
```

Dado que la definición del tipo `objeto` está dentro de la unidad que utiliza el programa, se puede invocar cualquier método del objeto dentro del programa principal.

Cuando haya finalizado de escribir la unidad y el programa, pulse **[F9]** para ejecutarlo. Si todo ha ido bien, debería ver por pantalla la siguiente salida:

```
Elipse con semiejes a: 4,00 y b: 3,00
Area: 37,70
Perimetro: 22,10
Excentricidad: 0,66
```

Las clases pueden llegar a ser realmente complejas, pero esa complejidad es tan solo el resultado de la propia técnica de diseño orientado a objetos, que permite la reutilización extensiva de código a través de clases heredadas y polimorfismo, que veremos con detalle en el siguiente capítulo.

# 9

# HERENCIA Y POLIMORFISMO

La herencia y el polimorfismo –junto con el encapsulamiento, la modularidad y la abstracción que representan las clases, y que ya se han estudiado– son las características esenciales de todo lenguaje de programación orientado a objetos. La herencia implica que las clases se pueden relacionar mediante una jerarquía, de modo que los objetos heredarán las propiedades y el comportamiento de la clase a la que pertenecen.

El **polimorfismo**, por el contrario, permite asignar a objetos distintos que comparten el mismo nombre comportamientos diferentes. Cuando se les llama, se utilizará el comportamiento correspondiente al objeto que se esté usando.

La **herencia** organiza y facilita, a su vez, este polimorfismo y posibilita definir y crear objetos como tipos especializados de objetos ya existentes. De modo que pueden compartir y extender su comportamiento sin tener que volver a implementarse.

## 9.1 LA HERENCIA

Como acabamos de decir, la **herencia** es una propiedad esencial de la programación orientada a objetos, pues permite crear nuevas clases a partir de otras ya existentes. Es la característica fundamental que distingue un lenguaje orientado a objetos, como el que usa Lazarus, de otros como C.

Posibilita a un objeto, por tanto, heredar propiedades de otra clase, de modo que aquel pueda contener sus propios métodos y heredar los de otros objetos.

Esta característica permite definir tipos objetos **descendientes** y **ascendientes** de un tipo objeto dado. Para definir un objeto descendiente basta con incluir el nombre del tipo ascendiente dentro de un paréntesis después de la palabra reservada `class`.

```
Nombre Tipo Descendiente = class (Nombre Ascendiente)
```

La propiedad de la herencia hace las tareas de programación mucho más fáciles y flexibles, ya que no es necesario escribir cada una de las características explícitamente para cada elemento. Por ejemplo, si se trata de representar información sobre `Animales`, se puede diferenciar entre `Mamíferos`, `Aves`, `Reptiles`, etc., que compartirán características comunes de `Animales`. Las `Aves` son descendientes del objeto `Animales`, y, por el contrario, `Animales` es ascendiente o antepasado de `Aves`. De igual modo, `Búho` es un descendiente de `Aves` y, en consecuencia, de `Animales` (Figura 9.1).

Cuando una **clase** deriva de otra recibe el nombre de **subclase**; a la vez, la clase raíz se denomina **superclase** (Figura 9.1).

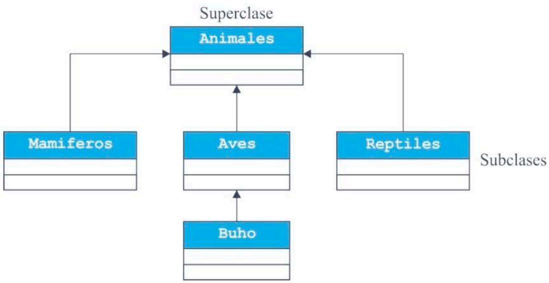



**Figura 9.1.** Herencia simple

Una **subclase** hereda todas las variables de instancias y métodos de la **superclase** y, a su vez, permite la definición de nuevos. La declaración de una subclase difiere de la de una clase en que el nombre de la clase raíz aparece entre paréntesis tras la palabra reservada `class`.

```
type subclase = class (superclase)
...
  end;
```

La herencia permite que una vez que una clase se ha depurado y probado, ya no necesite modificarse, pero puede definirse una nueva subclase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y quebraderos de cabeza, ya que solamente tiene que verificar la nueva conducta que proporciona la clase heredada.

La programación en entornos gráficos es un ejemplo muy ilustrativo de herencia. Los compiladores proporcionan librerías cuyas clases describen el aspecto y la conducta de las ventanas, controles y menús, por lo que se puede aprovechar la ingente cantidad y complejidad del código necesario para crear una interfaz gráfica de usuario añadiendo en la subclase el código necesario para ejecutar la aplicación definida.

En Lazarus todas las clases derivan implícitamente de la clase base `TObject`, por lo que heredan todos los métodos definidos en dicha clase.

### 9.1.1 La cláusula *inherited*

La palabra reservada **`inherited`** se utiliza en programación orientada a objetos para llamar, en la clase actual, al constructor o destructor del ascendiente. Es habitual, dentro de una buena práctica de programación, realizar esa llamada al comienzo de un constructor y al final de un destructor.

```
Create;
  begin
       inherited; // siempre al comienzo del constructor
       ...
  end;
Destroy;
begin
       ...
       inherited; // siempre al final del destructor
end;
```

Sin parámetros, **`inherited`** llama al mismo método de la clase ascendiente, y con los mismos argumentos.

Veamos un primer ejemplo para estudiar cómo funcionan la herencia de clases y la cláusula **`inherited`**. En Lazarus, abra un nuevo proyecto en modo

La herencia permite que una vez que una clase se ha depurado y probado, ya no necesite modificarse, pero puede definirse una nueva subclase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y quebraderos de cabeza, ya que solamente tiene que verificar la nueva conducta que proporciona la clase heredada.

La programación en entornos gráficos es un ejemplo muy ilustrativo de herencia. Los compiladores proporcionan librerías cuyas clases describen el aspecto y la conducta de las ventanas, controles y menús, por lo que se puede aprovechar la ingente cantidad y complejidad del código necesario para crear una interfaz gráfica de usuario añadiendo en la subclase el código necesario para ejecutar la aplicación definida.

En Lazarus todas las clases derivan implícitamente de la clase base `TObject`, por lo que heredan todos los métodos definidos en dicha clase.

### 9.1.1 La cláusula *inherited*

La palabra reservada **`inherited`** se utiliza en programación orientada a objetos para llamar, en la clase actual, al constructor o destructor del ascendiente. Es habitual, dentro de una buena práctica de programación, realizar esa llamada al comienzo de un constructor y al final de un destructor.

```
Create;
  begin
      inherited; // siempre al comienzo del constructor
      ...
  end;
Destroy;
begin
      ...
      inherited; // siempre al final del destructor
end;
```

Sin parámetros, **`inherited`** llama al mismo método de la clase ascendiente, y con los mismos argumentos.

Veamos un primer ejemplo para estudiar cómo funcionan la herencia de clases y la cláusula **`inherited`**. En Lazarus, abra un nuevo proyecto en modo

consola y guárdelo con el nombre **herencia**. Modifique a continuación la plantilla de código para que se parezca a lo siguiente:

```
program herencia;
{$mode objfpc}{$H+}
type
  // Define clase raíz, base de TObject por defecto
  TAnimal = class
     public
      Nombre : string;
      Constructor Create; overload;
      Constructor Create(aNombre : string); overload;
  end;
  // Define una subclase de TAnimal
  TAves = class(TAnimal)
     public
      Constructor Create(aNombre : string);
  end;
constructor TAnimal.Create; // Crea un objeto animal
begin
  // Ejecuta el constructor de TObject primero
  Inherited;
  Nombre := 'Animal';
end;
// Crea otro objeto animal
constructor TAnimal.Create(aNombre: string);
begin
  Inherited Create;
  Nombre := aNombre; // y guardamos su nombre
end;
// Crea un objeto ave
constructor TAves.Create(aNombre: string);
begin
  // Ejecuta el constructor del ancestro (TAnimal)
  Inherited Create(aNombre);
end;
var
  Animal  : TAnimal;
  insecto : TAnimal;
  buho    : TAves;
begin
  // Creamos 3 objetos animal
  animal  := TAnimal.Create;
  insecto := TAnimal.Create('Insecto');
```

```
    buho   := TAves.Create('Lechuza');
    // y vemos cuáles son animales
    if animal  Is TAnimal then WriteLn(animal.Nombre  +'
         es un animal');
    if insecto Is TAnimal then WriteLn(insecto.Nombre +'
         es un animal');
    if buho  Is TAnimal then WriteLn(buho.Nombre  +' es
         un animal');
    // y cuáles aves
    if animal  Is TAves then WriteLn(animal.Nombre   +'
         es un ave');
    if insecto Is TAves then WriteLn(insecto.Nombre  +'
         es un ave');
    if buho  Is TAves then WriteLn(buho.Nombre   +' es
         un ave');
     {$IFDEF WINDOWS}
       ReadLn;
     {$ENDIF}
  end.
```

Cuando compile y ejecute el programa, obtendrá una salida semejante a la siguiente:

```
Animal es un animal
Insecto es un animal
Lechuza es un animal
Lechuza es un ave
```

¿Cómo funciona el programa? En primer lugar, hemos definido la clase ancestro `TAnimal`, base de `TObject` para, a continuación, definir una subclase de aquella, `TAves`. De este modo, podremos crear después los objetos `animal` y `ave`. Para que los constructores resulten fáciles de recordar, los lenguajes de programación orientados a objetos suelen llamar a todos con el mismo nombre, lo que proporciona una importante sobrecarga. Al definir el constructor `Create` con el parámetro `aNombre`, se reemplaza la definición predeterminada por una nueva. Observe que en la clase `TAnimal` hemos declarado dos constructores `Create` distintos, uno sin parámetros, que oculta al predeterminado, y otro con valores iniciales. Fíjese cómo se ha llamado en primer lugar al constructor `Create` de la superclase `TObject` para después invocar a su método con la cláusula **`inherited`**. Para crear el objeto `ave` ejecutamos el constructor de su ancestro, `TAnimal`, y llamamos a su método con la misma cláusula. Por último, el programa comprueba de qué tipo son cada uno de los tres objetos animales creados: `animal`, `insecto` y `lechuza`. Lógicamente, `lechuza` es un ave y, por tanto, también un animal.

## 9.1.2 La cláusula *Self*

En cualquier método de un objeto existe siempre un parámetro oculto que de forma implícita se identifica con la variable `Self`.

En el cuerpo de un método, `Self` permite referirse al objeto. Esta propiedad se manifiesta de modo práctico cuando los identificadores de campos de un objeto tienen el mismo nombre que otro identificador del mismo ámbito. `Self` siempre se refiere a la instancia del objeto que hace la llamada del método. Por ejemplo, en el tipo objeto `TAnimal`

```
type
   TAnimal = class
      Nombre : string;
      Constructor Create; overload;
      Constructor Create(aNombre : string); overload;
  end;
```

dentro del método constructor `TAnimal.Create`, una asignación al campo `Nombre` del objeto se puede escribir como

```
Self.Nombre := 'Animal';
```

Veamos con un nuevo ejemplo cómo puede manejarse la cláusula `Self` para obtener información sobre las clases y su ubicación. Abra un nuevo proyecto de Lazarus y llámelo **self_herencia**. A continuación, aproveche el código fuente del programa herencia para que se parezca a esto:

```
program self_herencia;
{$mode objfpc}{$H+}
type
  TAnimal = class
       procedure MostrarInfo;
     end;
  TAves = class(TAnimal)
     end;
procedure TAnimal.MostrarInfo;
begin
  WriteLn;
  WriteLn('El nombre de la clase es ', self.ClassName);
  WriteLn('Su ancestro es ', self.ClassParent.ClassName);
  WriteLn('y esta declarada en ', self.UnitName);
```

```
end;
var clase: TAnimal;
begin
  clase:= TAnimal.Create;
  clase.MostrarInfo;
  clase.Free;

  clase:= TAves.Create;
  clase.MostrarInfo;
  clase.Free;
{$IFDEF WINDOWS}
  ReadLn;
{$ENDIF}
end.
```

Cuando ejecute el programa, obtendrá una salida semejante a esta:

```
El nombre de la clase es TAnimal
Su ancestro es TObject
y esta declarada en self_herencia

El nombre de la clase es TAves
Su ancestro es TAnimal
y esta declarada en self_herencia
```

En esta pequeña aplicación hay varios aspectos muy interesantes del comportamiento de las clases que es importante comentar. Las dos clases definidas (`TAnimal` y `TAves`) usan el mismo procedimiento `MostrarInfo`, aunque se ha declarado solo una vez en `TAnimal`. Esto es así porque la clase descendiente `TAves` hereda todos los métodos declarados en la clase ancestro. Otro detalle importante es que en el procedimiento declarado hemos hecho uso de la variable `Self`, que está disponible de forma implícita para todas las clases. Con ella nos referimos a cualquier instancia de la clase o inequívocamente a la propia clase. Los métodos `ClassName`, `ClassParent` y `UnitName` se encuentran implementados en la clase `TObject`, la más alta de la jerarquía de Lazarus, de la que pueden heredarse.

El mecanismo de la herencia, además, posibilita que podamos usar una única variable, `clase`, de tipo `TAnimal` para referirnos no solo a esa instancia, sino también a la instancia `TAves`. Las variables de la clase ancestro son compatibles en tipo con cualquiera de las descendientes.

## 9.2 EL POLIMORFISMO

Los de programación orientada a objetos son capaces de resolver llamadas a métodos y funciones no solo mediante enlaces estáticos, sino también con enlaces dinámicos. En este último caso, la dirección real del método se establece en tiempo de ejecución, según el tipo de instancia utilizada para hacer la llamada.

El **polimorfismo** y la **ligadura dinámica** de tipos son dos características específicas de la programación orientada a objetos. *Polimorfismo* significa que el mismo operador se puede utilizar con diferentes tipos de objetos, que responderán del modo apropiado a ese programador. Por ejemplo, el operador + es polimórfico, o sobrecargado, como ya vimos, ya que indica concatenación, suma o unión, en función de los operandos.

Esta técnica significa que se puede llamar a un método y aplicarlo a una variable, pero que el método al que realmente se llama depende del tipo de objeto con el que esté relacionada la variable. Lazarus no puede decidir la clase real del objeto al que se refiere la variable hasta estar en tiempo de ejecución, debido a la norma de compatibilidad de tipos. El polimorfismo, en su acepción más usual, se refiere a un método de un objeto que tendrá el mismo nombre que un método de su objeto ascendente. Esta característica, por tanto, permite escribir código más sencillo, tratar tipos de objetos distintos como si se tratara del mismo y conseguir el comportamiento adecuado en tiempo de ejecución.

Por ejemplo, supongamos que una clase y una clase heredada (`TAnimal` y `TAves`) definen ambas un nuevo método y que este tiene enlace posterior o dinámico. Se puede aplicar este método a una variable genérica como `miAnimal` que en tiempo de ejecución pueda referirse a un objeto de la clase `TAnimal` o a un objeto de la clase `TAve`. El método real al que se llama se determina en tiempo de ejecución, según la clase del objeto real.

En el ejemplo siguiente vamos a mostrar cómo funciona el polimorfismo de forma práctica. Las clases `TAnimal` y `TAves` tienen un método **`Voz`** que pretende reproducir la onomatopeya del animal. Este método se define como virtual (**`virtual`**) en la clase `TAnimal` y más tarde se sobrescribe (**`override`**) cuando se define la clase `TAves`:

```
type
  TAnimal = class
   public
       function Voz: string; virtual;
  TAves = class (TAnimal)
   public
       function Voz: string; override;
```

El hecho práctico más sobresaliente del polimorfismo reside en el hecho de que con enlace posterior se ejecute solo una versión del método, aunque luego las implementaciones sean diferentes. Para permitir que se pueda dar el polimorfismo es necesario hacer el **método** correspondiente **virtual** en lugar de un método estático, que es el sistema tradicional. El efecto de la llamada `miAnimal.Voz` puede entonces variar. Si la variable `miAnimal` se refiere en un instante dado a un objeto de la clase `TAnimal`, llamará al método `TAnimal.Voz` y sonará un gato en el ejemplo. Si se refiere a un objeto de la clase `TAves`, llamará al método `TAves.Voz` y oirá el sonido de un pato. Esto ocurre solo porque la función `Voz` se ha declarado como **`virtual`**.

La llamada a `miAnimal.Voz` funcionará para cualquier objeto que sea una instancia de cualquier descendiente de la clase `TAnimal`, aunque las clases estén definidas en otras unidades. El compilador no necesita conocer todos los descendientes para hacer una llamada compatible con ellos, solo requiere la clase ascendiente. Esta es la razón por la que los lenguajes POO favorecen la reutilización de código. Se puede escribir un código que emplee clases de una jerarquía sin conocer nada de las clases específicas que forman parte de dicha jerarquía.

Veamos su funcionamiento de un modo práctico. Abra un nuevo proyecto en modo consola en Lazarus, guárdelo con el nombre **`animales`** y escriba el siguiente código:

```
program animales;

{$mode objfpc}{$H+}

uses
  MMSystem;     // Para PlaySound
type
  TAnimal = class     // Ancestro
  public
    constructor Create;
    function LeerTipo: string;
    function Voz: string; virtual;
  private
    Clase: string;
  end;
  TAves = class (TAnimal)    // Descendiente
  public
    constructor Create;
    function Voz: string; override;
```

```
    end;
    constructor TAnimal.Create;
    begin
      Clase := 'un gato,';
    end;
    function TAnimal.LeerTipo: string;
    begin
      LeerTipo := Clase;
    end;
    function TAnimal.Voz: string;
    begin
      Voz := ' Miau Miau';
      PlaySound ('gato.wav', 0, snd_Async);
    end;

    constructor TAves.Create;
    begin
      Clase := 'un pato,';
    end;
    function TAves.Voz: string;
    begin
      Voz := ' Cuac Cuac';
      PlaySound ('pato.wav', 0, snd_Async);
    end;

  var miAnimal : TAnimal;
  begin
      miAnimal:= TAnimal.Create;    // Gato
      miAnimal.Voz;
      WriteLn('Ahora suena ' + miAnimal.LeerTipo +
              miAnimal.Voz);
      WriteLn('Pulse una tecla...');
      miAnimal.Free;
      ReadLn;
      miAnimal:= TAves.Create;  // Pato
      miAnimal.Voz;
      WriteLn('y ahora ' + miAnimal.LeerTipo +
              miAnimal.Voz);
      miAnimal.Free;
      ReadLn;
  end.
```

Antes de compilar y ejecutar la aplicación, asegúrese de que ha copiado los archivos `gato.wav` y `pato.wav` en la misma ruta donde se encuentra el programa. Tras ejecutarlo, oirá los sonidos correspondientes producidos por la llamada a `PlaySound`, a saber, los de un gato y un pato.

Como acaba de ver, para **sobrescribir** un **método** con enlace posterior en una clase descendiente, es necesario emplear la palabra clave **`override`**. Solo puede utilizarse, por supuesto, si se definió el **método** como **`virtual`** (dinámico) en la clase ascendiente.

Las normas son muy sencillas. Un **método** definido como **estático** sigue siendo estático en todas sus subclases, a no ser que se oculte con un nuevo método virtual que tenga el mismo nombre. Un **método virtual** sigue manteniendo el enlace posterior de cada subclase.

Para sobrescribir un método virtual, hay que especificar los mismos parámetros y usar la palabra clave **`override`**:

```
type
  TMiClase = class
      procedure Uno; virtual;
      procedure Dos;  // método estático
  end;
  TMiClaseDerivada = class (TMiClase)
      procedure Uno; override;
      procedure Dos;
  end;
```

Existen dos formas básicas de **sobrescribir** un método. La primera consiste en reemplazar el método de la clase ascendiente por una nueva versión. La segunda, en añadir nuevo código al método existente, para lo que se emplea la palabra clave **`inherited`**, que llama al mismo método de la clase ascendiente. Por ejemplo, podemos escribir:

```
procedure TMiClaseDerivada.Uno;
begin
    // Código nuevo
    ...
    // Llama al procedimiento MiClase.Uno
  inherited Uno;
end;
```

Cuando se sobrescribe un método virtual existente en una clase básica, hay que usar los mismos parámetros. Cuando se presenta una nueva versión de un método en una clase descendiente, sin embargo, se puede declarar con cualquier parámetro. De hecho, este será un nuevo método independiente del método ascendiente del mismo nombre, solo que tendrá el mismo nombre. Por ejemplo:

```
type
  TMiClase = class
    procedure Uno;
  end;
  TMiClaseDerivada = class (TMiClase)
    procedure Uno (S: string);
  end;
```

Cuando se crea el objeto de la clase `TMiClaseDerivada`, se puede usar el método definido como `Uno` con el parámetro de cadena, pero no la versión sin parámetros definida en la clase ascendiente `TMiClase`. Si se necesita esto último, se puede marcar el método de la clase derivada con la palabra clave **`overload`**:

```
TMiClaseDerivada = class (TMiClase)
  procedure Uno (S: string); overload;
end;
```

Si el método tiene parámetros diferentes a los de la versión de la clase ascendiente, se convierte, efectivamente, en un método sobrecargado.

### 9.2.1 Métodos virtuales, dinámicos y abstractos

En Lazarus existen dos formas distintas de activar el enlace posterior. Se puede declarar el método como **virtual** con la palabra clave **`virtual`**, como ya hemos visto; o como **dinámico**, con la palabra clave **`dynamic`**. Su estructura es exactamente la misma y el resultado de su uso también. Lo único que cambia es el mecanismo interno empleado por el compilador para implementar el enlace posterior.

Los **métodos virtuales** se basan en una tabla de métodos virtuales VTM. Estas tablas permiten que las llamadas a métodos se ejecuten rápidamente, pero se necesita una entrada para cada método virtual de cada clase descendiente.

Las llamadas a un **método dinámico**, por otro lado, se realizan usando un número único que indica el método, que se guarda en una clase solo si esta lo define o sobrescribe. La búsqueda de la función es más lenta, pero la ventaja es que las entradas del método dinámico solo se propagan a las descendientes cuando estos sobrescriben el método.

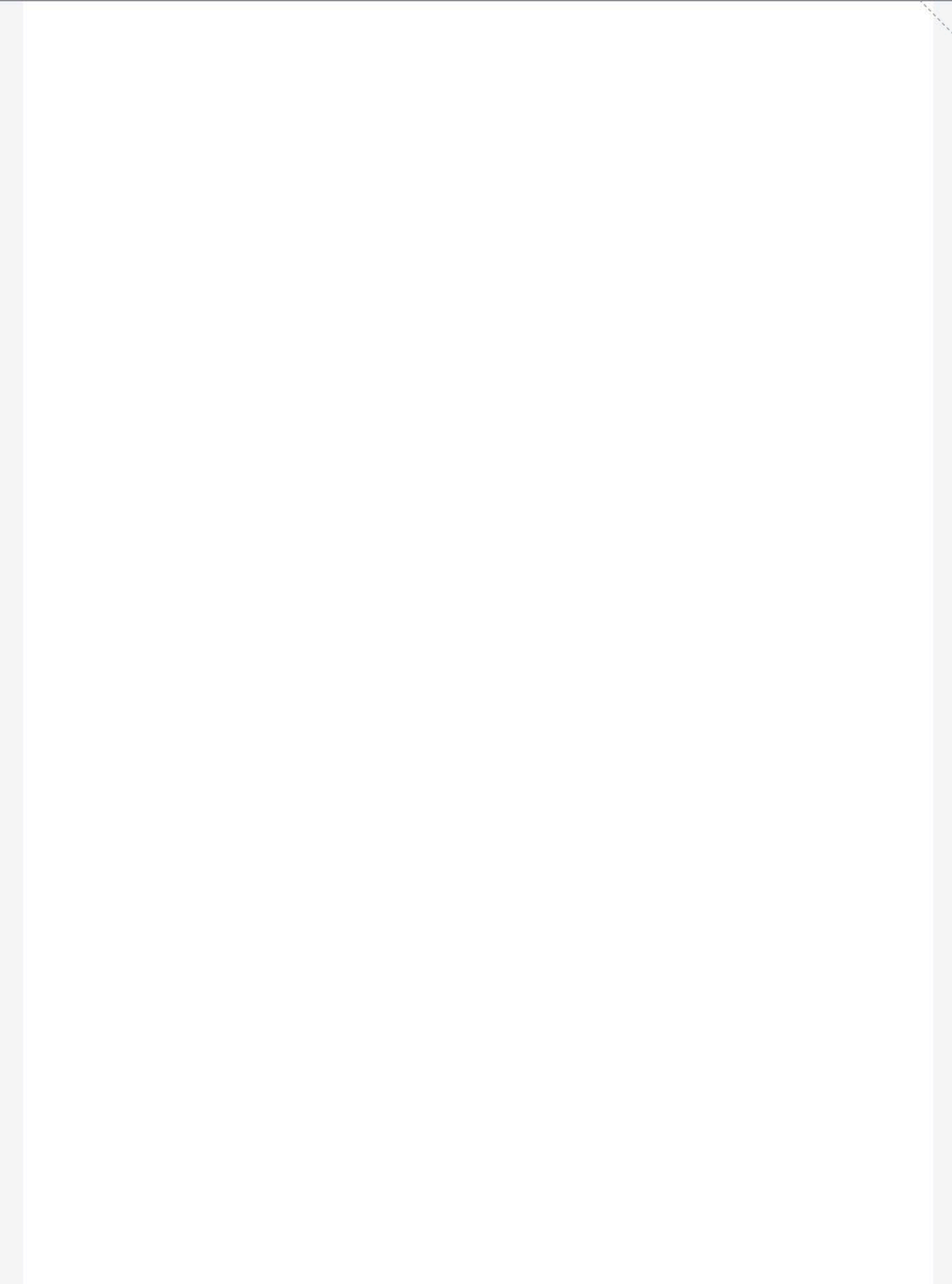
La palabra clave `abstract` se emplea para declarar **métodos** que van a definirse solo en **subclases** de la clase actual. La directiva `abstract` define por completo el método. Si se intenta definir el método, el compilador nos informará de ello con un mensaje de advertencia. En Lazarus se pueden crear instancias de clases que tengan métodos abstractos, sin embargo, si se llama a un método abstracto en tiempo de ejecución, Lazarus creará una excepción.

**IMPORTANTE**
Generalmente, no se pueden crear instancias de clases que contengan métodos abstractos.

## 9.2.2 Polimorfismo multiplataforma

El polimorfismo actúa como casi la única solución a la casuística que los programadores pueden encontrar a la hora de implementar software en distintas plataformas. Se trata de una imposición artificial a la simplicidad y aparente uniformidad de lo que en realidad es un cuadro mucho más complejo. La complejidad no se elimina, pero sí se enmascara.

El gran número de fabricantes de dispositivos electrónicos para los que se programan aplicaciones, junto con el de desarrolladores de sistemas operativos que han ido apareciendo en el mercado en los últimos 30 años, ha obligado a Lazarus, como solución multiplataforma, a imponer una estandarización en la programación de sus librerías para que cada componente que use el programador en las aplicaciones en modo gráfico se comporte de forma idéntica en cada plataforma, sea Windows, Linux, Unix, etc. No tendrán el mismo aspecto en pantalla, claro está, porque Lazarus intenta respetar la apariencia de cada interfaz de usuario empleando sus propios *widgets*, pero sí se comportarán igual. Esta es la gran baza de Lazarus, como reza su eslogan: “Escribe una vez, compila en cualquier sitio”.



# 10

# LAZARUS Y LA PROGRAMACIÓN VISUAL

Hasta ahora hemos utilizado la potencia de Lazarus para desarrollar aplicaciones en modo consola. En esta segunda parte de la obra comenzaremos a usar sus capacidades para el desarrollo rápido de aplicaciones en modo gráfico. La programación visual permite a los lenguajes de programación diseñar y desarrollar aplicaciones con un entorno visual amigable y fácil de utilizar para el usuario.

Lazarus permite crear **interfaces gráficas de usuario** (GUI) mediante un conjunto de imágenes y objetos gráficos que representan la información y acciones disponibles en la interfaz. Su principal uso consiste en proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de una máquina. Habitualmente, las acciones se realizan mediante manipulación directa, para facilitar la interacción del usuario con la computadora.

La **biblioteca** de **componentes gráficos** de Lazarus, conocida como LCL, consiste en una colección de módulos que provee los componentes y clases necesarios para las tareas visuales. Se basa en dos librerías de Free Pascal: la **librería** de **componentes** (FCL) y la **biblioteca** en **tiempo** de **ejecución** (RTL). Muchos de sus módulos, clases e identificadores poseen los mismos nombres y trabajan de igual forma que en Delphi, por lo que usualmente el código de Delphi se puede portar directamente a Lazarus.

De momento, Lazarus carece de soporte .NET, pero pueden programarse todo tipo de aplicaciones gráficas, servicios, demonios de Linux, bibliotecas, aplicaciones de bases de datos y aplicaciones web en general.

Los componentes visuales de la librería de Lazarus (LCL) se rigen por eventos, lo que significa que no es necesario escribir completamente y de forma secuencial toda la aplicación gráfica. El estado de los componentes de la biblioteca se determina únicamente por sus propiedades. Pueden modificarse en la forma deseada y colocarse visualmente en cualquier posición en el entorno de desarrollo integrado de Lazarus.

Cuando un componente nuevo se sitúa en una ventana o formulario, se genera una variable en el código fuente y, si existe algún evento asociado al componente, aparecerá también un método nuevo. Lazarus permite arrastrar componentes, programar las acciones y compilar la aplicación. A diferencia de C o C++, el compilador que acompaña a Lazarus se encarga de buscar todos los ficheros necesarios para compilar el código principal.

## 10.1 LA ESTRUCTURA DE UN PROGRAMA GRÁFICO

Al igual que hicimos con los programas en modo consola, la forma más sencilla de ver cuál es el esqueleto de cualquier aplicación en modo gráfico es pidiéndole a Lazarus que abra un proyecto gráfico.

Vaya al menú **Proyecto | Nuevo Proyecto** y elija la opción **Aplicación** (Figura 10.1). Esta opción le permitirá implementar un programa gráfico LCL/Free Pascal, cuyo código fuente será mantenido automáticamente por Lazarus.

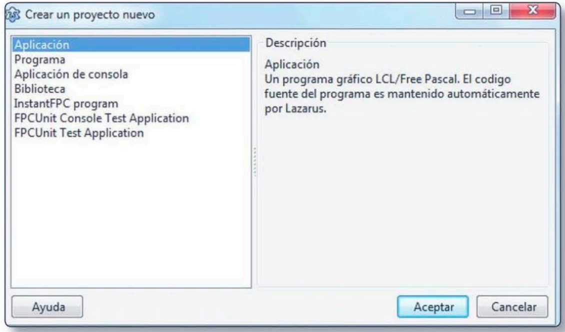



**Figura 10.1.** Creación de un programa en modo gráfico

Cuando pulse el botón **Aceptar**, Lazarus le mostrará en el **Editor** el esqueleto de una unidad de formulario, a la que por defecto llama `Unit1` y que resulta esencial para cualquier aplicación gráfica.

Toda aplicación visual necesita al menos dos ficheros Pascal como parte del programa:

- El **programa principal**, al que por defecto Lazarus le da el nombre de `project1.lpr`.
- **Una** o **varias unidades**, de nombre `nombre_unidad.pas` o `nombre_unidad.pp`.

Las unidades en programación visual serán, a su vez, de dos tipos:

- Ficheros de **código fuente** en Pascal, aplicables tanto a programas en modo consola como a aplicaciones gráficas, donde se implementa el programa.
- Ficheros de **formulario**, irrelevantes en las aplicaciones en modo consola, pero necesarios en las aplicaciones gráficas.

Por defecto, cuando abre una aplicación en modo gráfico, Lazarus le obsequia en el **Editor** de código con la estructura de una unidad de formulario. Su esqueleto es el siguiente:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, Forms, Controls,
     Graphics, Dialogs;
type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.lfm}
end.
```

Esta unidad, a la que Lazarus denomina `*Unit1`, va asociada en el área de trabajo a un formulario o ventana de nombre `Form1`. Si no viera el formulario, puede que se encuentre oculto tras alguna ventana. Pulse entonces **[F12]** para que aquel pase al primer plano.

Lazarus ha escrito todo el código del esqueleto de la unidad por usted. Ha declarado una clase `TForm1` heredada de `TForm` para el formulario y una variable global de tipo `TForm1` a la que llama `Form1`. La instancia de clase se crea por la llamada

```
Application.CreateForm (TForm1, Form1)
```

del programa principal `project1.lpr`, lo que mostrará el formulario `Form1` en tiempo de diseño según el código del editor y las propiedades fijadas en el Inspector de objetos, como veremos en unos instantes.

Lazarus también ha escrito una generosa cláusula **`uses`** en previsión de que se necesiten las siete unidades LCL para escribir el programa gráfico. Aunque no use todas, es bueno dejarlas si prevé emplear la característica de compleción automática de código de Lazarus. El compilador ignorará aquellas unidades que no se empleen y tampoco las enlazará en el ejecutable final.

Abra ahora el **Inspector de proyecto** a través del menú **Proyecto | Inspector de proyecto** y haga doble clic sobre el archivo llamado `project1.lpr` para que Lazarus cargue el programa principal en el Editor de código, que le mostrará ahora dos pestañas: `*Unit1` y `project1.lpr` (Figura 10.2).

El programa, de momento, solo existe en memoria hasta que no se guarde en disco. Cree una nueva carpeta llamada `capitulo10` y seleccione **Proyecto | Guardar proyecto como...** En el primer cuadro de diálogo, dé al proyecto el nombre **`primerGUI.lpi`** y nombre la unidad `unit1.pas` en el segundo cuadro de diálogo como **`ppal.pas`**.

**TRUCO**
Otra posibilidad para abrir el fichero principal es seleccionar **Proyecto | Ver fuente proyecto**.

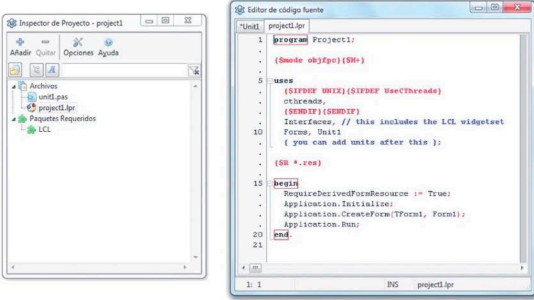



**Figura 10.2.** Archivos de un proyecto Lazarus en modo gráfico

Si observa ahora el contenido de la carpeta `capitulo10` verá que, además del subdirectorio `backup`, Lazarus ha creado siete archivos:

```
ppal.lfm
ppal.pas
primerGUI.ico
primerGUI.lpi
primerGUI.lpr
primerGUI.lps
primerGUI.res
```

El programa principal, `primerGUI.lpr`, presenta en el editor la siguiente estructura:

```
program primerGUI;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, ppal
  { you can add units after this };
{$R *.res}
begin
  RequireDerivedFormResource := True;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

La cláusula **uses** cita más componentes que los que se empleaban en los programas en modo consola: `cthreads` (en sistemas Unix), `Interfaces`, `Forms` y la unidad de formulario `ppal` que acaba de nombrar. El bloque principal del programa, entre las palabras **begin** y **end**, emplea dos variables globales predefinidas en la LCL: `RequireDerivedFormResource`, que se fija en `True`, y `Application`, una instancia de clase que representa al proyecto mismo.

Tras inicializar la instancia `Application`, el programa crea y muestra la ventana que hemos llamado `formulario` y que Lazarus representa como `Form1`. La directiva **{$R *.res}** indica a Lazarus que, durante la compilación y enlazado, genere un fichero `.res` que contenga el icono asociado al programa y la información que se haya especificado en las **Opciones del proyecto**.

Si compila en este punto el proyecto, en el que aún no hemos escrito ni una línea de código, observará que ocupa cerca de 15 MB, debido a todo el código que Lazarus incluye a través de las bibliotecas requeridas. Aunque el programa "no haga nada" –lo que no es cierto, pues puede arrastrar la ventana, modificar sus dimensiones y cerrarla– ocupa muchísimo por todo ese código necesario para el diseño y comportamiento del formulario. La ventaja es que cuando se escribe código, el tamaño del archivo crece ya muy despacio.

## 10.2 EL INSPECTOR DE PROYECTO

El **Inspector de proyecto** suministra al programador acceso inmediato a todos los ficheros del proyecto y permite **añadir** archivos, no necesariamente de Pascal, como ficheros de imágenes o de bases de datos, por ejemplo. También desde él se pueden **quitar** ficheros que ya no sean necesarios en un proyecto y acceder a las **opciones** del mismo a través de los iconos correspondientes del Inspector (Figura 10.3).

La parte inferior del Inspector de proyecto es una lista en la que de forma jerárquica se recogen todos los ficheros Pascal del proyecto y sus dependencias. Lazarus ya incluye por defecto como paquete requerido la librería LCL, necesaria para cualquier proyecto gráfico (Figura 10.3).

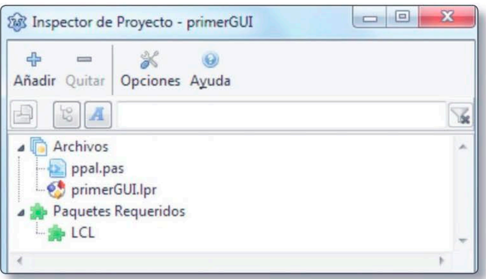



**Figura 10.3.** Inspector de proyecto de Lazarus

La mayor parte de la funcionalidad que se explota en las aplicaciones visuales se ha codificado conjuntamente por los equipos del compilador de Free Pascal y de Lazarus, y almacenado en la LCL y en sus dependencias. Los programas gráficos no podrán compilar sin ella porque su funcionamiento depende del código presente en la librería de componentes.

Otro punto a favor de Lazarus es que extiende el concepto de unidad o módulo a los paquetes y proyectos. Un **paquete**, para Lazarus, es una colección de ficheros relacionados por su código fuente, que incluye información sobre dónde puede localizar los distintos ficheros, cómo deben compilarse y si son necesarios otros. Puede ser una librería completa o un módulo que encapsula cierta funcionalidad para un proyecto. Los paquetes tienen por extensión `.lpk` y están precompilados, por lo que si no se han modificado, pueden usarse sin necesidad de compilarse de nuevo para ahorrar tiempo de desarrollo.

## 10.3 EL EDITOR DE CÓDIGO FUENTE

El **Editor de código fuente** es la herramienta más integrada en el entorno de desarrollo de Lazarus y es, quizás, su punto más sobresaliente, superando incluso en muchos aspectos las capacidades del editor de Delphi.

Todas las funciones pueden tener asociados atajos de teclado, que se configuran en el menú **Herramientas | Opciones | Editor | Atajos de teclado**. Los atajos consisten en una combinación de una o dos teclas junto con la combinación deseada de teclas modificadoras, que, en Windows, son **[Ctrl]**, **[May]** y **[Alt]**.

### 10.3.1 Navegación rápida

El Editor de Lazarus permite navegar rápidamente por el código fuente mediante la combinación de teclas **[Ctrl]** + **[May]** + ↑ y **[Ctrl]** + **[May]** + ↓. Su uso permite saltar en los sentidos indicados por las flechas entre la declaración e implementación de un procedimiento o función. Por ejemplo, observe el siguiente fragmento de código:

```
10   type
11     TForm1 = class(TForm)
12       Button1: TButton;
13       procedure Button1Click(Sender: TObject); // de aquí
14     private
15       { private declarations }
16     public
17       { public declarations }
18     end;
19   var
20     Form1: TForm1;

22   implementation

24   {$R *.lfm}

26   procedure TForm1.Button1Click(Sender: TObject);
27   begin
28             //  hasta aquí, o viceversa
29   end;
```

Si se sitúa en cualquier punto de la línea 13, donde se declara el procedimiento, y pulsa la combinación de teclas **[Ctrl]** + **[May]** + ↓, automáticamente el cursor se situará en la línea 28, justo en el cuerpo de la definición del método. Si en este punto pulsamos la combinación **[Ctrl]** + **[May]** + ↑, Lazarus subirá el cursor a la declaración, de nuevo a la línea 13 del ejemplo. En caso de que no se encuentre el procedimiento buscado, automáticamente el cursor se posiciona en la primera diferencia que encuentre en el procedimiento más parecido.

La función **Buscar declaración** desde cursor (**[Alt]** + ↑) busca la declaración de un identificador y siempre salta a la declaración más cercana. Por ejemplo, si sitúa el cursor sobre `TForm1` de la línea 26 y pulsa esta combinación de teclas, el cursor se desplazará al principio de `TForm1` de la línea 11.

## 10.3.2 Compleción automática de código

El IDE de Lazarus presenta dos opciones muy interesantes para completar código de forma automática y facilitar la labor a los programadores y desarrolladores. La primera de las funciones es **Completado de identificador**, a la que se accede con la combinación **[Ctrl] + [Espacio]**. Esta función muestra una lista de propiedades o métodos de clase en función del nombre o parte del nombre tecleado. Por ejemplo, si teclea entre las palabras reservadas **begin** y **end** del editor de código el fragmento de instrucción `showm`, y sin mover el cursor pulsa **[Ctrl] + [Espacio]**, le aparecerá una ventana emergente con la lista mostrada en la figura 10.4.

**Figura 10.4.** Completado de identificador con las opciones posibles

La lista se va filtrando a medida que escribe el nombre del método, aunque también puede usar el ratón o las flechas del teclado para desplazarse por ella hasta seleccionar el adecuado. Desplácese hasta `ShowMessage` y pulse **[Enter]** para insertar la selección en el editor, como sigue:

```
begin
  ShowMessage();
end;
```

Esta función de Lazarus funciona con todos los identificadores declarados en todas las unidades que se recogen en la cláusula **uses**. Por ello, aunque no use todas las que aparecen en el esqueleto de la unidad cuando abre el proyecto, es bueno dejarlas para que esta y otras funciones de ayuda puedan trabajar rápidamente en la compleción de código. En el caso que acabamos de comentar, el procedimiento `ShowMessage` está declarado en la unidad `Dialogs`. Si la elimina de la cláusula **uses**, la función Completado de identificador no encontrará `ShowMessage`.

La segunda función interesante de Lazarus es **Completar código**, a la que se accede con la combinación de teclas **[Ctrl] + [May] + [C]**. En función de la posición del cursor, el efecto de esta llamada será diferente. Si el cursor se encuentra en la sección de declaración de una clase, se activa la compleción de la clase. En

este caso, Lazarus intenta completar la declaración y generar implementaciones vacías para todos los métodos de la declaración. En este proceso, el IDE reconoce heurísticamente qué debería figurar en los campos y métodos.

Por ejemplo, suponga que ha escrito la siguiente declaración para una clase `TMiComponente`, heredada de `TComponente`:

```
type
  TMiComponente = class(TComponente)
public
property Matricula: String Read FMatricula write
    SetMatricula;
end;
```

Si sitúa el cursor dentro de la declaración de clase y llama a la función **Completar código**, Lazarus modificará todo el código de forma automática de la siguiente manera:

```
type
  { TMiComponente }
  TMiComponente = class(TComponente)
  private
    FMatricula: String;
    procedure SetMatricula(AValue: String);

  public
    property Matricula: String Read FMatricula write
      SetMatricula;
end;

{ TMiComponente }
procedure TMiComponente.SetMatricula(AValue: String);
begin
  if FMatricula = AValue then Exit;
  FMatricula := AValue;
end;
```

Fíjese en que el programa ha reconocido `SetMatricula` como un método y `FMatricula` como un campo privado. Además, crea el cuerpo del método `SetMatricula` por nosotros y lo completa con el código inicial. Lógicamente, Lazarus no sabe qué va a hacer el programador, pero siempre completará todos los campos, métodos y propiedades que encuentre.

La operación inversa también es posible. Por ejemplo, si el programador añade un nuevo procedimiento a la implementación de la clase:

```
procedure TMiComponente.CheckMatricula;
begin
  if Matricula = '' then RaiseException;
end;
```

Y sitúa el cursor en cualquier lugar dentro del cuerpo del procedimiento, cuando llame a la función Completar código, Lazarus añadirá automáticamente en la sección privada de la declaración de la clase el procedimiento **`procedure`** `CheckMatricula`.

## 10.4 EL PRIMER PROGRAMA GRÁFICO

Aprovechando el proyecto `primerGUI` creado en el punto 10.1, sitúe el foco sobre la ventana del formulario `Form1` y pulse el icono del **botón** de la pestaña *Standard* de la **Paleta de componentes** (el cuarto por la izquierda con el texto **Ok**) para seleccionarlo (Figura 10.5).

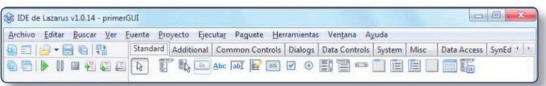



**Figura 10.5.** Ventana de proyecto con la Paleta de componentes de Lazarus

Ahora haga clic sobre cualquier zona del formulario para que Lazarus cree en ese punto un nuevo control de botón de nombre `Button1`. Si hace doble clic sobre él, el IDE generará un manejador en el editor para el evento `OnClick` en la unidad a la que dimos el nombre de `ppal.pas`. El cursor se situará automáticamente en la implementación del método que Lazarus ha llamado `TForm1.Button1Click()`:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Aquí sitúa el cursor }
end;
```

Escriba ahora la orden `ShowMessage('Hola Mundo');` en el cuerpo del procedimiento y compile y ejecute el programa con **[F9]**. Cuando pulse el botón `Button1` aparecerá un cuadro de diálogo con el texto "`Hola Mundo`" (Figura 10.6).

**Figura 10.6.** Ejecución del programa primerGUI sobre Windows

Aprovechando este ejemplo, vamos ahora a modificar ligeramente el código fuente para trabajar con fechas. Vuelva al editor y añada en la sección privada de la clase `TForm1` una función **`Hoy`**, de tipo **`String`**:

```
function Hoy: string;
```

Sitúe el cursor en cualquier punto de la declaración de esta función y pulse **[Ctrl] + [May] + [C]** para que Lazarus construya el cuerpo de la misma en la sección de implementación de la unidad. Añada ahora el código necesario, como sigue:

```
function TForm1.Hoy: string;
begin
  Result := FormatDateTime('dddd, d mm yyyy', Now);
end;
```

Desplácese ahora hasta el procedimiento `TForm1.Button1Click` y modifíquelo para que se parezca a esto:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Ahora := Time;
 ShowMessage('Hoy es ' + Hoy + ', ' +
TimeToStr(Ahora));
end;
```

Tan solo falta un pequeño detalle: declarar la variable global `Ahora`, de tipo `TDateTime`. Pulse **[F9]** para compilar y ejecutar el programa. Si todo ha ido bien, obtendrá una salida como la mostrada en la figura 10.7:

**Figura 10.7.** Ejecución del programa modificado

## 10.4.1 Modificar el título e icono del programa

Ya que tiene hecho su primer programa bajo un entorno gráfico, vamos a enseñarle a modificar el icono que el explorador del sistema operativo le muestra cuando lista el programa o lo selecciona.

Por defecto, Lazarus le asignó como icono una huella de guepardo, , por si no se había dado cuenta hasta ahora. Para cambiarlo, acceda al menú **Proyecto | Opciones del proyecto** o pulse la combinación de teclas **[Ctrl] + [May] + [F11]**. En la sección **Configuraciones de la aplicación** de la ventana emergente, verá el título y la imagen del icono asociado a su proyecto (Figura 10.8).

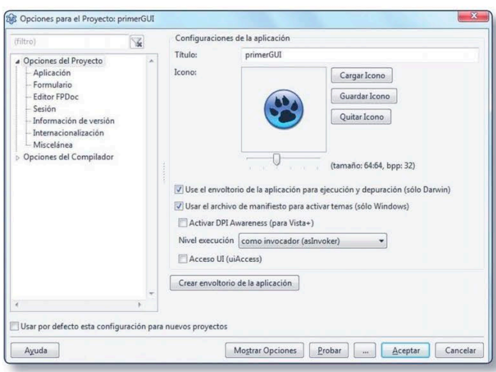



**Figura 10.8.** Opciones para el proyecto

Si lo desea, puede escribir un nuevo título para el proyecto, establecido ahora como `primerGUI` y cargar un nuevo icono desde el botón **Cargar icono**. Si la resolución del icono lo permite, podrá cambiar también su tamaño desde 16 × 16 hasta 256 × 256 con el deslizador bajo la imagen.

## 10.5 EL INSPECTOR DE OBJETOS

Lazarus le ofrece dos formas de asignar valores a las propiedades de los controles que arrastra desde la Paleta de componentes al formulario. El primer método consiste en escribir el código Pascal correspondiente al nombre del control y la propiedad que se desea modificar. Por ejemplo, aprovechando nuestro primer programa, cambiaríamos la leyenda del botón del formulario escribiendo esto:

```
Button1.Caption:= 'Mostrar Fecha';
```

La segunda opción es emplear el **Inspector de objetos** (IO), una ventana de Lazarus desde la que se pueden no solo ver, sino también modificar la mayoría de propiedades y eventos de un componente. Las propiedades y eventos se listan en la mitad izquierda del IO en orden alfabético. La mitad derecha de la ventana es editable y contiene el valor actual de cada propiedad o evento.

Abra el proyecto `primerGUI` y seleccione en el formulario el botón llamado `Button1` haciendo clic sobre él. Active la pestaña **Propiedades** del IO, si no lo está ya, y desplácese hasta la propiedad `Caption`. Escriba en ella el valor `Mostrar Fecha`. Automáticamente, el botón cambiará su leyenda (Figura 10.9).

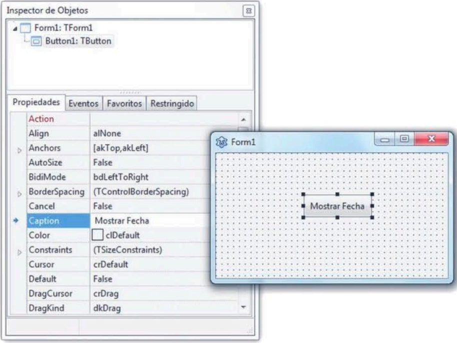



**Figura 10.9.** Modificación de la propiedad Caption del control botón

Ahora tiene un mejor aspecto, pues cualquier usuario sabrá lo que hace el botón: mostrar una fecha. Si dejamos como etiqueta del botón la asignada por Lazarus, `Button1`, poco le diría a nadie, ni siquiera al programador cuando tiempo después tenga que volver al código.

A medida que diseña un proyecto gráfico, Lazarus va añadiendo controles y nombres por defecto, pues de alguna manera han de llamarse y referenciarse. Sin embargo, al igual que le conminamos al comienzo del libro a que usara nombres que indicaran algo cuando definiera variables, funciones o procedimientos, volvemos a hacer lo mismo cuando se trate de controles en un formulario.

Intente ahora editar otros valores en el Inspector de objetos para ver los cambios sobre su programa. Seleccione el botón, suba hasta la propiedad `AutoSize`, por ejemplo, y cambie su valor a `True`. Ahora las dimensiones del botón se ajustan automáticamente al tamaño de su etiqueta. La altura se ha fijado en 25 píxeles, así que deje de nuevo en `False` aquella y baje hasta la propiedad `Height` para poner el nuevo valor en 30. Baje ahora hasta la propiedad `Name` y cambie su valor por defecto a `btnFecha`. Automáticamente, `Button1` será sustituido en el código fuente por su nuevo nombre, mucho más fácil a la hora de analizar el código fuente.

Finalmente, pulse fuera del botón para seleccionar el formulario y expanda la propiedad `BorderIcons` picando sobre la flecha ▸ o signo + junto a esta. Cambie los atributos `biMaximize` y `biMinimize` a `False` para que la barra de título de la ventana de su programa solo muestre el icono de **cerrar aplicación** ☒.

### 10.5.1 La pestaña Restringido

Aunque Lazarus permite compilar proyectos para decenas de plataformas, no existe una estandarización completa entre los distintos sistemas operativos. Esto significa que algunas propiedades o características de un determinado control no estarán disponibles para todas las plataformas. El Inspector de objetos tiene una pestaña de restricciones a la que llama **Restringido** y que recoge todas las limitaciones.

Seleccione el botón de su formulario y abra la pestaña **Restringido**. Observará que la propiedad Color presenta restricciones en Windows y Mac Os. Abra el menú **Color**, elija cualquiera y pulse **[Enter]** para confirmar la acción. Si usa Windows, verá cómo no se producirá ningún cambio (Figura 10.10).

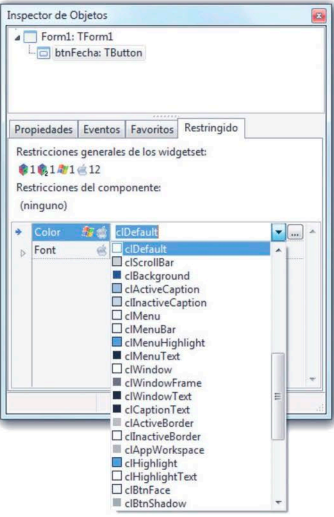



**Figura 10.10.** Pestaña de restricciones del Inspector de objetos

Elija lo que elija, el color del botón en Windows no cambiará, sin embargo, si corre el programa en Linux o Mac, el botón tendrá el color que usted haya seleccionado. Estas restricciones, no obstante, no son impuestas por la librería de componentes de Lazarus, sino por el propio sistema operativo. Tenga en cuenta estas restricciones cuando programe aplicaciones que deban ejecutarse en distintos sistemas operativos.

## 10.6 LA PALETA DE COMPONENTES

La **Paleta de componentes** es una barra de herramientas con pestañas situada en la ventana de proyecto que posee un gran número de iconos que representan los componentes utilizados para construir formularios. En la versión 1.0.14 se recogen catorce pestañas: *Standard*, *Additional*, *Common Controls*, *Dialogs*, *Data Controls*, *System*, *Misc*, *Data Access*, *SynEdit*, *LazControls*, *RTL*, *IPro*, *Chart* y *SQLdb* (Figura 10.11). En total aportan más de 200 componentes, visuales o no, con funcionalidades específicas.

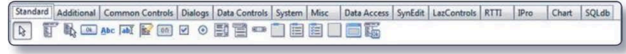



**Figura 10.11.** Paleta de componentes en Lazarus 1.0

Cada hoja muestra un conjunto diferente de iconos, que representa a un grupo funcional de componentes. Si deja el cursor del ratón inmóvil sobre cualquier componente de la paleta, sin pulsar sobre el icono, aparecerá el rótulo de ese componente. Advierta que cada rótulo empieza por la letra `T`, de *tipo de componente*.

Cuando selecciona un componente para incluirlo en un formulario, la clase se añade a la sección **`type`** de la sección **`interface`** de la unidad, normalmente como parte del `TForm1`, y se añade una instancia de esa clase a la sección **`var`**, generalmente como la variable `Form1`.

Un **componente**, en su definición más simple, no es más que un objeto descendiente del tipo `TComponent`, clase que proporciona las características de todo componente. Todos los componentes forman parte de la jerarquía de objetos de la biblioteca de componentes de Lazarus (LCL). Los componentes son la piedra angular de la programación en Lazarus. En vez de tener que operar en unidades, el programador simplemente tiene que hacer clic en el control y situarlo en la posición deseada del formulario. Eso es todo, Lazarus se encarga de lo demás.

Entre los más de 200 componentes de la paleta que pueden usarse, encontrar el adecuado es, en ocasiones, una tarea nada sencilla. Por razones históricas, sobre todo por compatibilidad con Delphi, la disposición de los controles en las diferentes pestañas de la paleta no es de gran ayuda. Para evitar tener que ver una a una todas las pestañas hasta localizar el componente deseado, Lazarus posee un buscador de componentes al que se puede acceder desde el menú **Ver | Lista de componentes** o mediante el atajo **[Ctrl] + [Alt] + [P]**. El filtro de búsqueda permite acortar la localización del control deseado. A medida que teclee una cadena, la lista de componentes se irá reduciendo (Figura 10.12).

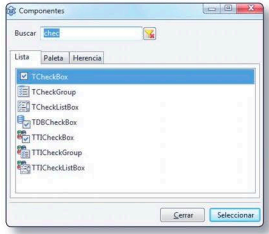



**Figura 10.12.** Lista de los componentes que contienen check

El buscador **Componentes** presenta tres hojas: **Lista**, **Paleta** y **Herencia**. La primera de ellas muestra los componentes por orden alfabético. **Paleta** lista los componentes según la pestaña de la Paleta de componentes en la que se encuentren. **Herencia**, por último, muestra la jerarquía de clases para cada componente y, salvo que vaya a dedicarse a escribir componentes, nunca la usará.

Si hace doble clic sobre cualquiera de los componentes del listado, Lazarus le insertará ese control en el formulario en la esquina superior izquierda, desde donde podrá arrastrarlo a la localización deseada.

Si solo resalta el componente con un clic, como es el caso de la casilla de verificación `TCheckBox` en la figura anterior, y pulsa el botón **Seleccionar**, se cerrará la ventana de diálogo **Componentes** y Lazarus le seleccionará el control elegido en la pestaña correspondiente de la Paleta de componentes.

Si observa con algo más de detalle los componentes de la paleta, verá que parece haber duplicidades con los mismos. Así, si busca en la ventana de diálogo **Componentes** el control *ventana de edición* (*edit*), verá que aparecen `TEdit`, `TTIEdit` y `TDBEdit`, en las paletas *Standard*, *RTTI* (*RunTime Type Information*) y *Data Controls*, respectivamente. Lo mismo ocurre con el componente *casilla de verificación*. Además de `TCheckBox` en la pestaña *Standard*, existen dos más: `TDBCheckBox` y `TTICheckBox`, en las pestañas *Data Controls* y *RTTI*, respectivamente. De modo que para cada componente básico existe a menudo otra versión en las pestañas *Data Controls* y *RTTI*.

La única diferencia entre ellos es que cada uno está especializado en una labor específica. Todos los componentes de la pestaña *Data Controls* son similares a los que aparecen en la pestaña *Standard*, salvo que poseen propiedades que permiten el acceso a bases de datos.

Lazarus permite a ciertos componentes interactuar con ficheros `.dbf` y `.csv`, y conectar con bases de datos como PostgreSQL, MySQL, Oracle Database, Microsoft SQL Server y Firebird, entre otras.

Los componentes de la pestaña *RTTI* son también semejantes a los de la hoja *Standard*, pero permiten obtener información de las clases en tiempo de ejecución.

# 11

# CONTROLES VISUALES

Ya sabemos que Lazarus incluye una gran cantidad de funciones y procedimientos, pero la auténtica potencia de la programación visual en Lazarus reside en la enorme biblioteca de componentes que proporciona.

Los componentes de Lazarus pueden utilizarse completamente desde el código o desde el diseñador visual de formularios. Algunas de ellas son clases componentes y aparecerán en la Paleta de componentes, otras son de propósito más general.

Los componentes son los elementos centrales de las aplicaciones Lazarus. Cuando se escribe un programa, básicamente, se debe escoger un cierto número de componentes, definir sus interacciones… y hecho.

Esto es realmente de lo que trata Lazarus. La programación visual mediante componentes es una característica clave de este entorno de desarrollo. Lazarus incluye una gran cantidad de componentes listos para usar. No vamos a describir cada componente con detalle, con sus propiedades y métodos; sería inacabable y muy tedioso. La intención de este capítulo y de los siguientes es mostrar cómo se usan algunas de las características básicas que ofrecen los componentes predefinidos de Lazarus para construir aplicaciones, y comentar técnicas específicas de programación.

Para empezar, analizaremos la clase básica `TControl`. Después, examinaremos los diversos componentes visuales para escoger los controles correctos que ayuden a realizar un proyecto más rápidamente.

## 11.1 LA CLASE *TCONTROL*

Una de las subclases más importantes de `TComponent` es `TControl`, que se corresponde con los componentes visuales. Esta clase básica define conceptos generales, como la posición y tamaño del control y el control padre que lo contiene, entre otros.

Existen dos subclases básicas:

- **Los controles basados en ventanas**. Son componentes visuales basados en una ventana del sistema operativo. Desde el punto de vista del usuario, los controles basados en una ventana pueden recibir el foco de entrada y algunos pueden contener otros controles. Este es el mayor grupo de componentes de la biblioteca de Lazarus.

- **Los controles gráficos**. Son componentes visuales que no se basan en una ventana del sistema operativo. Por lo tanto, no tienen manejador, no pueden recibir el foco y no pueden contener otros controles.

Algunas de las propiedades introducidas por `TControl` y comunes a todos los controles son aquellas relacionadas con el tamaño y la posición. La posición de un control la fijan sus propiedades `Left` y `Top`, y su tamaño las propiedades `Height` y `Width`. Técnicamente, todos los componentes tienen una posición, porque cuando abrimos de nuevo un formulario existente en tiempo de diseño, queremos que se puedan ver los iconos de los componentes no visuales en la posición exacta en la que los situamos. Esta posición es visible en el archivo de formulario.

Una característica importante de la posición de un componente es que, como cualquier otra coordenada, siempre se relaciona con la zona de cliente de su componente padre, indicada por su propiedad `Parent`. En el caso de un formulario, la zona del cliente es la superficie incluida dentro de sus bordes y la etiqueta. Sin embargo, fíjese en que las coordenadas de un control siempre son relativas al control padre, como un formulario u otro componente contenedor. Si se coloca un panel en un formulario y un botón en el panel, las coordenadas del botón son relativas al panel y no al formulario que contiene el panel. En este caso, el componente padre del botón es el panel.

Técnicamente, los componentes son subclases de la clase `TComponent`, que es una de las clases raíz de la jerarquía, como muestra la figura 11.1. Todos los componentes visuales descienden de `TControl` y algunos de `TWinControl` (Figura 11.1).

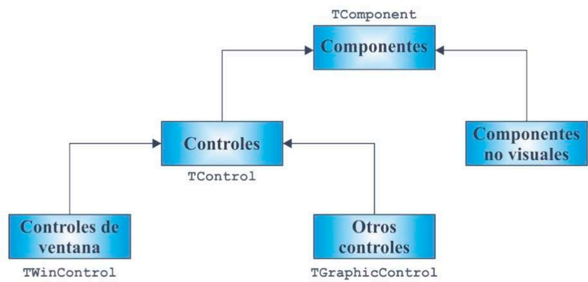



**Figura 11.1.** Representación de los principales grupos de componentes de la LCL

Además de los componentes, la biblioteca LCL incluye clases que heredan directamente de `TObject` y de `TPersistent`.

Las clases componentes pueden dividirse, además, en dos grupos principales: controles y componentes no visuales.

- **Controles**. Son todas las clases que descienden de `TControl`. Tienen una posición y tamaño en pantalla y aparecen en el formulario en tiempo de diseño en la misma posición que tendrán en tiempo de ejecución. Los controles tienen dos especificaciones diferentes, basados en ventanas o gráficos.

- **Componentes no visuales**. Son todos los componentes que no son controles. Descienden de `TComponent` pero no de `TControl`. En tiempo de diseño, un componente no visual aparece en el formulario o módulo de datos como un icono. En tiempo de ejecución, algunos de estos componentes pueden resultar visibles y otros están visibles siempre.

## 11.2 EL NOMBRE DE LOS COMPONENTES

Cada componente en Lazarus debe tener un nombre único dentro del componente propietario, que, por lo general, es el formulario en el que se coloca el componente. Esto significa que una aplicación puede tener dos formularios diferentes, cada uno con un componente con el mismo nombre.

El nombre del componente se establece con la propiedad **`Name`**. Como le venimos recordando a lo largo del libro, use un nombre lo suficiente descriptivo para saber de qué se trata y evitar confusiones entre los diferentes objetos. Normalmente, los programadores asociamos al nombre de un componente un prefijo con el tipo de componente: `btn` para `TButton`, `chk` para `TCheckBox`, `lbl` para `Tlabel`, etc. Esto hace que el código resulte más fácil de leer y permite que Lazarus agrupe los componentes en el Inspector de objetos, donde se clasifican por nombre.

Existen tres elementos importantes relacionados con la propiedad `Name` de los componentes. En primer lugar, en tiempo de diseño, el valor de la propiedad `Name` se usa para definir el nombre del campo de formulario en la declaración de la clase del formulario. Este es el nombre que normalmente se va a usar en el código para referirse al objeto. Por esa razón, el valor de la propiedad ha de ser un identificador válido de lenguaje Pascal. Segundo, si se establece la propiedad `Name` de un control antes de modificar sus propiedades `Caption` o `Text`, el nuevo nombre se copia normalmente en el título. Es decir, si el nombre y el título son idénticos, entonces, al cambiar el nombre, también cambiará el título. Por último, Lazarus usa el nombre del componente para crear el nombre predefinido de los métodos relacionados con estos eventos. Si tenemos un componente `Button1`, el controlador predefinido del evento `OnClick` se llamará `Button1Click`, a no ser que se especifique un nombre diferente. Si más tarde se cambia el nombre del componente, Lazarus modificará los nombres de los métodos relacionados en función de ello. Por ejemplo, si se cambia el nombre del botón a `btnFecha`, el método `Button1Click` se transforma automáticamente en `btnFechaClick`.

## 11.3 PROPIEDADES DE ACTIVACIÓN Y VISIBILIDAD

Se pueden usar dos propiedades básicas para dejar que el usuario active u oculte un componente. La más sencilla es la propiedad **`Enabled`**. Cuando se desactiva un componente, `Enabled` se define como `False`, normalmente hay alguna pista visual que se lo indica al usuario. En tiempo de diseño, la propiedad desactivada no siempre provoca un efecto, pero, en tiempo de ejecución, los componentes están, por lo general, en gris.

Para ver una técnica más drástica, se puede ocultar completamente un componente, ya sea utilizando el correspondiente método **`Hide`** o definiendo su propiedad **`Visible`** como `False`. Sin embargo, hay que tener en cuenta que leer el estado de la propiedad `Visible` no indica si el control es realmente visible. En realidad, si el contenedor de un control está oculto, incluso aunque el control este configurado como `Visible`, no se puede ver.

## 11.4 EVENTOS

En realidad, los componentes de Lazarus se programan usando propiedades, métodos y eventos. Aunque a estas alturas ya conoce qué son métodos y propiedades, los eventos todavía no se han comentado. La razón es que los eventos no implican una nueva función del lenguaje, sino que son una técnica estándar de programación. Un **evento**, de hecho, es técnicamente una propiedad, con la única diferencia de que se refiere a un método, un tipo de puntero a método, para ser precisos, en lugar de a otros tipos de datos.

Cuando un usuario interactúa con un componente, como al hacer clic sobre él, el componente genera un evento. Técnicamente, la mayoría de los eventos en Lazarus se desencadenan al recibir el mensaje correspondiente del sistema operativo, aunque no existe un mensaje individual para cada evento. En Lazarus, el controlador de eventos de un componente es normalmente un método del formulario que contiene el componente, no del propio componente. En otras palabras, el componente confía en su propietario para controlar eventos. Esta técnica se denomina **delegación** y resulta básica para el modelo basado en componentes de Lazarus.

Cuando se añade un controlador de **`OnClick`** para un botón, Lazarus hace exactamente eso. El botón tiene una propiedad de tipo de puntero a método, llamada `OnClick`, y se le puede asignar directa o indirectamente un método de otro objeto, como un formulario. Cuando un usuario hace clic sobre el botón, se ejecuta este método, aunque lo hayamos definido dentro de otra clase.

Otro concepto importante que ya hemos mencionado es que los eventos son propiedades. Esto significa que para controlar un evento de un componente, se asigna un método a la propiedad de evento correspondiente. Cuando hacemos doble clic sobre un valor de evento en el Inspector de objetos, se añade un nuevo método al formulario propietario y se asigna a la propiedad de evento correcta del componente.

Esta es la razón por la cual es posible compartir el mismo controlador de eventos para diversos eventos o cambiar un controlador de eventos en tiempo de ejecución. Para utilizar esta característica no se necesita mucho conocimiento sobre el lenguaje. De hecho, cuando se selecciona un evento en el Inspector de objetos, se puede pulsar el botón de **flecha** situado a la derecha del nombre del evento para ver una lista desplegable de métodos compatibles. Al usar el Inspector de objetos, es fácil seleccionar el mismo método para el mismo evento de diferentes componentes o para diferentes eventos compatibles del mismo componente.

## 11.5 LOS COMPONENTES DE VISUALIZACIÓN

Los componentes visuales son, lógicamente, los controles más llamativos de la programación visual. Algunos se han diseñado exclusivamente para la visualización de algún tipo de información; otros se emplean para edición, como entrada de texto; y algunos más, como selección de opciones.

### 11.5.1 *TLabel*

La etiqueta **`TLabel`** es un control Abc de la pestaña *Standard* de la paleta de componentes que sirve para sobreimprimir texto estático no seleccionable por el usuario sobre la ventana del formulario. Únicamente muestra texto en una o varias líneas, con una única fuente y color. Es muy útil para indicarle cosas al usuario o para describir la función de otros componentes de la aplicación.

Las propiedades más interesantes de `TLabel`, a las que podemos acceder desde el Inspector de objetos, son las siguientes:

- **`Alignment`**. Determina si el texto de la etiqueta se dispone a la izquierda, a la derecha o se centra sobre esta. Si la propiedad `AutoSize` es `True`, este efecto queda enmascarado.

- **`AutoSize`**. Establece si la etiqueta ajusta su tamaño a todo el texto que contiene. Si se establece en `False` y el texto es demasiado largo, este se truncará.

- **`Caption`**. Es el texto que muestra la etiqueta. Si uno de los caracteres se precede por el símbolo &, aparecerá subrayado y actuará como acceso rápido si se pulsa la tecla **[Alt]** y ese carácter. El foco se enviará entonces al control especificado en la propiedad `FocusControl`.

- **`FocusControl`**. Es el control que recibe el foco cuando se pulse la tecla rápida.

- **`Layout`**. Gobierna la alineación vertical, indicando si el texto se sitúa en la parte superior, inferior o centrado en la dimensión vertical. Si la propiedad `Autosize` es `True`, carece de efecto.

- **`ShowAccelChar`**. Indica si el símbolo & delante de un carácter representa una tecla de acceso rápido.

- **`Transparent`**. Si es `True`, el color de fondo de la etiqueta será transparente.

- **`WordWrap`**. Si es `True` cortará un texto demasiado largo para la etiqueta en varias líneas. Si la propiedad `AutoSize` está activa, esta propiedad no surtirá efecto alguno.

Para habituarse al uso de `TLabel`, abra una nueva aplicación gráfica en Lazarus de nombre **`etiquetas`** y renombre `unit1` como `main`, por ejemplo. Asegúrese de que el diseñador visual de formularios esté visible (pulse **[F12]** en caso contrario) y arrastre sobre `Form1` dos botones y una etiqueta desde la pestaña *Standard* de la Paleta de componentes. Seleccione uno de los botones y póngale como leyenda `Cerrar` y como nombre `btnCerrar`. El otro botón deberá tener por leyenda `Foco` (Figura 11.2). Seleccione la etiqueta y desplácese en el Inspector de objetos hasta la propiedad `FocusControl`. La lista desplegable tendrá las opciones `(ninguno)`, `btnCerrar`, `Button1`. Elija `btnCerrar`. Para que el foco pase a este botón, necesitamos definir en la etiqueta una tecla de acceso rápido. En la propiedad `Caption` de `TLabel`, escriba `&Cerrar` y pulse **[Enter]**. Ahora en el formulario verá el texto `Cerrar`. Revise que la propiedad `ShowAccelChar` sea `True` (Figura 11.2).

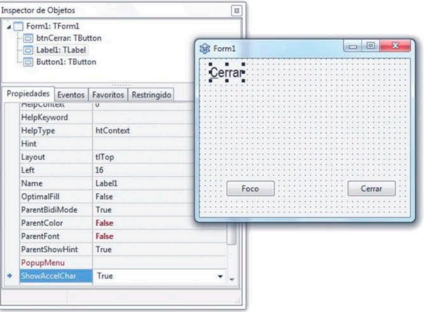



**Figura 11.2.** Etiqueta y dos botones en el diseño de la aplicación etiquetas

Establezca la propiedad `TabOrder` de `Button1` (`Foco`) a `0` y compruebe que la de `btnCerrar` (`Cerrar`) sea `1` y que en ambos botones la propiedad `TabStop` esté activada. Con estas opciones nos aseguramos de que cuando se ejecute la aplicación sea el botón `Foco` el que reciba el foco.

Haga doble clic sobre el botón **`Cerrar`** y complete el procedimiento en el Editor de código fuente como sigue:

```
procedure TForm1.btnCerrarClick(Sender: TObject);
begin
   Close();
end;
```

Pulse **[F9]** para compilar el programa. Verá que el botón que presenta el foco es el homónimo. Si pulsa la **barra espaciadora** observará cómo aquel se pulsa, aunque no haga nada. Con la tecla **[Tab]** podrá pasar el foco de uno a otro botón. Así mismo, aunque la etiqueta no pueda recibir el foco, sí puede pasarlo al botón `Cerrar` con la combinación **[Alt] + [C]**. Haga clic sobre este para cerrar el programa y regresar al Inspector de objetos.

Seleccione ahora la etiqueta y busque en el Inspector de objetos las propiedades `Left` y `Top`. Arrastre el componente a lo largo del formulario. Observe que ambos valores se están actualizando continuamente para reflejar la nueva posición del control. El IO sincroniza constantemente estos valores con el diseñador visual de formularios y con el fichero `main.lfm`.

### 11.5.2 *TStaticText*

El control `TStaticText` es un componente de la pestaña *Additional* de la paleta de componentes que permite también sobreimprimir texto estático en el formulario. No obstante, posee algunas diferencias con respecto a `TLabel`. `TStaticText` sí puede recibir el foco, puede participar en el orden de tabulación del formulario y puede servir como contenedor para otros controles; aunque carece de las propiedades `Layout` y `WordWrap` de `TLabel`. El control `TStaticText` posee un borde físico identificable con la propiedad `BorderStyle`. En la figura 11.3 puede observar dos controles `TStaticText` alineados en la parte superior e inferior del formulario.

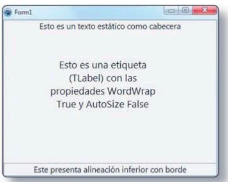



**Figura 11.3.** Etiqueta entre dos controles TStaticText

### 11.5.3 *TBevel*

`TBevel` y `TDividerBevel` son componentes meramente cosméticos en un formulario y se emplean para agrupar otros controles. Un **bisel**, representado por el icono de la pestaña *Additional* de la Paleta de componentes, es un marco rectangular vacío. Sus propiedades más importantes son `Shape` y `Style`. `Shape` admite las formas `BottomLine`, `Box`, `Frame`, `LeftLine`, `RightLine`, `Spacer` y `TopLine`. Con ellas podrá elegir qué lados del marco o rectángulo desea remarcar. `Style` admite las opciones `Raised` y `Lowered`.

`TDividerBevel` es un componente similar, localizado en la pestaña *LazControls* de la Paleta de componentes. Forma una línea horizontal que admite un texto a través de su propiedad `Caption`, por lo que suele usarse para nombrar una sección del formulario. Observe en la siguiente figura un par de usos de estos componentes.

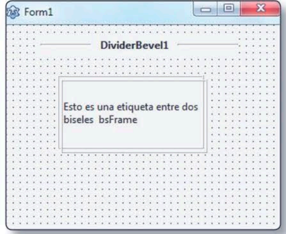



**Figura 11.4.** Dos usos de TBevel y TDividerBevel

### 11.5.4 *TStatusBar*

Concluiremos la sección correspondiente a los componentes de visualización con un control que actúa de forma similar en la mayoría de sistemas operativos: la barra de estado. Este control se representa con el icono en la pestaña *Common Controls*.

Abra una nueva aplicación y dele el nombre **`barradestado.lpr`**, con una unidad llamada **`barradestado_form.pas`**. Asigne al formulario el nombre `Barra de Estado` y póngale una anchura de `400`. Arrastre una barra de estado al formulario. Observará que de forma automática se sitúa en la parte inferior del mismo. Llámela `BarraDeEstado` a través de su propiedad `Name`.

Por defecto, la propiedad `SimplePanel` se encuentra activada. Edite la propiedad `Panels` y añada cuatro paneles, como muestra la figura 11.5.

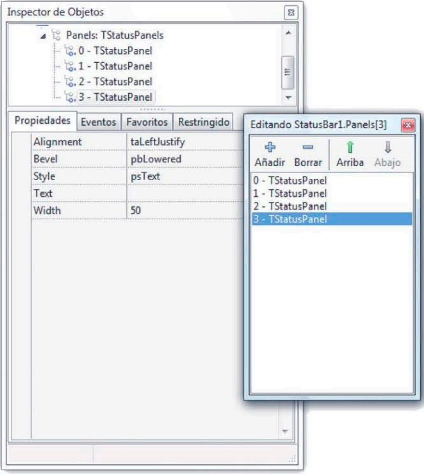



**Figura 11.5.** Adición de cuatro paneles a la barra de estado

Seleccione en el Inspector de objetos con la tecla **[Ctrl]** los dos primeros paneles, que se identifican por los números `0` y `1`. En la pestaña **Propiedades** del IO ponga una anchura de `110`. Asigne del mismo modo a los paneles `2` y `3` una anchura de `90` píxeles.

Añada ahora un conjunto de controles desde la Paleta de componentes: `TLabel`, `TButton`, `TEdit`, `TStaticText` y `TButtonPanel`. Use **[Ctrl]** + **[Alt]** + **[P]** para localizar los componentes si no recuerda dónde están. Seleccione ahora uno a uno estos cinco controles que ha arrastrado al formulario manteniendo apretada la tecla **[May]** entre uno y otro.

Haga clic con el ratón sobre la pestaña **Eventos** en el IO para que le muestre qué eventos hay comunes a esos cinco controles. Sitúese a la derecha del evento `OnMouseMove` y haga doble clic con el ratón para generar el manejador común a los controles. Lazarus declarará un procedimiento `MouseMove` con el nombre del

primer control que seleccionó y situará el cursor en la parte correspondiente del editor de código.

Para acortar el nombre que por defecto ha elegido Lazarus, acceda a la pestaña **Eventos** del IO y edite el nombre para dejarlo solo como `MouseMove`. Complete entonces en el editor el código del procedimiento como sigue:

```
procedure TForm1.MouseMove(Sender: TObject; Shift:
                    TShiftState; X, Y: Integer);
var ctrl: TControl;
begin
   BarraDeEstado.Panels[0].Text:= Sender.ClassName;
   ctrl := TControl(Sender);
   BarraDeEstado.Panels[1].Text:= ctrl.Name;
   BarraDeEstado.Panels[2].Text:= Format('Top: %d',
                              [ctrl.Top]);
   BarraDeEstado.Panels[3].Text:= Format('Left: %d',
                              [ctrl.Left]);
end;
```

Compile y ejecute la aplicación. Observe cómo a medida que el ratón va pasando por los diferentes controles del formulario, la barra de estado le informa sobre los mismos: su clase, su nombre y las posiciones que ocupan en el formulario, como ve en la figura 11.6.

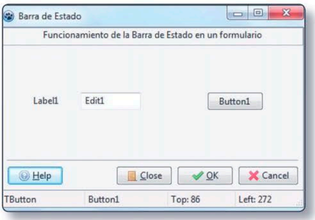



**Figura 11.6.** Barra de estado con información sobre Button1

Observe las diferencias entre `ClassName` y `Name`. La primera se refiere a la clase misma, el tipo de control; mientras que la segunda recoge el nombre que se le ha dado a la instancia, que puede ser el que por defecto asigna Lazarus: `Label1`, `Edit1`...

## 11.6 LOS COMPONENTES DE ENTRADA DE TEXTO

Aunque un formulario o componente puede controlar directamente la entrada del teclado por parte de un usuario utilizando un evento **OnKeyPress**, no se trata de una operación muy común. Lazarus ofrece controles preparados para obtener entradas de cadena.

Todas las aplicaciones gráficas necesitan interactuar con el usuario del programa, bien sea introduciendo datos directamente, bien seleccionándolos a través de algunos controles. Todos estos controles se llaman genéricamente **controles de edición**. Estos deben ser capaces de recibir el foco de una aplicación en cualquier momento. Lazarus maneja el foco de un control con las propiedades `TabStop` y `TabOrder` y los métodos `SetFocus`, `RemoveFocus`, `CanFocus`, `Focused`, `SelectNext` y `PerformTab`.

### 11.6.1 *TEdit* y *TLabeledEdit*

Los componentes **TEdit** y **TLabeledEdit** son quizás los componentes más habituales en los programas gráficos. `TEdit` se representa por el icono de la pestaña *Standard*, mientras que `TLabeledEdit` se encuentra en la pestaña *Additional*. Ambos permiten introducir una única línea de texto dinámico. Su propiedad más importante es `Text`, que recoge la cadena introducida por el usuario. `TLabeledEdit` es simplemente un control `TEdit` con una etiqueta adjunta. Esta etiqueta aparece como propiedad del control compuesto, que hereda de `TCustomEdit`. Es un componente más cómodo, pues permite reducir el número de controles del formulario, moverlos de manera más sencilla y tener una mejor organización.

La única condición que se puede imponer a ambos controles es el número máximo de caracteres aceptados, lo que se controla con la propiedad `MaxLength`. Si se desea que solo se admitan unos caracteres específicos, se puede controlar con el evento `OnKeyPress` del cuadro de edición. Por ejemplo, podemos escribir un método que compruebe si el carácter introducido por teclado es un número o la tecla **[Retroceso]**, que tiene un valor numérico de 8. Si no es así, se cambia el valor de la tecla al carácter 0, para que el control de edición no lo procese y se produzca un sonido de advertencia:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var
                key: Char);
begin
// verifica si la tecla es un numero o retroceso
  if not (Key in [' 0 ' . . ' 9', #8]) then
    begin
      Key := #0;
      Beep;
    end ;
end ;
```

Otras propiedades relativas a la funcionalidad de estos controles de edición son `ReadOnly`, que previene la edición; `EchoMode`, que permite seleccionar la opción `Password` para que la entrada por teclado se enmascare con el carácter elegido en la propiedad `PasswordChar`; y `CharCase`, que posee las opciones `LowerCase UpperCase` para que las entradas se introduzcan en minúsculas o mayúsculas, respectivamente.

Para ver cómo funcionan estos controles de edición, abra una nueva aplicación de Lazarus con el nombre **props_edicion** y establezca la anchura del formulario en 400 píxeles. Arrastre sobre el formulario un control `TEdit` y otro `TToggleBox` de la pestaña *Standard* y un `TLabeledEdit` de la pestaña *Additional*.

Ponga como leyenda del botón `Mandar selección a LabeledEdit1` y ajuste su tamaño para que se muestre todo el contenido. Haga doble clic sobre el botón y en el esqueleto del código del manejador `ToggleBox1Change` del editor, escriba lo siguiente

```
procedure TForm1.ToggleBox1Change(Sender: TObject);
begin
   case ToggleBox1.Checked of
   False : begin
           Edit1.SetFocus;  //establece el foco
           ToggleBox1.Caption := 'Mandar selección a
                LabeledEdit1';
           end;
   True: begin
           LabeledEdit1.SetFocus;
           ToggleBox1.Caption := 'Mandar selección a
                Edit1';
         end;
    end;
end;
```

Compile y ejecute el programa. El foco se encontrará sobre el cuadro de edición `TEdit`. El botón `ToggleBox` posee dos estados: `unchecked` y `checked`. Cuando se encuentra en su estado por defecto (`False`), el foco se establece en el cuadro de edición `Edit1` y muestra la leyenda `Mandar selección a LabeledEdit1` (Figura 11.7 A). Cuando se pulsa (`True`), se pasa el foco a `LabeledEdit1` y le cambiamos la leyenda a `Mandar selección a Edit1` (Figura 11.7 B).

**Figura 11.7.** Estados del formulario en tiempo de ejecución

Investigue ahora cómo afectan las propiedades descritas en la página anterior al funcionamiento de estos controles.

## 11.6.2 *TMaskEdit*

El componente **`TMaskEdit`** es un control de la pestaña *Additional* representado con el icono en la Paleta de componentes. Con él se puede personalizar un cuadro de edición mediante una máscara de entrada. La máscara es una cadena de símbolos que se define a través de la propiedad `EditMask` (Figura 11.8).

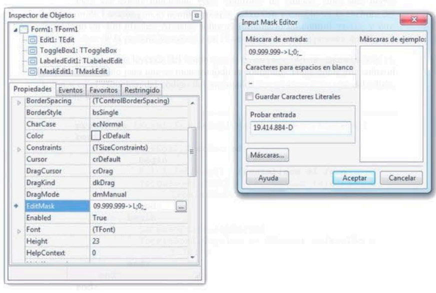



**Figura 11.8.** Acceso al editor de máscaras del control TMaskEdit

El **editor de máscaras** permite introducir la máscara adecuada, pero también solicita un carácter que reserve sitio para la entrada y decidir si se guarda la máscara junto con la cadena final. Por ejemplo, se puede elegir recoger un documento de identidad con este control como 19.414.884-D y guardarlo como 19414884D.

**TRUCO**
Si hace clic sobre el botón **Máscaras** puede cargar ficheros con máscaras predefinidas.

`TMaskEdit` es en realidad una cadena que posee tres campos separados por puntos y comas. El primero es la máscara en sí misma. El segundo es el carácter que determina si se guardan o no los caracteres literales de la máscara junto con el valor introducido por el usuario. El tercer campo es el símbolo que se emplea para representar caracteres no introducidos en la máscara.

Los caracteres que pueden emplearse para diseñar las máscaras son los recogidos en la tabla siguiente:

| Carácter | Significado |
|---|---|
| `!` | Representa un carácter opcional, que se desplegará como un carácter en blanco |
| `>` | Los caracteres que siguen irán en mayúsculas |
| `<` | Los caracteres que siguen irán en minúsculas |
| `\` | Se emplea para introducir como un literal un carácter especial de máscara |
| `L` | Requiere una letra en esta posición |
| `l` | Se permite una letra, pero no es obligatorio |
| `A` | Requiere una letra o número en esa posición |
| `a` | Se permite una letra o número, pero no es obligatorio |
| `0` | Requiere un número en esa posición |
| `9` | Permite la presencia de un número, pero no es obligatorio |
| `#` | Permite un número o los signos más o menos en esa posición |
| `:` | Separa horas : minutos : segundos |
| `/` | Separa días/meses/año |

**Tabla 11.1.** Caracteres especiales para definir máscaras en **`TMaskEdit`**

Cualquier carácter que no aparezca en la tabla se considera como un literal en la máscara, es decir, se insertan automáticamente y el cursor los saltará cuando el usuario introduzca los datos en el cuadro de edición.

Por ejemplo, la máscara `99.999.990->L;0;_` es válida para un documento de identidad español. El primer campo recoge los 9 números de identificación más un guión y la letra de control, en mayúsculas. El segundo campo, `0`, indica a Lazarus que guarde los nueve caracteres tecleados por el usuario, en lugar de los trece que tiene la máscara.

### 11.6.3 Edición de números enteros y reales

En la pestaña *Misc* de la Paleta de componentes existen dos controles específicos para introducir y validar números enteros y reales en coma flotante: `TSpinEdit` y `TFloatSpinEdit`, respectivamente. Los números se pueden teclear directamente o mediante las flechas que aparecen en los controles.

La propiedad **`Value`** de ambos controles no es una cadena, sino un número, un entero para `TSpinEdit` y un real de tipo double para `TFloatSpinEdit`. Esto es una ventaja, pues no es necesario convertir manualmente los tipos y, además, no hace falta validar la entrada del usuario, pues solo pueden introducirse enteros o reales en coma flotante.

Veamos con un ejemplo cómo utilizar estos controles programando una sencilla calculadora de números enteros. Abra un proyecto de Lazarus de nombre **`calculadora`** y asigne el nombre **`calc_unit.pas`** a su formulario. Añada la unidad `math` a la cláusula **`uses`** y arrastre al formulario dos etiquetas de nombres `lblA` y `lblB` con leyendas `EnteroA` y `EnteroB`, respectivamente.

Bajo las etiquetas, añada dos controles `TSpinEdit` con los nombres `enteroA` y `enteroB` y ensánchelos un poco para que acepten números grandes. En las propiedades `MinValue` ponga `-1000`, y en `MaxValue`, `1000`. Añada ahora bajo ellos un componente `TDividerBevel` con la leyenda `Resultados calculados` y un componente `TListBox` de nombre `lbResultados` y dimensiones `256 × 104`. El formulario debe parecerse al mostrado en la figura 11.9.

**Figura 11.9.** Formulario en tiempo de diseño

Una forma muy efectiva de describir a otros programadores la disposición de los controles, sus nombres y demás propiedades es con el contenido del fichero `lfm`, que, en este caso, es así:

```
object Form1: TForm1
  Caption = 'Calculadora'
  object enteroA: TSpinEdit
    MaxValue = 1000
    MinValue = -1000
  end
  object enteroB: TSpinEdit
    MaxValue = 1000
    MinValue = -1000
  end
  object lblA: TLabel
    Caption = 'Entero A'
  end
  object lblB: TLabel
    Caption = 'Entero B'
  end
  object DividerBevel1: TDividerBevel
     Caption = 'Resultados calculados'
  end
  object lbResultados: TListBox
    Height = 104
    Width = 256
  end
end
```

Observe cómo se recogen para el formulario todos los controles que se han arrastrado sobre él, sus nombres, el tipo de control y las propiedades modificadas.

Seleccione los dos controles de edición y en la pestaña `Eventos` del Inspector de objetos haga doble clic sobre `OnChange` para crear un nuevo manejador. Sustituya el nombre que le ha dado Lazarus por `CambioAoB` y complete en el editor el esqueleto del procedimiento como sigue:

```
procedure TForm1.CambioAoB(Sender: TObject);
begin
    RealizarCalculos;
end;
```

Cada vez que cambie el contenido de los controles llamamos a un procedimiento de nombre `RealizarCalculos`, que aún no hemos escrito. En la sección privada de la clase `TForm1`, declare el procedimiento `RealizarCalculos` y emplee la compleción automática de código para generar el esqueleto del mismo. Complételo como sigue:

```
procedure TForm1.RealizarCalculos;
var  OK: boolean;
      n: Int64;
begin
  lbResultados.Items.Clear;
  lbResultados.Items.Add('A + B : ' +
IntToStr(CalcSum));
  lbResultados.Items.Add('A - B : ' +
IntToStr(CalcDif));
  n := CalcIntDivision(OK);
  if OK then
    begin
      lbResultados.Items.Add('A DIV B : ' + IntToStr(n));
      mlbResultados.Items.Add('A MOD B :
         ' + IntToStr(CalcModulo(OK)));
    end
  else lbResultados.Items.Add('A DIV B y A MOD B no son
         calculables');
    lbResultados.Items.Add('A x B : ' + IntToStr(CalcProducto));
    lbResultados.Items.Add('A^B : ' + FloatToStr(CalcPotencia));
end;
```

En este procedimiento estamos llamando a distintos procedimientos, que escribiremos a continuación. En la sección privada de la clase añada las declaraciones para esos procedimientos y use la compleción de código para generar el esqueleto de los mismos:

```
private
 procedure RealizarCalculos;
 function CalcSum: int64;
 function CalcDif: int64;
 function CalcProducto: int64;
 function CalcPotencia: double;
 function CalcIntDivision(var isValid: boolean): int64;
 function CalcModulo(var isValid: boolean): int64;
end;
...
implementation
...
function TForm1.CalcSum: int64;
begin
     Result := enteroA.Value + enteroB.Value;
end;

function TForm1.CalcDif: int64;
begin
     Result := enteroA.Value - enteroB.Value;
end;

function TForm1.CalcProducto: int64;
begin
     Result := enteroA.Value * enteroB.Value;
end;

function TForm1.CalcPotencia: double;
begin
     Result := intpower(enteroA.Value,enteroB.Value);
end;

function TForm1.CalcIntDivision(var isValid: boolean):
                   int64;
begin
     Result := 0;
     isValid := (enteroB.Value <> 0);
     if isValid
```

```
          then Result := enteroA.Value div enteroB.Value;
end;
function TForm1.CalcModulo(var isValid: boolean):
int64;
begin
  Result := 0;
 isValid := (enteroB.Value <> 0);
 if isValid
   then Result := enteroA.Value mod enteroB.Value;
end;
```

Cada vez que cambie el valor de un control `TSpinEdit` se llama al procedimiento `RealizarCalculos`, que, simplemente, escribe un conjunto de líneas en el componente `TListBox`. Hemos de asegurarnos de que esta actualización ocurre cuando se ejecuta por primera vez el programa, lo que conseguimos con el evento `OnActivate`. Seleccione el formulario y haga doble clic sobre este evento en el Inspector de objetos y complete el código así:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  CambioAoB(nil);
end;
```

Ya está listo para compilar y ejecutar el programa. Compruebe el funcionamiento con varios valores para los enteros A y B sin superar los límites que impusimos (Figura 11.10).

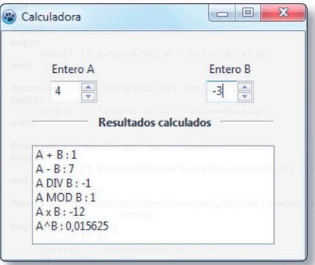



**Figura 11.10.** Aplicación en tiempo de ejecución

## 11.7 SELECCIÓN DE OPCIONES

Existen dos controles fundamentales en toda aplicación gráfica con los que un usuario puede escoger distintas opciones: **casillas de verificación** y **botones de radio**; y otros dos controles para agrupar conjuntos de opciones: **listas** y **rangos**.

### 11.7.1 *TCheckBox* y *TRadioButton*

El componente **`TCheckBox`**, representado por el icono ☑ de la pestaña *Standard*, hace referencia a la casilla de verificación, que corresponde a una opción seleccionable sea cual sea el estado de otras casillas. Posee dos estados fundamentales: `Checked` y `Unchecked`, que pueden elegirse desde la propiedad `State`. El botón de radio, `TRadioButton`, representado por el icono ⊙ del mismo panel, indica una selección exclusiva. Dos botones de radio dentro del mismo contenedor de grupo de radio no pueden permanecer seleccionados a la vez. Por defecto, uno de ellos debe estar siempre seleccionado, lo que se consigue activando la propiedad `Checked`.

Para alojar varios grupos de botones o casillas de verificación, se puede usar un control `TGroupBox` para mantenerlos juntos tanto visual como funcionalmente. Para construir un grupo de botones de radio, solo hay que colocar en el formulario el componente `TRadioGroup` y, a continuación, arrastrar los botones de radio al marco del grupo, como en el ejemplo siguiente:

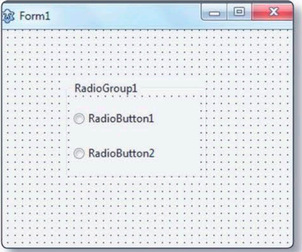



**Figura 11.11.** Contenedor RadioGroup con dos botones de radio

También es posible añadir los ítems al componente a través de la propiedad `Items` del componente `TRadioGroup`. Los botones de radio se gestionan automáticamente desde el control contenedor. Utilizar un grupo de radio es sencillo,

puesto que los elementos forman parte de una lista. Otra ventaja es que un componente `TRadioGroup` puede alinear los botones de radio de forma automática en una o más columnas según se indique en la propiedad `Columns`.

Los controles `TRadioGroup` y `TCheckGroup` son contenedores de controles en el más amplio sentido, pues una vez que un control está en su interior, este pertenece al contenedor y no podrá arrastrarse fuera de él.

Para mostrar cómo se trabaja con estos controles de selección, vamos a programar una sencilla aplicación para formatear una cadena de texto según la elección del usuario. Abra una aplicación en Lazarus de nombre **opciones** con una unidad llamada **main.pas**.

El ejemplo tiene tres `RadioGroup` y un `CheckGroup`. La estructura del formulario y la de sus componentes y acciones es la siguiente.

```
object Form1: TForm1
  Caption = 'Opciones'
  ClientHeight = 240
  ClientWidth = 459
  object edTexto: TLabeledEdit
    Width = 214
    EditLabel.Caption = '&Texto'
    MaxLength = 25
    TabOrder = 0
    OnChange = edTextoChange
  end
  object cgEstilo: TCheckGroup
    Height = 105
    Width = 113
    Caption = 'Estilo'
    TabOrder = 1
    object cbNegrita: TCheckBox
     end
    object cbCursiva: TCheckBox
    Caption = 'Cursiva'
    end
    object cbSubrayado: TCheckBox
    Caption = 'Subrayado'
    end
  end
  object rgFuente: TRadioGroup
    Height = 105
    Width = 88
```

```
      Caption = 'Fuente'
      Items.Strings = (
        'Arial'
        'Courier'
        'Times '
      )
      TabOrder = 2
    end
    object rgTamano: TRadioGroup
      Height = 105
      Width = 96
      Caption = 'Tamaño'
      Items.Strings = (
        '12 puntos'
        '16 puntos'
      )
      TabOrder = 3
    end
    object rgColores: TRadioGroup
      Height = 105
      Width = 81
      Caption = 'Color'
      Items.Strings = (
        'Negro'
        'Rojo'
        'Azul'
      )
      TabOrder = 4
    end
    object btnAplicar: TButton
      Caption = 'Aplicar'
      OnClick = btnAplicarClick
      TabOrder = 5
    end
    object lbTexto: TLabel
    end
  end
```

Observe que el control `cgEstilo` contiene tres casillas de verificación: `cbNegrita`, `cbCursiva` y `cbSubrayado`, y que en los grupos de botones de radio `rgFuente`, `rgTamano` y `rgColores` hemos empleado la propiedad Items para definir el número y leyenda de cada uno de ellos. Así mismo, se han definido dos eventos: `OnClick` en el botón `Aplicar` y `OnChange` en el cuadro de edición. Veamos el código asociado a cada uno de ellos:

```
procedure TForm1.btnAplicarClick(Sender: TObject);
var
  Estilos: TFontStyles;
begin
  edTexto.SetFocus;
  Estilos := [];
  if cbNegrita.Checked then include(Estilos, fsBold);
  if cbCursiva.Checked then include(Estilos, fsItalic);
  if cbSubrayado.Checked then include(Estilos, fsUnder-
line);

  lbTexto.Font.Style := Estilos;

  case rgFuente. ItemIndex of
    0: lbTexto.Font.Name := 'Arial';
    1: lbTexto.Font.Name := 'Courier';
    2: lbTexto.Font.Name := 'Times';
  end;

  case rgTamano.ItemIndex of
    0: lbTexto.Font.Size := 12;
    1: lbTexto.Font.Size := 16;
  end;

  case rgColores.ItemIndex of
    0: lbTexto.Font.Color := clBlack;
    1: lbTexto.Font.Color := clRed;
    2: lbTexto.Font.Color := clBlue;
  end;
  lbTexto.Caption := edTexto.Text;
end;

procedure TForm1.edTextoChange(Sender: TObject);
begin
       lbTexto.Caption := edTexto.Text;
end;
```

Cuando ejecute el programa, el foco estará sobre el cuadro de edición esperando a que introduzca una cadena. A medida que lo hace, se irá copiando el texto como leyenda de la etiqueta `lbTexto`. Una vez que haya seleccionado una o varias opciones de **`Estilo`**, **`Fuente`**, **`Tamaño`** y **`Color`**, pulse sobre el botón **`Aplicar`**. En ese momento, el texto de la etiqueta cambiará según el formato elegido.

Es interesante que se fije en que en las casillas de verificación comprobamos la selección a través de los literales de sus nombres, mientras que en los grupos de botones de radio, al definir sus contenidos con la propiedad `Items`, lo hacemos por su índice.

Observe en la figura 11.12 la aplicación en tiempo de ejecución:

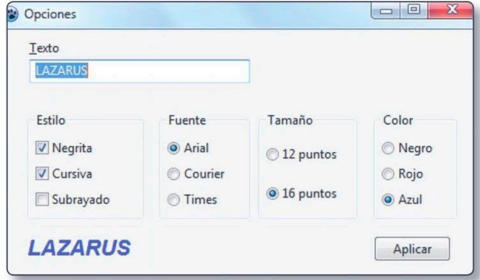



**Figura 11.12.** Aplicación Opciones en tiempo de ejecución

## 11.7.2 Listas

Los botones de radio no son muy adecuados cuando el número de elementos seleccionables es elevado. En estos casos empleamos cuadros de lista o relacionados para permitir la selección de uno o varios de ellos.

### 11.7.2.1 EL COMPONENTE *TLISTBOX*

Este elemento de la pestaña *Standard* de la Paleta de componentes es el más simple para mostrar listas de elementos seleccionables. La lista de cadenas se almacena en la propiedad `Items` del control y se accede a cada ítem con la propiedad `ItemIndex`.

Otra propiedad importante es que con el componente `TListBox` se puede escoger si seleccionar solo un elemento, como en un grupo de botones de radio, o permitir selecciones múltiples, como en un grupo de casillas de verificación. Esto se consigue especificando el valor de la propiedad `MultiSelect` como `True`.

Para hacer una selección múltiple el usuario pulsa las teclas **[May]** o **[Ctrl]** para seleccionar diversos elementos consecutivos o no, respectivamente. Esta segunda opción la determina el estado `True` de la propiedad `ExtendedSelect`. El control se puede configurar para utilizar un número fijo de columnas con la propiedad `Columns`.

Para probar el funcionamiento de las listas, hemos creado un ejemplo, de nombre **`ListBox`**, que permite añadir ítems a una lista desde un cuadro de edición y eliminarlos de ella según la selección del usuario (Figura 11.13).

Los usuarios escriben elementos en el cuadro de edición y al pulsar sobre el botón **Añadir**, el elemento pasará a la lista. El programa comprueba que no se está intentando añadir una cadena vacía. En este caso, un mensaje de error advertirá de ello y el foco continuará en el cuadro de edición esperando una cadena. Para eliminar elementos, el usuario los selecciona de forma consecutiva o no consecutiva antes de pulsar el botón **Eliminar**. El botón **Cerrar** permite salir de la aplicación.

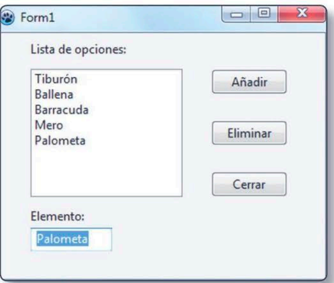



**Figura 11.13.** El ejemplo ListBox en tiempo de ejecución

Los controles que emplea el programa son los siguientes:

```
btnAnadir: TButton
btnEliminar: TButton
btnCerrar: TButton
lbOpciones: TListBox
edElemento: TLabeledEdit
```

Y el código asociado a los mismos:

```
procedure TForm1.btnAnadirClick(Sender: TObject);
begin
  if edElemento.Text = '' then
  begin
    edElemento.SetFocus;
    MessageDlg ('Es necesario un elemento', mtError,
            [mbOK], 0);
  end
  else
  begin
    lbOpciones.Items.Add(edElemento.Text);
    edElemento.SelectAll;
    edElemento.SetFocus;
  end;
end;

procedure TForm1.btnCerrarClick(Sender: TObject);
begin
  Close;
end;

procedure TForm1.btnEliminarClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := 0;
  with lbOpciones do
    while Index < Items.Count do
    begin
      if Selected[Index] then
      begin
        Items.Delete(Index);
        Index := 0;
      end
      else
        Inc(Index);
    end;
edElemento.SetFocus;
end;
```

### 11.7.2.2 CUADROS COMBINADOS

El problema fundamental de las listas es que ocupan mucho espacio en pantalla. El control `TComboBox`, por otro lado, combina un cuadro de edición y una lista desplegable en el mínimo espacio. El comportamiento de este control depende mucho del valor de su propiedad `Style` (Figura 11.14):

- **`DropDown`**. Define un cuadro combinado que permite editar directamente y mostrar una lista mediante solicitud.
- **`DropDownList`**. Define un cuadro combinado que no permite su edición.
- **`Simple`**. Define un cuadro combinado bajo el cual siempre se muestra la lista.

Acceder al texto del valor seleccionado por el usuario en un cuadro combinado es más sencillo que hacer la misma operación en un cuadro de lista, ya que se puede utilizar la propiedad `Text`.

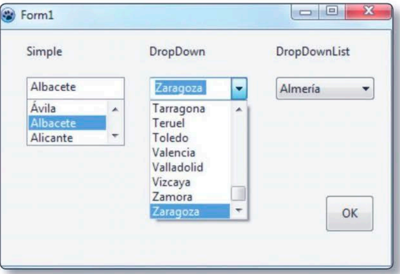



**Figura 11.14.** Varios tipos de cuadros combinados

Lazarus incluye el evento `OnCloseUp`, que se corresponde con el cierre de la lista desplegable y complementa al evento `OnDropDown`. Por su parte, el evento `OnSelect` solo se lanza cuando el usuario realiza una selección en la lista desplegable, en vez de escribir en la parte de edición.

Otra propiedad importante es `AutoComplete`. Cuando se establece, el control `TComboBox` busca automáticamente la cadena más parecida a la que el usuario está escribiendo

## 11.7.3 Rangos

Los rangos se emplean para entradas numéricas y como selección de un elemento de una lista.

### 11.7.3.1 EL CONTROL *TTRACKBAR*

En muchas ocasiones, los valores que solicita un programa al usuario son numéricos y se encuentran entre unos ciertos límites conocidos. Aunque podemos utilizar un control `TEdit` para pedir datos de este tipo, existen bastantes posibilidades más, entre las que se encuentra el control **`TTrackBar`**. Este control se representa por el icono de la pestaña *Common Controls* de la Paleta de componentes. Mediante este componente, además, es posible seleccionar rangos de valores en lugar de un solo valor. En apariencia, un control `TTrackBar` es bastante similar a una barra de desplazamiento y de hecho su funcionamiento es muy parecido. Este control cuenta con unos límites y una posición actual, mostrada visualmente mediante un cursor. En el interior del control es posible disponer unas marcas a intervalos regulares, entre el mínimo y el máximo. Estas características y algunas más las podremos establecer mediante sus propiedades, alguna de las cuales recogemos en la tabla 11.2.

| Propiedad | Contenido |
|---|---|
| **`Frecuency`** | Frecuencia de las marcas de medida |
| **`LineSize`** | Incremento o decremento pequeño |
| **`Max`** | Valor máximo |
| **`Min`** | Valor mínimo |
| **`PageSize`** | Incremento o decremento grande |
| **`Position`** | Posición actual del cursor en el punto de control |
| **`SelEnd`** | Punto de inicio de la selección actual |
| **`SelStart`** | Punto de fin de la selección actual |

**Tabla 11.2.** Propiedades básicas del control TTrackBar

Mediante las propiedades `Min` y `Max` fijaremos los valores mínimo y máximo de la medida; también podemos asignar una posición inicial modificando el valor de la propiedad `Position`, que por defecto tomará el mismo valor que asignemos a `Min`. El valor de la propiedad `Position` será lo único que cambie por una actuación del usuario en tiempo de ejecución, cuando el cursor del control se desplace a otro punto, ya sea mediante el ratón o con el teclado. En cualquiera de estos casos el

control `TTrackBar` generará un evento `OnChange`, que podemos aprovechar para actualizar cualquier parámetro dependiendo de la posición actual en este control.

Un control `TTrackBar` puede aparecer en el formulario en sentido horizontal, que es el estado por defecto, o vertical, según el valor que asignemos a la propiedad `Orientation`.

A la izquierda y derecha o arriba y abajo, dependiendo de la orientación, el control puede mostrar unas marcas intermedias, que permitirán tener una idea aproximada del valor actual que se está usando. Mediante la propiedad `TickMarks` podemos determinar la situación de esas marcas en el control.

En la figura siguiente puede ver en tiempo de ejecución un formulario con dos controles `TTrackBar`.

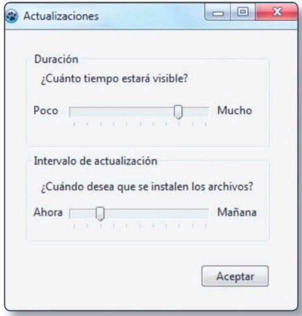



**Figura 11.15.** Programa en tiempo de ejecución con dos TTrackBar

Además de para mostrar o fijar una posición del cursor, lo que en definitiva se traduce en un cierto valor para la variable de tipo entero `NombreControl.Position`, el control `TTrackBar` también nos permite seleccionar un rango o intervalo de valores, mediante las propiedades `SelStart` y `SelEnd`. Inicialmente, estas dos propiedades tienen el valor cero, lo que indica que no hay seleccionado un intervalo. Al asignar a `SelStart` y `SelEnd` un valor, podremos ver que en el control aparece de un color diferente el rango seleccionado y que, además, aparecen dos marcas especiales, dos pequeños triángulos, que delimitan perfectamente el intervalo.

### 11.7.3.2 EL CONTROL *TPROGRESSBAR*

El control **TProgressBar** de Lazarus se representa con el icono en la pestaña *Common Controls* de la Paleta de componentes. Su finalidad es mostrar de forma gráfica el estado actual de un proceso en curso. Este control cuenta con propiedades similares a las de un control TTrackBar.

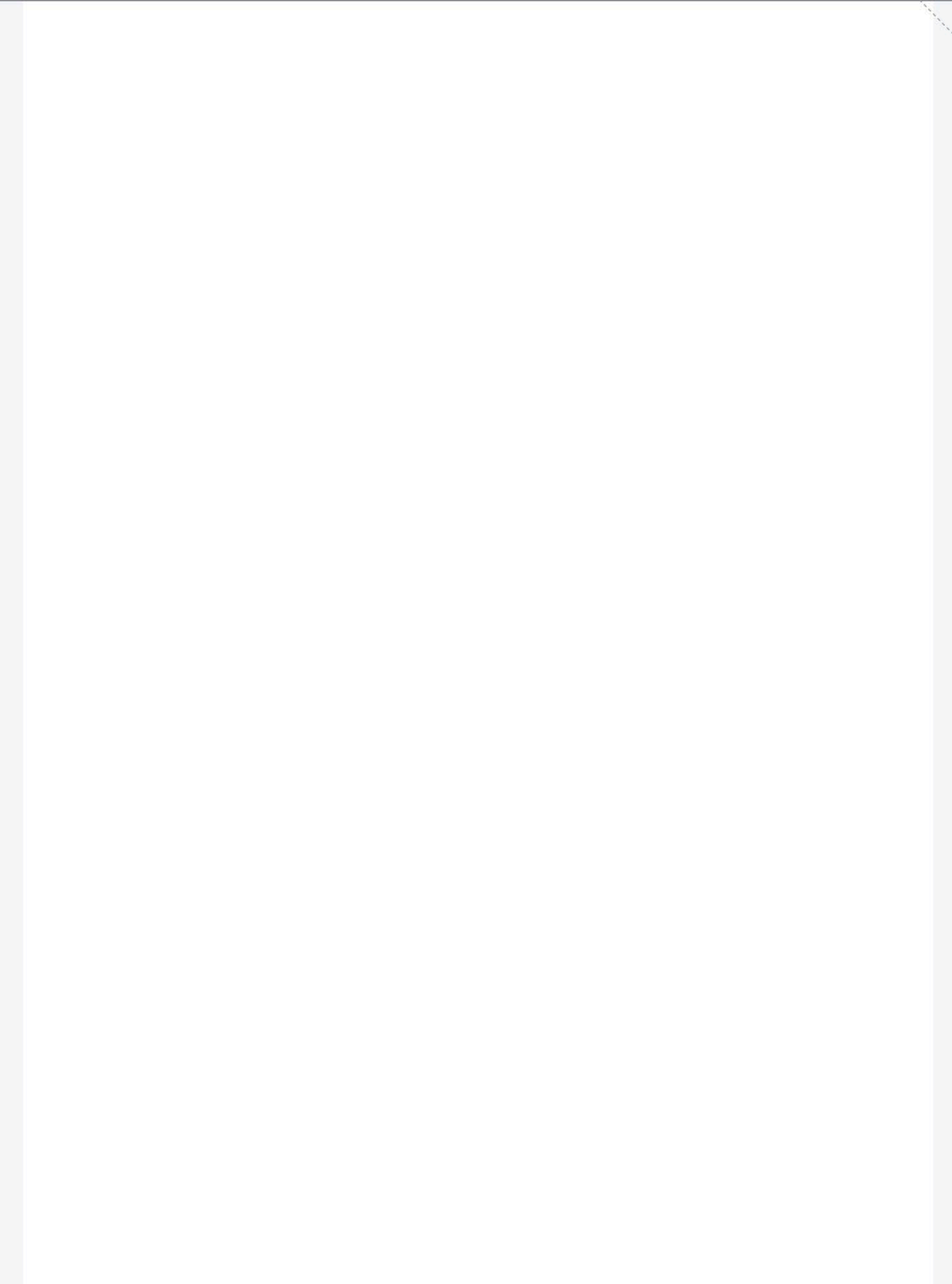
Antes de iniciar el proceso cuyo progreso se va a reflejar mediante este control, hay que asignar a las propiedades Min y Max del control un valor mínimo y un valor máximo que se correspondan con el estado de inicio y fin del proceso.

A medida que el curso del proceso va avanzando, se debe actualizar el valor de su propiedad Position, lo que tendrá un reflejo inmediato en el aspecto del control, que se irá llenando de bloques de color que indicarán la parte que ya se ha completado. En lugar de manipular directamente un valor de la propiedad Position, se puede asignar previamente un valor de incremento a la propiedad Step, llamando posteriormente, en cada paso, al método StepIt, que se encargará de realizar la actualización.

Para ver cómo funciona, arrastre sobre un formulario un control TProgressBar y asigne su propiedad Max = 1000 y Step = 1. Además, coloque un botón y haga doble clic en él para programar su evento OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  progressbar1.Position := 0;
  while progressbar1.Position < progressbar1.Max do
    progressbar1.StepIt;
end;
```

Compile y ejecute el programa. Pulse sobre el botón y verá que la barra de progreso avanza hasta recorrer todo su dominio.



# 12

# LA INTERFAZ DE USUARIO

En el capítulo anterior hemos estudiado los conceptos básicos de la clase `TControl` y sus clases derivadas en la biblioteca LCL. Así mismo, hemos repasado los principales controles que como programador novel debe conocer para construir una interfaz. En este capítulo vamos a centrarnos en otros dos controles más avanzados, **`PageControl`** y **`TabControl`**, para definir el diseño global de la interfaz gráfica del usuario. Después de estos componentes, vamos a comentar las barras de herramientas, con algunas características bastante avanzadas. Con esto conseguiremos la base para el resto del capítulo, en el que se habla de acciones.

Las aplicaciones gráficas en cualquier sistema operativo suelen tener varios modos de ofrecer órdenes, como menús y barras de herramientas. Para separar las órdenes que puede dar un usuario de sus varias representaciones en la interfaz, Lazarus emplea el concepto de **acciones**. La construcción de la interfaz sobre estas acciones también es completamente visual.

## 12.1 FORMULARIOS DE VARIAS PÁGINAS

Cuando se debe mostrar mucha información y un gran número de controles en un formulario, es necesario emplear varias páginas o pestañas, de modo que el usuario pueda seleccionarlas en tiempo de ejecución. Para estas acciones, pueden emplearse dos controles de la pestaña *Common Controls*:

- **`TPageControl`**. Tiene pestañas en uno de los laterales y varias fichas u hojas que cubren la totalidad de la superficie. Como existe una ficha por solapa, podrá arrastrar componentes en cada una para obtener el efecto deseado tanto en tiempo de diseño como de ejecución.

- **`TTabControl`**. Solo posee la parte de la pestaña, pero no ofrece fichas en las que almacenar la información. En este caso, como programador, es conveniente que use uno o más componentes para imitar la operación de cambio de ficha, o que coloque varios formularios distintos dentro de las pestañas para simular páginas.

Una tercera clase es `TTabSheet`, que representa una única ficha de `PageControl`. No es independiente de este control, sino que se crea en tiempo de diseño desde el menú contextual de `PageControl`, en la opción **Añadir página** (Figura 12.1).

### 12.1.1 *TPageControl* y *TTabSheet*

Para no describir todas y cada una de las propiedades y métodos que soporta el componente `TPageControl`, hemos creado un ejemplo que muestra cómo añadir pestañas a un formulario con funcionalidades distintas. El ejemplo se llama **`pagecontrol`** y presenta tres páginas o fichas. La estructura básica del formulario es la siguiente:

```
object Form1: TForm1
  Height = 276
  Width = 321
  BorderIcons = [biSystemMenu, biMinimize]
  Caption = 'PageControl'
  object PageControl1: TPageControl
    ActivePage = tsTexto
    Images = ImageList1
    object tsHojas: TTabSheet
      Caption = 'Hojas'
      object Label3: TLabel
      object ListBox1: TListBox
    end
    object tsImagenes: TTabSheet
      Caption = 'Imágenes'
      ImageIndex = 1
      object lblAnchura: TLabel
       // resto de controles
```

```
      end
      object tsTexto: TTabSheet
        Caption = 'Tabs Text'
        ImageIndex = 2
        object Memo1: TMemo
          Anchors = [akTop, akLeft, akRight, akBottom]
          OnChange = Memo1Change
        end
        object BitBtnCambiar: TBitBtn
          Anchors = [akTop, akRight]
          Caption = '&Cambiar'
             OnClick = BitBtnCambiarClick
        end
        object BitBtnAnadir: TBitBtn
          Caption = 'Añadir Pestaña'
          OnClick = BitBtnAnadirClick
          TabOrder = 2
        end
      end
    end
    object BitBtnAnterior: TBitBtn
      Anchors = [akRight, akBottom]
      Caption = 'An&terior'
      OnClick = BitBtnAnteriorClick
    end
    object BitBtnSiguiente: TBitBtn
      Anchors = [akLeft, akBottom]
      Caption = '&Siguiente'
      OnClick = BitBtnSiguienteClick
    end
    object ImageList1: TImageList
      Bitmap = {...}
    end
  end
```

Fíjese en que las pestañas están conectadas a un mapa de bits mediante un control `TImageList` y en que algunos controles usan la propiedad `Anchors` para mantener una distancia fija con los bordes del formulario.

Cada objeto `TTabSheet` tiene su propia leyenda, que aparecerá en la solapa de la hoja. En tiempo de diseño puede usar el menú contextual para crear fichas nuevas y moverse por ellas. Observe en la figura 12.1 el menú contextual del componente `TPageControl`, junto con la segunda ficha. Esta hoja contiene un cuadro de imagen y comparte dos botones con las otras pestañas.

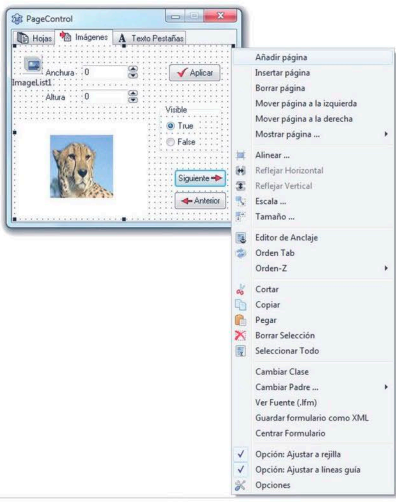



**Figura 12.1.** La segunda hoja del ejemplo Pagecontrol con su menú contextual

Si coloca un control en una hoja, este estará disponible solo en esa ficha. Para tener el mismo componente en cada ficha, en nuestro ejemplo dos botones de mapas de bits, simplemente debemos colocarlos en el formulario, pero fuera del control `TPageControl`, y, a continuación, traerlos hacia delante de las fichas mediante la orden **Mover al frente** del menú contextual. Los dos botones que hemos colocado en cada hoja (**Siguiente** y **Anterior**) se usan para mover las fichas adelante y atrás. Veamos el código relativo a cada uno de ellos:

```
procedure TForm1.BitBtnSiguienteClick(Sender: TObject);
begin
   PageControl1.SelectNextPage (True);
end;

procedure TForm1.BitBtnAnteriorClick(Sender: TObject);
begin
  PageControl1.SelectNextPage (False);
end;
```

Fíjese en que no es necesario verificar si estamos en la primera o última ficha. Ahora nos podemos centrar en la primera hoja. Tiene un cuadro de lista que en tiempo de ejecución contiene los nombres de las pestañas. Si se hace clic sobre cualquier elemento de la lista, el flujo de programa se pasa a la hoja elegida, del mismo modo que puede hacerse al picar sobre las solapas o a través de los botones **Siguiente** y **Anterior**. El cuadro de lista se rellena mediante el método **`FormCreate`**, asociado al evento **`OnCreate`** del formulario, y copia el título de cada hoja. La propiedad **`Pages`** contiene una lista de objetos `TTabSheet`, de modo que añade sus nombres como ítems de la lista así:

```
for I := 0 to PageControl1. PageCount -1 do
  ListBox1.Items.Add (PageControl1.Pages [I].Caption);
```

Cuando se hace clic sobre un elemento de la lista, se selecciona la ficha correspondiente:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.Pages [ListBox1.ItemIndex];
end;
```

La segunda página contiene dos cuadros de edición conectados a dos componentes **`UpDown`**, dos botones de radio y un control **`TImage`** con la imagen de un guepardo. El usuario puede escribir un número o escogerlo pulsando sobre los botones con el ratón o con las flechas del teclado, marcar si desea que la imagen sea o no transparente y hacer clic sobre el botón **Aplicar** para realizar los cambios:

```
procedure TForm1.BitBtnAplicarClick(Sender: TObject);
begin
  // Establece medidas de la imagen
  Image1.Width := StrToInt (EditWidth.Text);
  Image1.Height := StrToInt (EditHeight.Text);
  // Establece la visibilidad de la misma
  if RadioButton1.Checked then
    Image1.Visible := True
  else
    Image1.Visible := False;
end;
```

La última hoja posee un componente `TMemo` con los nombres de las fichas añadidas en el método `FormCreate`, como puede ver en la figura 12.2. Es posible editar los nombres de las fichas y hacer clic sobre el botón **Cambiar** para modificar el texto de las pestañas:

```
procedure TForm1.BitBtnCambiarClick(Sender: TObject);
var
  I: Integer;
begin
  if Memo1.Lines.Count = PageControl1.PageCount then
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption := Memo1.Lines
[I];
  BitBtnCambiar.Enabled := False;
end;
```

Por último, el botón **Añadir pestaña** permite añadir una hoja nueva al control de ficha, aunque el programa no añada ningún componente (Figura 12.2). El objeto nuevo se crea usando el control ficha como su propietario, pero no puede funcionar sin configurar antes la propiedad `PageControl`. No obstante, antes de eso es necesario que la pestaña sea visible:

```
procedure TForm1.BitBtnAnadirClick(Sender: TObject);
var
  strLeyenda: string;
  NuevaPestana: TTabSheet;
begin
  strLeyenda := 'Nueva hoja';
  if InputQuery ('Nueva hoja', 'Leyenda', strLeyenda)
    then begin
    // Añade una nueva hoja vacía al control
    NuevaPestana := TTabSheet.Create (PageControl1);
    NuevaPestana.Visible := True;
    NuevaPestana.Caption := strLeyenda;
    NuevaPestana.PageControl := PageControl1;
    PageControl1.ActivePage := NuevaPestana;
    // Añade la hoja a ambas listas
    Memo1.Lines.Add (strLeyenda);
    ListBox1.Items.Add (strLeyenda);
  end;
end;
```

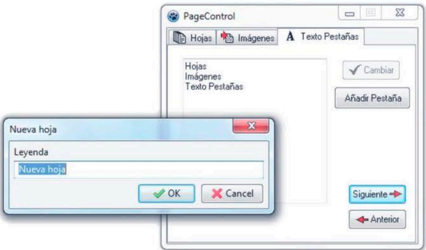



**Figura 12.2.** La tercera página del ejemplo se usa para añadir hojas en tiempo de ejecución o para modificar sus nombres

**TRUCO**
Siempre que se escribe un formulario basado en un control `TPageControl`, la primera ficha que aparece en tiempo de ejecución es aquella en la que nos encontrábamos en el momento de compilar el código. Para solucionarlo, basta con añadir el código `PageControl1.ActivePage := NombreFicha;` al método `FormCreate`.

## 12.1.2 La interfaz de un asistente

A veces resulta muy interesante utilizar un control `TPageControl` sin solapas. Es lo que vamos a usar ahora para programar un sencillo asistente con el que dirigir al usuario mediante una serie de pasos. Así, en lugar de pestañas que se puedan seleccionar en un orden cualquiera, los asistentes ofrecen los botones **Siguiente** y **Atrás** para desplazarse. El ejemplo se llama **`asistente`**.

El punto inicial es crear una serie de hojas en un `TPageControl` y establecer la propiedad `TabVisible` de cada hoja (`TTabSheet`) como `False`, mientras se mantiene la propiedad `Visible` del control `TPageControl` como `True`.

En la primera ficha (`Intro`) hemos colocado en el lado izquierdo una imagen y un control de biselado y al otro lado una etiqueta, una casilla de verificación y dos botones (Figura 12.3). En realidad, el botón **Siguiente** está dentro de la ficha, mientras que el botón **Atrás** está sobre ella y lo comparten todas las hojas. Las fichas siguientes tienen una apariencia similar, con etiquetas, casillas de verificación y botones en el lado derecho.

**Figura 12.3.** Primera ficha del ejemplo Asistente en tiempo de ejecución

Cuando hacemos clic sobre el botón **Siguiente** de la primera hoja, el programa recupera el estado de la casilla de verificación y decide qué ficha es la siguiente:

```
procedure TForm1.btnSiguiente1Click(Sender: TObject);
begin
  if cbAutor.Checked then
    MoverA (tsAutor)
  else
    MoverA (tsLazarus);
end;
```

El método `MoverA` añade la última ficha a una lista de hojas visitadas que se comporta como una pila. El objeto **`BackPages`** de la clase `TList` se crea al ejecutar el programa y la última ficha siempre se añade al final.

```
procedure TForm1.MoverA(TabSheet: TTabSheet);
begin
  // añade la última página a la lista
  BackPages.Add (PageControl1.ActivePage);
  btnAtras.Enabled := True;
  // cambia la hoja
  PageControl1.ActivePage := TabSheet;
  // mueve imagen y bisel
  Bevel1.Parent := PageControl1.ActivePage;
  Imagen1.Parent := PageControl1.ActivePage;
end;
```

Al hacer clic sobre el botón **Atrás**, que no depende de la ficha, el programa extrae la última hoja de la lista, borra su entrada y se mueve a dicha ficha:

```
procedure TForm1.btnAtrasClick(Sender: TObject);
var
  UltimaPagina: TTabSheet;
begin
  // salta a la última página
  UltimaPagina := TTabSheet (BackPages [BackPages.Count
             - 1]);
  PageControl1.ActivePage := UltimaPagina;
  // borra la última página de la lista
  BackPages.Delete (BackPages.Count - 1);
  // desactiva el botón Atrás
  btnAtras.Enabled := not (BackPages.Count = 0);
  // mueve imagen y bisel
  Bevel1.Parent := PageControl1.ActivePage;
  Imagen1.Parent := PageControl1.ActivePage;
end;
```

Con este código, el usuario puede moverse varias fichas atrás hasta que la lista quede vacía, momento en el que el botón **Atrás** se desactiva.

El resto del código del programa muestra algunas direcciones de Internet. En realidad, como la mayoría de las etiquetas del asistente muestra direcciones http, el usuario puede hacer clic sobre ellas para abrirlas en el explorador que su sistema operativo tenga predefinido. Para ello, extraemos la dirección de la etiqueta y llamamos a la función **`ShellExecute`**.

```
procedure TForm1.LabelLinkClick(Sender: TObject);
var
  Leyenda, StrUrl: string;
begin
  Leyenda := (Sender as TLabel).Caption;
  StrUrl := Copy (Leyenda, Pos ('http://', Caption),
          1000);
  ShellExecute (Handle, 'open', PChar (StrUrl), '', '',
          sw_Show);
end;
```

Este método está unido al evento `OnClick` de varias etiquetas del formulario, que se han convertido en enlaces al configurar la propiedad **`Cursor`** como `crHandPoint`. Esta es una de las etiquetas:

```
object Label4: TLabel
       Cursor = crHandPoint
       Caption = 'http://www.davidarboledas.es'
       OnClick = LabelLinkClick
end
```

## 12.2 EL CONTROL *TTOOLBAR*

Para diseñar una barra de herramientas, Lazarus incluye el componente **`TToolBar`**, en la pestaña *Common Controls* de la Paleta de componentes. Este componente facilita una barra de herramientas con sus propios botones y presenta muchas capacidades avanzadas. Para utilizarlo, se coloca sobre el formulario y, a continuación, desde el menú contextual de la barra de herramientas, se añaden los botones, separadores y divisores necesarios.

La barra de herramientas está formada por componentes de la clase `TToolButton`. Estos tienen una propiedad `Style` que determina su comportamiento:

- **El estilo `tbsButton`**. Hace referencia a un botón pulsador estándar.
- **El estilo `tbsCheck`**. Indica un botón que se comporta como una casilla de verificación, o de un botón de radio si el botón está agrupado con otros en su bloque.
- **El estilo `tbsDropDown`**. Muestra un botón desplegable, como un cuadro combinado. Esta parte desplegable se implementa fácilmente en Lazarus con un control **`TPopupMenu`** conectado a la propiedad `DropdownMenu` del control, como veremos en el ejemplo siguiente.
- **Los estilos `tbsDivide` y `tbsSeparator`**. Muestran separadores con líneas verticales diferentes o sin ellas, dependiendo de cómo se defina la propiedad **`Flat`** del control `TToolBar`.

Si la barra de herramientas que se desea diseñar en el formulario va a ser gráfica, se debe añadir un componente **`TImageList`** al mismo, sobre el que se cargarán los mapas de bits que constituirán las imágenes de los botones. A continuación, solo faltaría conectar la propiedad **`Images`** de la barra de herramientas

con el componente `TImageList`. Por defecto, las imágenes se asignan a los botones en el mismo orden en el que aparecen, pero puede cambiarse su comportamiento fijando la propiedad **`ImageIndex`** de cada botón `TToolButton`.

### 12.2.1 El ejemplo *ToolBarDemo*

Como ejemplo de diseño de una barra de herramientas, hemos creado la aplicación **`toolbardemo`**, que demuestra la utilización de un sencillo menú enlazado a una barra de herramientas. Posee un componente `TMemo` con el que se podría trabajar perfectamente utilizando las herramientas recogidas en la barra. El programa tiene botones para abrir y guardar archivos, para cambiar el formato del texto y su alineación en el componente `Memo`.

Aquí nos centraremos en las características específicas del componente **`TToolBar`** empleado en el ejemplo y visible en la figura 12.4. Esta barra de herramientas tiene botones, separadores e incluso un menú desplegable con la alineación de texto.

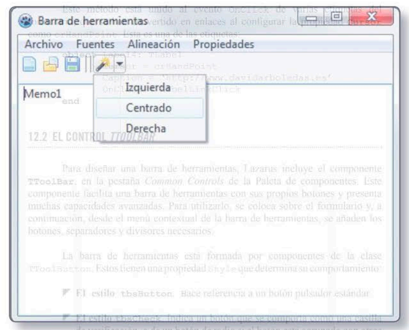



**Figura 12.4.** Menú y barra de herramientas del ejemplo ToolBarDemo

Los distintos botones, los accesos por menú y atajos de teclado no implementan las acciones reales que deberían cubrir en la aplicación, sino tan solo un mensaje que advierte al usuario del botón o acción del menú seleccionados, lo que se consigue con los siguientes procedimientos:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  with (Sender as TToolButton) do
    begin
    ShowMessage('Se ha pulsado el botón de nombre ' +
            Name);
    end;
end;

procedure TMainForm.DoMenuClick (Sender: TObject);
begin
  with (Sender as TMenuItem) do
    begin
    ShowMessage('Se ha seleccionado el elemento del
            menú ' + Caption);
    end;
end;
```

Además de los tres botones (**Nuevo**, **Abrir** y **Guardar**), la barra de herramientas del ejemplo posee un menú desplegable (**Alinear**), una característica compartida por muchas aplicaciones habituales. Este botón desplegable permitiría seleccionar la alineación del texto escrito en el componente `TMemo`. Observe la definición de estos botones en la barra de herramientas:

```
object tbPpal: TToolBar
    object btnNuevo: TToolButton
      Hint = 'Nuevo'
      Caption = '&Nuevo'
      OnClick = ButtonClick
    end
    object btnAbrir: TToolButton
      Hint = 'Abrir'
      Caption = '&Abrir'
      OnClick = ButtonClick
    end
    object btnGuardar: TToolButton
      Hint = 'Guardar'
      Caption = '&Guardar'
      OnClick = ButtonClick
    end
    object ToolButton1: TToolButton
      Style = tbsSeparator
```

```
    end
    object btnAlinear: TToolButton
      Hint = 'Alinear'
      Caption = 'Alineación'
      DropdownMenu = pmAlin
      OnClick = ButtonClick
      Style = tbsDropDown
    end
  end
```

Fíjese en que hemos hecho la implementación del menú desplegable del botón **Alinear** de la barra de herramientas con un control `TPopupMenu` de nombre `pmAlin` conectado a la propiedad **DropdownMenu** del botón `btnAlinear`. Así mismo, el ejemplo usa las propiedades **ShowHint** del componente `TToolBar` y **Hint** de los `TToolButton` para mostrar sugerencias contextuales cuando el ratón se sitúa sobre el icono correspondiente.

## 12.3 EL COMPONENTE *TACTIONLIST*

La arquitectura de eventos de Lazarus es muy abierta, pues es posible escribir un sencillo controlador de eventos y conectarlo a los eventos `OnClick` de un botón de la barra de herramientas y a un menú. Se puede, incluso, conectar el mismo controlador de eventos a diferentes botones o elementos de menú, dado que el controlador puede utilizar el parámetro **Sender** para referirse al objeto que lanzó el evento. Es algo más difícil sincronizar el estado de los botones de la barra de herramientas y los elementos de menú. Si dispone de un elemento de menú y un botón de la barra de herramientas y ambos accionan la misma operación, cada vez que se active esta tendría que añadir la marca de comprobación al elemento de menú y cambiar el estado del botón para que aparezca como pulsado.

Para superar este problema, Lazarus incluye una estructura de gestión de eventos basada en acciones. Una **acción** indica tanto la operación que se realiza cuando se pulsa un elemento de menú o botón que determina el estado de todos los elementos conectados a dicha acción. La conexión de la acción con la interfaz de usuario de los controles enlazados resulta muy importante y es el ámbito en el que podemos entender las auténticas ventajas de esta estructura.

En esta estructura de manipulación de eventos participan diversos agentes. La acción principal la realizan los objetos de la acción. Un objeto de acción tiene un nombre, como cualquier otro componente, y unas propiedades que se aplicarán a

los controles enlazados. Entre dichas propiedades están `Caption`, la representación gráfica (`ImageIndex`), el estado (`Checked`, `Enable` y `Visible`) y la información para el usuario (`Hint` y `Helpcontext`). También están la propiedad `Autocheck` para acciones de dos estados, el soporte de ayuda y la propiedad `Category`, utilizada para organizar las acciones en grupos lógicos.

La clase básica para todos los objetos de acción es **`TBasicAction`**, que introduce el comportamiento abstracto fundamental de una acción, sin ningún enlace específico ni corrección. La clase derivada **`TContainedAction`** introduce propiedades y métodos que permiten que las acciones aparezcan en una lista de acciones o administrador de acciones. La clase derivada **`TCustomAction`** introduce soporte para las propiedades y métodos de los elementos de menú y controles que están enlazados a los objetos de acción.

Cada control enlazado está unido a uno o más objetos clientes a través de un objeto **`ActionLink`**. Como indica su propiedad `Action`, varios controles de diferentes tipos pueden compartir el mismo objeto de acción.

Técnicamente, los objetos `ActionLink` mantienen una conexión bidireccional entre el objeto cliente y la acción. El objeto `ActionLink` es necesario porque la conexión funciona en ambas direcciones. Una operación realizada sobre el objeto, como un sencillo clic, se reenvía al objeto de acción y origina una llamada a su evento **`OnExecute`**, así como una actualización del estado del objeto de acción se refleja en los controles clientes conectados.

Normalmente, los controles de cliente que se conectan a acciones son elementos de menú y diversos tipos de botones (botones pulsador, casillas de verificación, botones de radio, botones de la barra de herramientas y similares).

Por último, los objetos de acción se encuentran dentro de un componente **`TActionList`**, la única clase de la estructura básica que aparece en la Paleta de componentes, concretamente en la pestaña de controles estándar. La lista de acciones recibe las acciones ejecutadas que no controlan los objetos de acción específicos y activa `OnExecuteAction`. El componente `TActionList` tiene un editor especial, accesible desde su menú contextual, que se puede utilizar para crear diversas acciones estándar desde su menú **Nueva acción estándar** o mediante la combinación **[Ctrl]** + **[Ins]**, como se muestra en la figura 12.5.

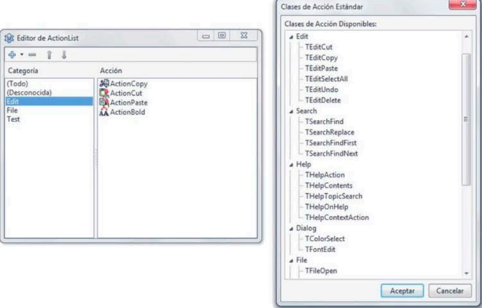



**Figura 12.5.** El editor de ActionList, con una lista de acciones predefinidas

En el **editor**, las acciones aparecen en categorías, como indica su propiedad **`Category`**. Estas categorías son básicamente grupos lógicos, aunque en algunos casos un grupo de acciones puede funcionar solo con un tipo específico de componente de destino.

## 12.3.1 Acciones predefinidas en Lazarus

Con la lista de acciones y el editor de `ActionList`, se puede crear una acción nueva o escoger una de las acciones ya registradas en Lazarus y listadas en el cuadro de diálogo secundario **Clases de acción estándar**, como se ha visto en la figura 12.5. Hay muchas acciones predefinidas que pueden dividirse en grupos lógicos:

- **Acciones de edición**. Cortar, copiar, pegar, seleccionar todo, deshacer y borrar.
- **Acciones de búsqueda**. Buscar, reemplazar, buscar primero, buscar siguiente:
- **Acciones de ayuda**. Permiten activar la página de contenidos o el índice del archivo de ayuda de la aplicación.
- **Acciones de diálogo**. Activan color y fuente.
- **Acciones de archivo**. Abrir, abrir con, guardar como y salir.
- **Acciones de conjuntos de datos**. Relacionadas con tablas de bases de datos y con consultas. Todas las operaciones que se pueden realizar en un conjunto de datos.

Además de manejar el evento OnExecute de la acción y cambiar el estado de esta para causar un efecto en la interfaz de usuario de los controles clientes, una acción puede controlar también el evento OnUpdate, que se activa cuando la aplicación no está en uso. Esto proporciona la oportunidad de verificar el estado del sistema y cambiar la interfaz de usuario de los controles en función de ello. Por ejemplo, la acción estándar TEditCut activa los controles de cliente solo cuando hay algún texto seleccionado en el portapapeles.

#### 12.3.1.1 EL EJEMPLO *ACCIONES*

Ahora que se entienden las ideas principales de esta característica tan importante de Lazarus, estudiaremos el programa **acciones** para comprender cómo funcionan las acciones de forma práctica. En esta aplicación hemos colocado un componente TActionList en el formulario y añadido tres acciones estándar de edición y algunas personalizadas.

El formulario tiene también una barra de herramientas, un menú principal y un control TMemo, que es el objetivo de las acciones de edición implementadas.

Observe las acciones definidas para la aplicación extraídas de su archivo accionesF.lfm:

```
object ActionList1: TActionList
    Images = ListaIconos
    left = 96
    top = 152
    object AccionCopiar: TEditCopy
      Category = 'Editar'
      Caption = '&Copiar'
      Hint = 'Copiar'
      ImageIndex = 1
      ShortCut = 16451
    end
    object AccionCortar: TEditCut
      Category = 'Editar'
      Caption = 'Cor&tar'
      Hint = 'Cortar'
      ImageIndex = 0
      ShortCut = 16472
    end
    object AccionPegar: TEditPaste
      Category = 'Editar'
```

```
    Caption = '&Pegar'
    Hint = 'Pegar'
    ImageIndex = 2
    ShortCut = 16470
  end
  object AccionNuevo: TAction
    Category = 'Archivo'
    Caption = '&Nuevo'
    Hint = 'Nuevo'
    ImageIndex = 3
    OnExecute = AccionNuevoExecute
    ShortCut = 113
  end
  object AccionSalir: TAction
    Category = 'Archivo'
    Caption = 'S&alir'
    Hint = 'Salir'
    ImageIndex = 5
    OnExecute = AccionSalirExecute
    ShortCut = 32883
  end
  object NoAction: TAction
    Category = 'Pruebas'
    Caption = '&No Action'
    Hint = 'No Action'
  end
  object AccionContar: TAction
    Category = 'Pruebas'
    Caption = '&Contar Caracteres'
    Hint = 'Cuenta caracterres'
    ImageIndex = 6
    OnExecute = AccionContarExecute
    OnUpdate = AccionContarUpdate
  end
  object AccionNegrita: TAction
    Category = 'Editar'
    Caption = '&Negrita'
    Hint = 'Negrita'
    ImageIndex = 4
    OnExecute = AccionNegritaExecute
    ShortCut = 16450
  end
  object ActivarAccion: TAction
    Category = 'Pruebas'
    Caption = '&Activar NoAction'
```

```
    Caption = '&Pegar'
    Hint = 'Pegar'
    ImageIndex = 2
    ShortCut = 16470
  end
  object AccionNuevo: TAction
    Category = 'Archivo'
    Caption = '&Nuevo'
    Hint = 'Nuevo'
    ImageIndex = 3
    OnExecute = AccionNuevoExecute
    ShortCut = 113
  end
  object AccionSalir: TAction
    Category = 'Archivo'
    Caption = 'S&alir'
    Hint = 'Salir'
    ImageIndex = 5
    OnExecute = AccionSalirExecute
    ShortCut = 32883
  end
  object NoAction: TAction
    Category = 'Pruebas'
    Caption = '&No Action'
    Hint = 'No Action'
  end
  object AccionContar: TAction
    Category = 'Pruebas'
    Caption = '&Contar Caracteres'
    Hint = 'Cuenta caracterres'
    ImageIndex = 6
    OnExecute = AccionContarExecute
    OnUpdate = AccionContarUpdate
  end
  object AccionNegrita: TAction
    Category = 'Editar'
    Caption = '&Negrita'
    Hint = 'Negrita'
    ImageIndex = 4
    OnExecute = AccionNegritaExecute
    ShortCut = 16450
  end
  object ActivarAccion: TAction
    Category = 'Pruebas'
    Caption = '&Activar NoAction'
```

```
      Hint = 'Activa No Action'
      OnExecute = ActivarAccionExecute
    end
    object AccionRemitente: TAction
      Category = 'Pruebas'
      Caption = 'Probar &Remitente'
      Hint = 'Probar Remitente
      OnExecute = AccionRemitenteExecute
    end
  end
```

**NOTA**
Las teclas de método abreviado se almacenan en los archivos LFM con números de teclas virtuales.

Todas las acciones están conectadas a los elementos del componente **TMainMenu** que forma el menú principal de la aplicación. Algunas de las acciones, también, se conectan a los botones de los controles `TToolBar`.

Como muestra la figura 12.6, las imágenes seleccionadas en el control `TActionList` afectan solo a las acciones del editor. Para que se muestren también en los elementos del menú y en los botones de la barra de herramientas, es necesario seleccionar también la lista de imágenes en los componentes `TToolBar` y `TMainMenu`.

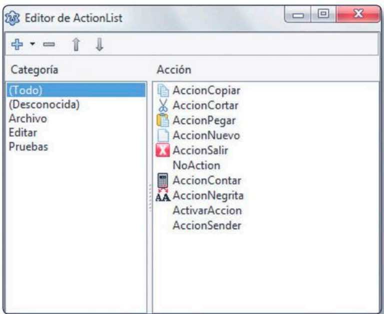



**Figura 12.6.** El editor de ActionList del ejemplo Acciones

Las tres acciones predeterminadas del menú **Editar** (Copiar, Cortar y Pegar) no tienen controladores asociados, pero estos objetos especiales tienen un código interno para realizar la acción relacionada con el control memo activo. Estas acciones se activan y desactivan también a sí mismas, dependiendo del contenido del portapapeles y de la existencia de texto seleccionado en el control de edición activo. La mayoría de las otras acciones tienen un código personalizado, menos en el caso del objeto NoAction. Al no tener código, el elemento de menú y el botón asociado a esta orden están desactivados, aunque la propiedad Enabled de esta acción está definida como True. Hemos añadido al ejemplo y al menú **Pruebas** otra acción que activa el elemento de menú conectado al objeto NoAction:

```
procedure TForm1.ActivarAccionExecute(Sender: TObject);
begin
  NoAction.DisableIfNoHandler := False;
  NoAction.Enabled := True;
  ActivarAccion.Enabled := False;
end;
```

Definir Enabled como True producirá el resultado durante un corto período de tiempo, a menos que se defina la propiedad DisableIfNoHandler, como se ha visto en el apartado anterior. Tras haber realizado esta operación, hay que desactivar la acción en uso, porque no es necesario dar de nuevo la misma orden. Esta situación es distinta a la que se produce cuando activamos una acción, como el elemento del menú **Editar | Negrita** y su correspondiente botón en la barra de herramientas. A continuación, vemos el código para la acción **Negrita**, que tiene su propiedad AutoCheck fijada como False, por lo que resulta necesario modificar el estado de la propiedad Checked en el código:

```
procedure TForm1.AccionNegritaExecute(Sender: TObject);
begin
  with Memo1.Font do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
  // conmuta el estado
  AccionNegrita.Checked := not AccionNegrita.Checked;
end;
```

El objeto **AccionContar** tiene un código muy sencillo, pero muestra el funcionamiento de un controlador `OnUpdate`. Cuando el control de `memo` está vacío, se desactiva automáticamente. A continuación, aparece el código de los dos controladores de esta acción:

```
procedure TForm1.AccionContarExecute(Sender: TObject);
begin
  ShowMessage ('Caracteres: ' + IntToStr (Length
           (Memo1.Text)));
end;

procedure TForm1.AccionContarUpdate(Sender: TObject);
begin
  AccionContar.Enabled := Memo1.Text <> '';
end;
```

Para finalizar, hemos añadido una acción especial que comprueba el objeto remitente del controlador de eventos de la acción y obtiene otra información sobre el sistema. Además de mostrar la clase y nombre del objeto, hemos implementado un código que accede al objeto de la lista de acciones, básicamente para mostrar cómo acceder a esta información:

```
procedure TForm1.AccionRemitenteExecute(Sender: TObject);
begin
  Memo1.Lines.Add (
    'Clase remitente: ' + Sender.ClassName);
  Memo1.Lines.Add (
    'Nombre del remitente: ' + (Sender as TComponent)
             .Name);
  Memo1.Lines.Add (
    'Categoría: ' + (Sender as TAction).Category);
  Memo1.Lines.Add (
    'Nombre del control TActionList: ' + (Sender as
             TAction).ActionList.Name );
end;
```

Se puede ver el resultado de este código en la figura 12.7, junto con la interfaz de usuario del ejemplo. Observe que el `Remitente` no es el elemento de menú seleccionado, aunque el controlador está conectado a él. El objeto `Sender` que activa el evento es la acción que intercepta la operación de usuario.

Por último, hay que tener presente que también se pueden escribir controladores para eventos del propio objeto `ActionList` que representen el papel de controladores globales para todas las acciones de la lista y para el objeto global **`TApplication`**, que se dispara para todas las acciones de la aplicación. Antes de invocar al evento `OnExecute` de la acción, Lazarus activa el evento `OnExecute` de la `ActionList` y el evento `OnActionEvent` del objeto global `TApplication`. Estos eventos se fijarán en la acción, ejecutando eventualmente algo de código compartido, y después detendrán la ejecución mediante el parámetro `Handled` o dejarán que se propague hasta el siguiente nivel.

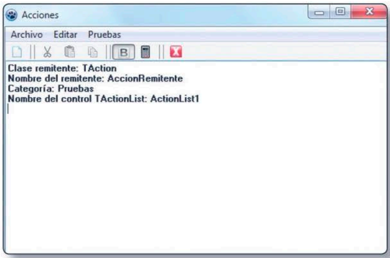
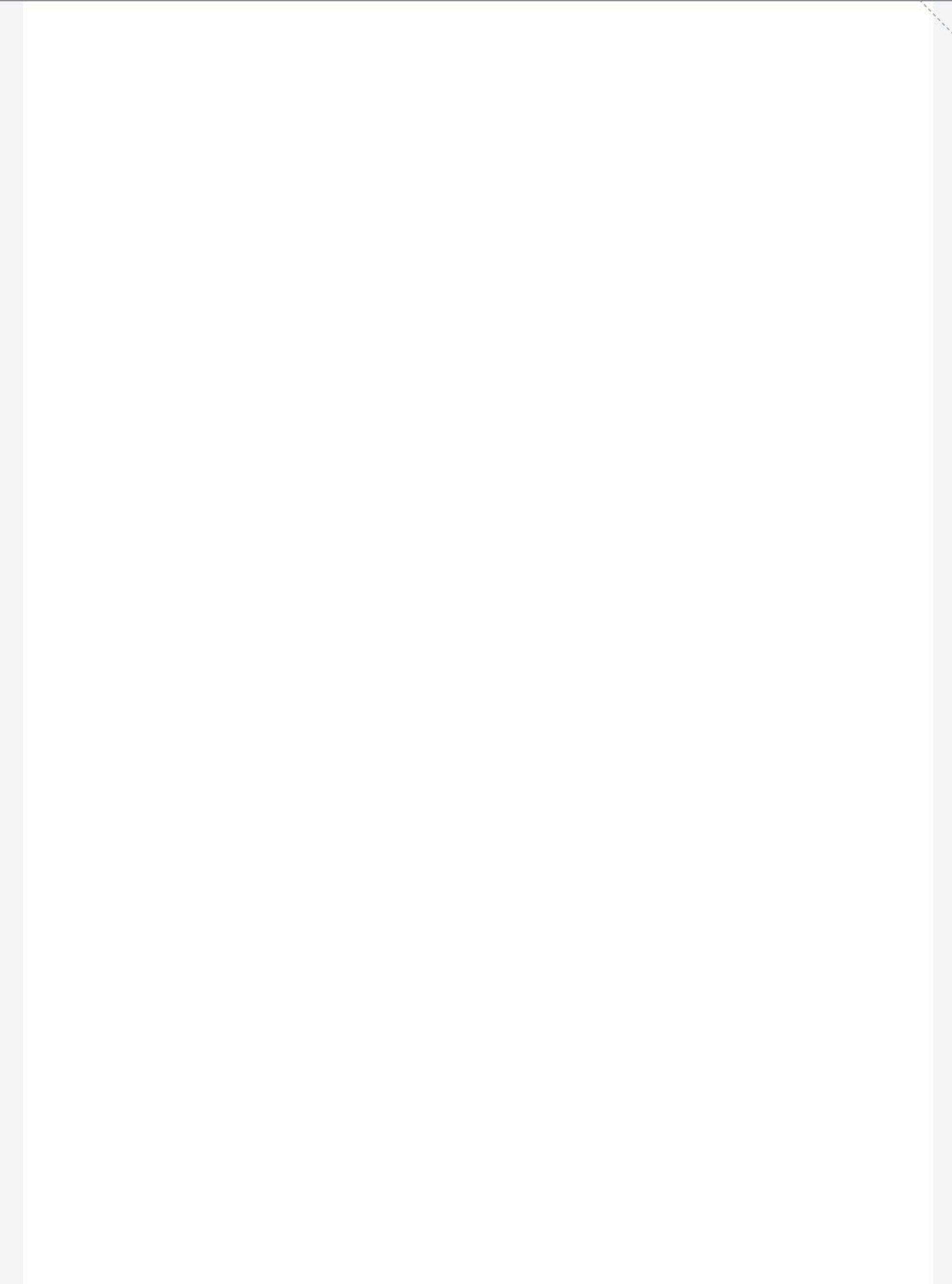



**Figura 12.7.** El ejemplo Acciones, con una descripción del remitente (Sender) del evento OnExecute de un objeto.



# 13

# CLASES NO VISUALES

Existen cientos de clases declaradas en las bibliotecas FCL y LCL que se emplean en el diseño de interfaces gráficas de usuario sin que los programadores sean conscientes *a priori*. Esto es así porque los controles visuales dependen a menudo de otras muchas clases, además de su ancestro original.

Las clases de Lazarus pueden utilizarse completamente desde el código o desde el diseñador visual de formularios. Algunas de ellas son clases componentes, que aparecerán en la Paleta de componentes, otras son de propósito más general, como las no visuales.

En este capítulo nos centraremos en algunas clases principales de la biblioteca que, aun siendo no visuales, son muy importantes para construir programas gráficos robustos. Exploraremos las clases más habitualmente utilizadas, como las colecciones, listas, listas de cadenas y *streams* o flujos. El hecho de que no sean componentes impide que podamos encontrarlas en la Paleta de componentes y arrastrarlas al formulario.

## 13.1 LA CLASE *TPERSISTENT*

Todos los controles que podamos arrastrar sobre un formulario y manipular desde el Inspector de objetos dependen para su correcto funcionamiento en tiempo de ejecución de la información suministrada por la *RTTI*. La clase base que introduce este comportamiento es la que se denomina **`TPersistent`**. Esta clase de Lazarus es bastante atípica, pues tiene poco código y casi no tiene uso directo, pero es la que nos da la base para una programación visual. Como indica el propio nombre de la clase,

controla la permanencia, es decir, el hecho de almacenar el valor de un objeto en un fichero para usarlo más tarde y volver a crear el objeto en el mismo estado y con idénticos datos. La **permanencia** es el elemento estrella de la programación visual. De hecho, en Lazarus se manipulan objetos reales en tiempo de diseño que se guardan en archivos LFM y se vuelven a crear en tiempo de ejecución al mismo tiempo que el contenedor del componente, sea este un formulario o un módulo de datos.

De los métodos definidos en la clase `TPersistent`, el único que se empleará, como norma general, es el procedimiento **`Assign`**, que puede utilizarse para copiar el valor real de un objeto.

Un gran número de clases descienden en Lazarus de `TPersistent`, entre las que se incluyen todos los componentes, que heredan, a su vez, de `TComponent` y todas las clases `TStrings`, así como las clases `TCollection` y `TCollectionItem`, como puede ver en la figura 13.1.

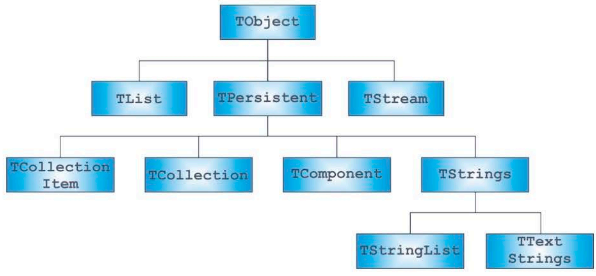



**Figura 13.1.** Jerarquía inicial de clases en la biblioteca de componentes de Lazarus

## 13.2 COLECCIONES

Las **colecciones** contienen solo dos clases: `TCollection` y `TCollectionItem`. `TCollection` define una lista homogénea de objetos. Los objetos de la colección han de descender de la clase `TCollectionItem`. Si se necesita una colección que almacene objetos específicos, hay que crear una subclase de `TCollection` y una subclase correspondiente de `TCollectionItem`. Las **colecciones** son clases diseñadas para trabajar con datos de elementos estructurados de un modo semejante, de la misma forma que hacemos cuando empleamos tablas

de valores. Con este objetivo, vamos a implementar un programa para mostrar de forma tabulada los nombres, símbolos y valencias de los doce primeros elementos de la tabla periódica, para, de este modo, aprender a trabajar con colecciones con un componente *RTTI*, `TTIGrid`.

Abra un nuevo proyecto de nombre **valencias** con una unidad llamada **valenciasF.pas**. En el Inspector de objetos dé el nombre `ValColeccion` al formulario. Arrastre sobre él desde la pestaña *RTTI* el componente `TTIGrid` y renómbrelo como `Tabla`.

La clase `TCollection` siempre trabaja junto con `TCollectionItem`. En el ejemplo vamos a usar una pequeña clase llamada `TElementos` que poseerá cuatro propiedades relacionadas con cada elemento químico, basadas en sus correspondientes campos privados: `FNombre`, `FSimbolo`, `FValencia` y número atómico (`FZ`). Las propiedades se implementan mediante accesos de lectoescritura (**read/write**).

`TElementos` se declara como descendiente de `TCollectrionItem`. Puesto que `TCollectionItem` es descendiente de `TPersistent`, esto le asigna completa funcionalidad en tiempo de ejecución sin necesidad de escribir ni una línea de código.

En la declaración de tipos de la unidad, añada la declaración para `TElementos` como sigue:

```
type
  TElementos = class(TCollectionItem)
  private
   FNombre: string;
   FSimbolo: string;
   FValencia: string;
   FZ: byte;
  published
   property Nombre: string read FNombre write FNombre;
   property Simbolo: string read FSimbolo write FSimbolo;
   property Valencia: string read FValencia write FValencia;
   property Z: byte read FZ write FZ;
  end;
```

La única idea destacable aquí es el uso de `byte` como tipo para el número atómico, porque incluso los elementos artificiales no superan el número 117. `TElementos` es básicamente una clase que contiene datos que tiene la funcionalidad de `TCollectionItem`, adquirida mediante herencia.

Añada **grids** al conjunto de unidades recogidas en la cláusula **uses** y, en la sección **privada** de la clase formulario, añada un campo llamado FElementos de tipo TCollection, seguido de un procedimiento llamado IniciarTabla. Ahora use la compleción automática de código **[Ctrl] + [May] + [C]** para que Lazarus escriba el esqueleto por usted. Esto creará el siguiente procedimiento:

```
procedure TValColeccion.IniciarTabla;
begin

end;
```

En este método añadiremos los elementos a la colección e iniciaremos algunas propiedades de la tabla, el componente TTIGrid, mediante tres subprocedimientos (AnadirElemento, AnadirElementos e IniciarRejilla) para facilitar el flujo de programa. Este es el código que debe escribir:

```
procedure TValColeccion.IniciarTabla;
  procedure AnadirElemento(aZ: byte; const aNombre,
            aSimbolo, aValencia: string);
  var ei: TElementos;
  begin
    ei := TElementos(Felementos.Add);
    ei.Z:= aZ;
    ei.Nombre:= aNombre;
    ei.Simbolo:= aSimbolo;
    ei.Valencia:= aValencia;
  end;
  procedure AnadirElementos;
  begin
    AnadirElemento(1, 'Hidrógeno', 'H', '1');
    AnadirElemento(2, 'Helio', 'He', '0');
    AnadirElemento(3, 'Litio', 'Li', '1');
    AnadirElemento(4, 'Berilio', 'Be', '2');
    AnadirElemento(5, 'Boro', 'B', '3');
    AnadirElemento(6, 'Carbono', 'C', '2, 4');
    AnadirElemento(7, 'Nitrógeno', 'N', '1, 2, 3, 4, 5');
    AnadirElemento(8, 'Oxígeno', 'O', '2');
    AnadirElemento(9, 'Flúor', 'F', '1');
    AnadirElemento(10,'Neón', 'Ne', '0');
    AnadirElemento(11,'Sodio', 'Na', '1');
    AnadirElemento(12,'Magnesio', 'Mg', '2');
  end;
  procedure IniciarRejilla;
```

```
    begin
      Self.Height:= 310;
      Self.Width:= 340;
      Tabla.Align:= alClient;
      Tabla.TIOptions:= Tabla.TIOptions + [tgoStartIndexAtOne];
      Tabla.Options:= Tabla.Options + [goDrawFocusSelected];
      Tabla.FixedColor:= clSkyBlue;
      Tabla.DefaultDrawing:= True;
      Tabla.AutoFillColumns:= True;
    end;
  begin
    FElementos:= TCollection.Create(TElementos);
    AnadirElementos;
    IniciarRejilla;
    Tabla.ListObject:= FElementos;
  end;
```

Observe que hay dos aspectos interesantes e inusuales a la hora de trabajar con colecciones. En primer lugar, diseñamos la colección invocando a su constructor `Create()`, al que pasamos el parámetro que indica qué tipo de colección será:

```
FElementos := TCollection.Create(TElementos);
```

Por otro lado, más que crear explícitamente cada instancia `TElementos` con una llamada al constructor `Create`, lo hacemos con el método `Add` de la clase:

```
ei := TElementos(FElementos.Add);
```

`FElementos.Add` inicia un nuevo ítem (`TCollectionItem`) y lo añade a la colección. Como este método de añadir objetos es general para cualquier descendiente de `TCollectionItem`, tenemos que asignarle un tipo, `TElementos` en este ejemplo.

Ahora solo queda iniciar la tabla de datos, lo que hacemos con una llamada al procedimiento `IniciarTabla` nada más construir el formulario. Para ello, con el formulario seleccionado, seleccione la pestaña **Eventos** del Inspector de objetos y haga doble clic junto al evento `OnCreate` para que Lazarus genere el código del manejador; complételo con la llamada al procedimiento:

```
procedure TValColeccion.FormCreate(Sender: TObject);
begin
  IniciarTabla;
end;
```

En este momento solo es necesario dar un paso más para tener a punto el programa. Una vez que se cierre la aplicación, deberíamos asegurarnos de liberar la memoria que se ha reservado con el constructor `FElementos`, y es un error común olvidarse de ello. Así pues, en la hoja **Eventos** del IO haga doble clic junto al evento `OnDestroy` para crear el manejador que se invocará cuando el formulario se vaya a destruir. Complételo con la llamada al destructor:

```
procedure TValColeccion.FormDestroy(Sender: TObject);
begin
  FElementos.Free;
end;
```

¿Por qué no necesitamos liberar todas las instancias `TElementos` que se han creado? La respuesta es sencilla, se debe a que `FElementos` elimina toda la colección cuando hacemos lo propio con él. Es parte de la funcionalidad de la clase `TCollectionItem` que se heredó cuando se declaró `TElementos` como descendiente de aquella. Si algo ya existe y está probado, no es recomendable inventar otra solución. Tenga esto siempre en cuenta en su carrera como programador. Es más, a medida que vaya estudiando más y más código, más experto se hará. Un programador se hace, y se hace estudiando programas, una de las mayores ventajas de emplear software de código abierto. Aprovéchela.

Compile y ejecute el programa. Verá en unos instantes una tabla con los doce primeros elementos de la tabla periódica con los cuatro parámetros que le pasamos (Figura 13.2).

ValColeccion

| | Nombre | Simbolo | Valencia | Z |
|---|---|---|---|---|
| 1 | Hidrógeno | H | 1 | 1 |
| 2 | Helio | He | 0 | 2 |
| 3 | Litio | Li | 1 | 3 |
| 4 | Berilio | Be | 2 | 4 |
| 5 | Boro | B | 3 | 5 |
| 6 | Carbono | C | 2, 4 | 6 |
| 7 | Nitrógeno | N | 1, 2, 3, 4, 5 | 7 |
| 8 | Oxígeno | O | 2 | 8 |
| 9 | Flúor | F | 1 | 9 |
| 10 | Neón | Ne | 0 | 10 |
| 11 | Sodio | Na | 1 | 11 |
| 12 | Magnesio | Mg | 2 | 12 |

**Figura 13.2.** Aplicación valencias en tiempo de ejecución

Con este ejemplo hemos demostrado que a veces es muy útil poder iniciar controles mediante código, en vez de usar el Inspector de objetos.

## 13.3 LISTAS Y LISTAS DE CADENA

En programación suele ser importante controlar grupos de componentes. Además de matrices, existen unas cuantas clases no visuales que se usan ampliamente en los componentes de la biblioteca LCL que representan listas de otros objetos, como colecciones y listas.

Las **listas** se representan mediante la lista genérica de objetos `TList`, y con las dos listas de cadenas `TStrings` y `TStringList` (Figura 13.1):

- **`TList`**. Define una lista de punteros que se usa para almacenar objetos de cualquier clase. Una lista de esta clase es más versátil que una matriz dinámica, pues se amplía de forma automática mediante la adición de nuevos elementos.

- **`TStrings`**. Es una clase abstracta descendiente de `TPersistent` utilizada para representar todas las formas de las listas de cadena. Una **clase abstracta** es aquella que no posee ninguna instancia actual, sino que se diseña como ancestro de una familia de clases más especializadas. Como `TStrings` se implementa para trabajar y manipular listas de cadena sin posibilidad de almacenarlas, los objetos `TStrings` solo se pueden usar como propiedades de componentes que sí sean capaces de almacenar las propias cadenas.

- **`TStringList`**. Es una subclase de `TStrings` y define una lista de cadenas con su propio almacenamiento. Se usa, entonces, para definir listas de cadenas en un programa. Es por ello una de las clases de la biblioteca RTL más ampliamente utilizada en Lazarus.

Los objetos `TStringList` y `TStrings` poseen ambos una lista de cadenas y una lista de objetos asociados con las mismas, lo que hace posible una serie de usos diferenciados para ambas clases. Las dos clases de listas de cadenas también tienen métodos preparados para guardar o cargar sus contenidos en un archivo de texto: **`SaveToFile`** y **`LoadFromFile`**.

Para movernos a lo largo de una lista, se puede usar una sencilla instrucción basada en su índice, como si la lista fuera una matriz. Puede, por ello, emplearse la notación matricial (`[y]`), tanto para leer como para cambiar elementos. Existe, así mismo, una propiedad **`Count`**, así como métodos de acceso comunes, como `Add`, `Insert`, `Delete` y `Remove`; y métodos de búsqueda, como **`IndexOf`**.

Es más, la clase TList posee un método Assign que, además de copiar los datos fuente, puede realizar operaciones lógicas en las dos listas, como and, or y xor, entre otras.

Para rellenar una lista de cadenas con elementos y verificar después si uno de ellos está o no presente, se puede escribir un sencillo código como el siguiente:

```
var
   cadena: TStringList;
   indice: Integer;
begin
   cadena := TStringList.Create;
   try
      cadena.Add ('Susi');
      cadena.Add ('Marta');
      cadena.Add ('Irene');
      // para verificar la existencia de una cadena
      indice := cadena.IndexOf  ('Marta');
      if indice >= 0 then
         ShowMessage ('Cadena encontrada');
   finally
      cadena.Free;
   end ;
end;
```

## 13.3.1 Pares nombre-valor

La clase TStringList siempre ha tenido una característica muy cómoda: el soporte de pares nombre-valor. Si se añade a una lista una cadena de la forma 'hija=Marta', puede buscarse la existencia del par empleando la función **IndexOfName** o la propiedad de matriz **Values**. Por ejemplo, si la variable anterior de tipo TStringList tiene por nombre cadena, puede obtenerse el valor 'Marta' mediante una llamada a cadena.Values['hija'].

Se puede utilizar esta característica para crear estructuras de datos mucho más complejas, como diccionarios, y beneficiarse todavía de la posibilidad de adjuntar un objeto a la cadena.

Esta estructura de datos se proyecta directamente sobre los archivos de iniciación y otros formatos habituales.

Lazarus amplía aún más las posibilidades del soporte de pares nombre-valor ofreciendo acceso directo a la parte del valor de una cadena que se encuentre en una posición dada con la propiedad **ValueFromIndex**:

```
MiStringList.ValueFromIndex [I];
```

### 13.3.2 Trabajar con listas de cadenas

Podemos escribir un ejemplo que se centre en el uso de la clase `TStringList`. A diferencia de su ancestro `TStrings`, `TStringList` permite almacenar en un arreglo un máximo teórico de $2^{27}$ cadenas. Puede manejar textos en los que cada línea sea de la forma `nombre = valor` y textos en los que cada línea sea una serie de frases separadas por comas. Proporciona, además, métodos de ordenación de cadenas y puede leerlas y guardarlas en archivos en disco.

El ejemplo **palabras** demuestra algunas de las capacidades de la clase `TStringList`. La aplicación analiza un texto y devuelve el número total de palabras y el de palabras no repetidas ordenadas alfabéticamente.

El formulario de la aplicación posee un componente `TLabel`, un `TListBox` y un `TMemo`, como se puede ver en la figura siguiente:

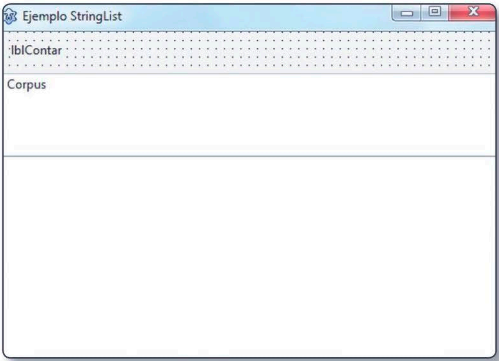



**Figura 13.3.** El ejemplo palabras en tiempo de diseño

Lazarus amplía aún más las posibilidades del soporte de pares nombre-valor ofreciendo acceso directo a la parte del valor de una cadena que se encuentre en una posición dada con la propiedad **ValueFromIndex**:

```
MiStringList.ValueFromIndex [I];
```

## 13.3.2 Trabajar con listas de cadenas

Podemos escribir un ejemplo que se centre en el uso de la clase `TStringList`. A diferencia de su ancestro `TStrings`, `TStringList` permite almacenar en un arreglo un máximo teórico de $2^{27}$ cadenas. Puede manejar textos en los que cada línea sea de la forma `nombre = valor` y textos en los que cada línea sea una serie de frases separadas por comas. Proporciona, además, métodos de ordenación de cadenas y puede leerlas y guardarlas en archivos en disco.

El ejemplo **palabras** demuestra algunas de las capacidades de la clase `TStringList`. La aplicación analiza un texto y devuelve el número total de palabras y el de palabras no repetidas ordenadas alfabéticamente.

El formulario de la aplicación posee un componente `TLabel`, un `TListBox` y un `TMemo`, como se puede ver en la figura siguiente:

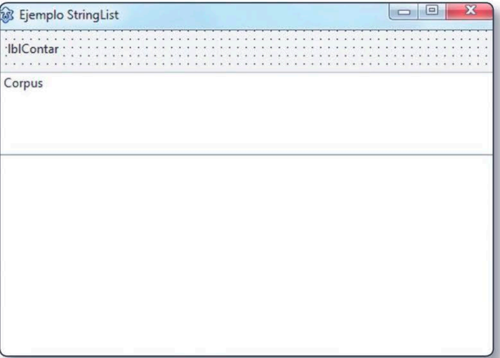



**Figura 13.3.** El ejemplo palabras en tiempo de diseño

La definición de los controles anteriores, tal y como figura en el archivo LFM, es la siguiente:

```
object Form1: TForm1
  Height = 320
  Width = 480
  Caption = 'Ejemplo StringList'
  object lblContar: TLabel
    Left = 10
    Top = 10
  end
  object lbPalabras: TListBox
    Height = 200
    Align = alBottom
    Columns = 4
  end
  object Corpus: TMemo
    Height = 80
    Align = alBottom
  end
end
```

Entre las secciones **var** e **implementation** de la unidad, escriba el texto que formará el corpus de trabajo:

```
const muestra = 'En un lugar de la Mancha, de cuyo nombre
no' + ' quiero acordarme, no ha mucho tiempo que vivía
un' + ' hidalgo de los de lanza en astillero, adarga
antigua,' + ' rocín flaco y galgo corredor.' + ' Una olla
de algo más vaca' + ' que carnero, salpicón las más
noches, duelos y quebrantos' + ' los sábados, lentejas
los viernes,';
```

Haga doble clic sobre el formulario para generar un evento `OnCreate` y complete el esqueleto de la rutina de este modo:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  sl, unicas: TStringList;
  j: integer;
begin
  Corpus.Text:= muestra;
  sl := TStringList.Create;
```

```
      unicas := TStringList.Create;
      try
     sl.CommaText:= muestra;
       sl.Sort;
       for j := 0 to sl.Count-1 do
         if (unicas.IndexOf(sl[j]) < 0)
           then unicas.Add(sl[j]);
       lbPalabras.Items.AddStrings(unicas);
       lblContar.Caption:= Format('El corpus tiene %d
               palabras, de las que %d son únicas',
               [sl.Count, unicas.Count]);
     finally
       sl.Free;
       unicas.Free;
     end;
   end;
```

Compile y ejecute el programa. Observará cómo la aplicación analiza las palabras del corpus para escribir en la etiqueta `lblContar` cuántas hay, y en el control `lbPalabras` una lista con las palabras ordenadas alfabéticamente en cuatro columnas (Figura 13.4).

**Figura 13.4.** El ejemplo palabras en tiempo de ejecución

En el ejemplo hemos empleado dos instancias `TStringList`: `sl` y `unicas`. El componente `TStringList` maneja dos propiedades para separar texto, **`CommaText`** y **`DelimitedText`**. Son similares, excepto que la primera separa las cadenas por comas y `DelimitedText` puede definir en la propiedad `Delimiter`

el carácter deseado para separarlas. En el ejemplo, `sl` trabaja con la propiedad `CommaText`:

```
sl.CommaText := muestra; // separa palabras con comas
```

Además, hemos empleado después el método **Sort** de `TStringList` para ordenar todas las palabras. La segunda lista de cadenas, `unicas`, la utilizamos exclusivamente para filtrar las palabras repetidas en `sl`. Tras crear `unicas`, la lista está vacía y recorremos una a una cada cadena (palabra) en `sl` comprobando si está o no repetida. Esto lo hacemos con la función `IndexOf()`, que devuelve un índice positivo con la posición de una cadena en la lista, si se encuentra en ella, y −1 si está ausente. En este último caso, añadimos la cadena a la lista `únicas` y la copiamos al control `TListBox`, `lbPalabras`, con la propiedad `Items` mediante el método `AddStrings()`:

```
lbPalabras.Items.AddStrings(unicas);
```

El bloque **try...finally...end** asegura que la memoria reservada para la creación de las dos listas de cadenas se libera correctamente tras usarse, incluso aunque se produzca un error.

## 13.4 FLUJOS DE DATOS

Una parcela importante de la biblioteca de clases de Lazarus es el soporte para flujos de datos, que en la literatura anglosajona denominan *streaming*, e incluye administración de datos secuenciales, como ficheros, áreas de memoria, *sockets* y otras fuentes de información organizadas de forma secuencial. La idea del *streaming* consiste en moverse a través de los datos mientras se leen.

### 13.4.1 La clase *TStream*

Lazarus define la clase abstracta `TStream` como una encapsulación orientada a objetos del compilador de Free Pascal para trabajar con secuencias de bytes desde y hacia su almacenamiento. La idea, como hemos dicho, es moverse a lo largo de los datos mientras se leen secuencialmente.

La clase padre, `TStream`, posee solo unas cuantas propiedades y jamás se podrá crear una instancia de la misma por su abstracción, pero sí integra una interesante lista de métodos que, por lo general, se utilizará cuando se trabaje con

sus clases derivadas. Todos los flujos de datos tienen un tamaño específico y habrá de especificarse una posición en la que se quiere leer o escribir la información dentro del flujo. Por tanto, la clase `TStream` define dos propiedades: **`Position`** y **`Size`**. La primera denota la localización donde se ejecutarán las operaciones de lectoescritura en el flujo, mientras que `Size` indica el tamaño actual en bytes. Lógicamente, al escribir información adicional al final del flujo, este crece y `Size` aumenta en consonancia.

La lectura y escritura de bytes depende de la clase de flujo de datos empleado, pero, en cualquier caso, tan solo es necesario saber el tamaño y la posición relativa dentro del flujo de datos para leer o escribir datos.

Además de estas dos propiedades, la clase `TStream` define varios métodos importantes, la mayoría de los cuales son virtuales y abstractos. El uso básico de un flujo de datos implica llamadas a los métodos **`ReadBuffer`** y **`WriteBuffer`**. Por ejemplo, se puede guardar en un archivo una cadena mediante este código:

```
var
  flujo: TStream;
  cadena: string;
begin
  cadena:= 'cadena de prueba';
  flujo:= TFileStream.Create ( 'c:\tmp\test', fmCreate);
  flujo.WriteBuffer (cadena[1] , Length (cadena));
  flujo.Free;
end.
```

Los métodos principales de esta clase son, sin embargo, `CopyFrom`, `Read`, `Seek` y `Write`. `CopyFrom` se utiliza para copiar datos de uno a otro flujo. `Read` lee los datos que comienzan en la posición indicada por `Position`. `Seek` se emplea para cambiar la posición y `Write` escribe los datos del flujo desde la posición indicada en `Position`.

### 13.4.2 Clases de flujos

No tiene sentido crear una instancia de `TStream` porque esta clase es abstracta y no ofrece soporte directo para guardar datos. En su lugar, se puede utilizar una de las clases derivadas para cargar datos desde ella o almacenarlos en un archivo real, un campo de objeto binario ancho, un *socket* o un bloque de memoria.

Existen diversas unidades que definen las clases derivadas de `TStream`. En la unidad `Classes` se encuentran las siguientes clases:

- **`TCustomMemoryStream`**. Es la clase básica para los flujos almacenados en memoria, aunque no se emplea directamente.
- **`TFileStream`**. Define un flujo de datos que manipula un archivo de disco local o de red representado por un nombre de archivo.
- **`THandleStream`**. Define un flujo de datos que manipula un archivo de disco representado por un manejador de archivo.
- **`TMemoryStream`**. Define un flujo que manipula una secuencia de bytes en memoria. Hereda de `TCustomMemoryStream`.
- **`TStringStream`**. Ofrece una forma sencilla de asociar un flujo de datos a una cadena en memoria para poder acceder a la cadena mediante la interfaz `TStream` y copiar la cadena en o desde otro flujo.

Las clases derivadas definidas en otras unidades son, entre otras:

- **`TOleStream`**. Define un flujo para leer y escribir información sobre la interfaz para el *streaming* proporcionado por un objeto OLE.
- **`TWinSocketStream`**. Ofrece soporte de *streaming* para una conexión *socket*.

### 13.4.3 Flujos de datos de archivo

Crear y usar un flujo de datos de archivo es tan sencillo como crear una variable de un tipo que descienda de `TStream` y llamar a los métodos de los componentes para cargar cl contenido desde el archivo:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
  begin
    S:= TFileStream.Create (OpenDialog1.FileName,fmOpenRead);
    try
      Memo1.Lines.LoadFromStream (S);
    finally
      S.Free;
    end;
  end;
end;
```

Como se puede ver en el código, para crear un flujo de datos de fichero tan solo es necesario pasar al constructor `Create()` dos parámetros: el nombre de archivo y un atributo que indica el modo de acceso solicitado, en este caso, `fmOpenRead` para leer el archivo.

Los modos de acceso más interesantes son estos:

- **`fmOpenRead`**. Se emplea para abrir un archivo en modo lectura.
- **`fmOpenWrite`**. Se utiliza para abrir un archivo en modo escritura.
- **`fmOpenReadWrite`**. Es el modo de acceso utilizado para leer y escribir en el fichero.

Y en cuanto a ficheros compartidos:

- **`fmShareDenyWrite`**. Se usa cuando se leen datos de un archivo compartido.
- **`fmShareExclusive`**. Se utiliza para escribir datos en un archivo compartido.
- **`fmShareCompat`**. Abre un archivo compartido compatible con DOS.
- **`fmShareDenyRead`**. Bloquea un archivo compartido para que ningún otro proceso pueda leerlo.
- **`fmShareDenyNone`**. Abre el archivo compartido sin bloquearlo de ningún modo.

**IMPORTANTE**

Recuerde que no puede abrir un fichero si aún no existe. Para crear un archivo nuevo o reescribir uno antiguo, emplee la función `FileCreate`.

Como los flujos de datos son intercambiables, se pueden emplear para mejorar la velocidad de aplicaciones que hagan un uso intensivo de ficheros, pues se puede trabajar con flujos de datos de memoria para guardarlos después en un archivo.

Veamos cómo podría implementarse un procedimiento para copiar un archivo desde un origen a un destino:

```
procedure CopiarFichero (Origen, Destino: String);
var
  Flujo1, Flujo2: TFileStream;
begin
  Flujo1:= TFileStream.Create (Origen, fmOpenRead);
  try
    Flujo2:= TFileStream.Create (Destino, fmCreate);
    try
      Flujo2.CopyFrom (Flujo1, Flujo1.Size);
    finally
      Flujo2.Free;
    end;
  finally
    Flujo1.Free;
  end;
end;
```

El código es muy sencillo, así que vamos a aprovecharlo para implementar una aplicación gráfica y perfectamente funcional que use flujos de datos de archivo para copiar cualquier fichero en su directorio original, con la posibilidad incluso de mantener o no la fecha original.

Abra una aplicación en Lazarus de nombre **copiarfichero** con una unidad de formulario llamada **copiarF**. Arrastre sobre el formulario un botón, dos etiquetas, una casilla de verificación y un cuadro de edición de ficheros (`TFileNameEdit`). La estructura del formulario, sus componentes y acciones, tal y como figura en el archivo `copiarF.lfm`, es la siguiente:

```
object Form1: TForm1
  Height = 240
  Width = 680
  Caption = 'Copiar ficheros'
  object lblCopiarDe: TLabel
    Left = 16
    Top = 14
    Caption = 'Original:'
  end
  object edCopiarDe: TFileNameEdit
    Width = 515
```

```
      OnAcceptFileName = edCopiarDeAcceptFileName
    end
    object lblCopiarA: TLabel
      Caption = 'Se copiará en:'
    end
    object cbFecha: TCheckBox
      Caption = 'Copiar la fecha original'
    end
    object btnCopiar: TButton
      Caption = 'Copiar Fichero'
      OnClick = btnCopiarClick
    end
  end
```

El formulario se parecerá al mostrado en la figura 13.5.

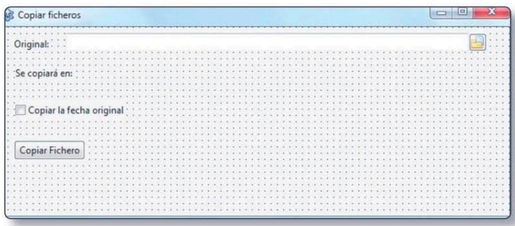



**Figura 13.5.** El formulario del ejemplo en tiempo de diseño

Borre la sección pública del formulario y añada los siguientes métodos en la sección privada de la declaración de `TForm1`:

```
private
    Origen: string;
    Destino: string;
    procedure CopiarFichero(aOrigen, aDestino: string;
                            CopiarFecha: boolean);
end;
```

Ahora generemos los manejadores para los eventos `OnClick` del botón y `OnAcceptFileName` del editor de nombres de fichero. Use la compleción automática de código para escribir el esqueleto del procedimiento `CopiarFichero` y complete los tres métodos como sigue:

```
procedure TForm1.btnCopiarClick(Sender: TObject);
var anexar: string;
begin
  CopiarFichero(Origen, Destino, cbFecha.Checked);
  btnCopiar.Enabled:= False;
  edCopiarDe.Text:= EmptyStr;
  case cbFecha.Checked of
    False: anexar:= ' copiado con la fecha actual';
    True: anexar:= ' copiado con la fecha original';
  end;
  lblCopiarA.Caption:= Destino + anexar;
end;

procedure TForm1.edCopiarDeAcceptFileName(Sender:
                    TObject; var Value: String);
var ruta, fNombre: string;
begin
  if (Value = EmptyStr) then Exit;
  ruta := ExtractFilePath(Value);
  fNombre:= ExtractFileName(Value);
  Destino:= ruta + 'Copia de ' + fNombre;
  case FileExistsUTF8(Destino) of
    False: begin
          lblCopiarA.Caption:= 'El fichero se copiará
                          en: ' + Destino;
          btnCopiar.Enabled:= True;
          Origen:= Value;
          end;
  True : case QuestionDlg('¡Cuidado!', Destino + ' ya
              existe' + sLineBreak+'¿Desea reescribir
                lo?', mtWarning, [mrYes,'Copiar', mrNo,
              'Cancelar'],0) of
             mrYes: begin
              lblCopiarA.Caption := 'El
                fichero se copiará en:  '  +
                Destino;
                btnCopiar.Enabled:= True;
                Origen:= Value;
                end;
            else begin
              Value:= EmptyStr;
              btnCopiar.Enabled:= False;
```

```
                Origen:= EmptyStr;
                Destino:= EmptyStr;
                    end;
            end;
    end;
end;

procedure TForm1.CopiarFichero(aOrigen, aDestino:
                string; CopiarFecha: boolean);
var src: TFileStream = nil;
    dest: TFileStream = nil;
begin
  if SameText(aOrigen, aDestino) then Exit;
  src := TFileStream.Create(UTF8ToSys(aOrigen),
                fmOpenRead);
  try
    dest := TFileStream.Create(UTF8ToSys(aDestino),
              fmOpenWrite or fmCreate);
    try
      dest.CopyFrom(src, src.Size);
      if CopiarFecha then
        FileSetDate(dest.Handle,FileGetDate(src.Handle));
    finally
      dest.Free;
    end
  finally
    src.Free;
  end;
end;
```

El procedimiento `CopiarFichero` usa `TFileStream.Create` y `TFileStream.CopyFrom` para realizar la copia del archivo. El procedimiento `OnClick` del botón llama al procedimiento `CopiarFichero`, desactiva el botón, borra el campo de edición del nombre del fichero e indica el resultado de la operación de copiado.

El evento `AcceptFileName` del campo de edición revisa si existe un archivo del mismo nombre que el que se intenta crear y actúa en consecuencia. Si el archivo no existe, se copia. En caso contrario, la función `QuestionDlg()` nos permite cancelar la operación o reescribir el archivo.

### 13.4.4 Compresión de flujos de datos con *ZStream*

Lazarus implementa en la unidad **ZStream** un descendiente de TStream, **TCompressionStream**, que usa un algoritmo de compresión para escribir datos comprimidos en el flujo de salida especificado. Así mismo, se puede leer y descomprimir datos de un flujo de entrada comprimido con el descendiente **TDecompressionStream.**

Como ejemplo del uso de estas clases, estudiaremos un pequeño programa llamado **ZSComp** que comprime y descomprime archivos con la unidad ZStream. El programa tiene dos cuadros de edición en los que se indica el nombre del archivo que va a comprimirse y el nombre del archivo resultante, que se crea en caso de que no exista previamente.

Cuando se hace clic sobre el botón **Comprimir**, el archivo original se utiliza para crear el archivo destino, de extensión .zsc. Al hacer clic sobre el botón **Descomprimir**, se lleva el archivo comprimido de vuelta a un flujo de datos de memoria. En ambos casos, el resultado de la compresión o descompresión se muestra en un campo de memo, por lo que es conveniente que use como archivo de origen un fichero de texto para visualizar todo correctamente.

Los controles y acciones definidos para la aplicación son los siguientes:

```
object Form1: TForm1
  Caption = 'Compresión de flujos con ZStream'
  object lblOrigen: TLabel
    Caption = 'Fichero Original:'
  end
  object lblDestino: TLabel
    Caption = 'Fichero Comprimido:'
  end
  object btnComprimir: TButton
    Caption = 'Comprimir'
    OnClick = btnComprimirClick
  end
  object btnDescomprimir: TButton
    Caption = 'Descomprimir'
    OnClick = btnDescomprimirClick
  end
  object Memo1: TMemo
    Lines.Strings = (
      'Salida del fichero comprimido'
    )
```

```
  end
  object edComp: TEdit
  end
  object edOrigen: TFileNameEdit
    OnAcceptFileName = edOrigenAcceptFileName
  end
end
```

Para poder reutilizar mejor el código de esta pequeña aplicación pueden escribirse dos funciones independientes: una para comprimir y otra para descomprimir un flujo de datos en otro. Este es el código:

```
function ComprimirFlujo (aOrigen, aDestino: TStream):
                         Single;
var
  compFlujo: TCompressionStream;
begin
  compFlujo := TCompressionStream.Create(clFastest,
                         aDestino);
  try
    compFlujo.CopyFrom(aOrigen, aOrigen.Size);
    // Porcentaje de compresión
    Result := compFlujo.get_CompressionRate;
  finally
    compFlujo.Free;
  end;
end;

procedure DescomprimirFlujo (aOrigen, aDestino: TS-
tream);
var
  decompFlujo: TDecompressionStream;
  Lectura: Integer;
  Buffer: array [0..1023] of Char;
begin
  decompFlujo := TDecompressionStream.Create(aOrigen);
  try
     {aFlujoDest.CopyFrom (decompFlujo, tamano) no
        funciona porque no se sabe su tamaño}
    repeat
      Lectura := decompFlujo.Read(Buffer, 1024);
      aDestino.Write (Buffer, Lectura);
    until Lectura = 0;
```

```
  finally
    decompFlujo.Free;
  end;
end;
```

Como se puede ver en los comentarios del código, la operación de descompresión resulta más compleja, ya que no se puede utilizar el método `CopyFrom` al desconocer el tamaño del flujo resultante.

Ahora generemos los manejadores para los eventos `OnClick` de los dos botones y `OnAcceptFileName` del cuadro de edición `edOrigen`. Complete los tres métodos como sigue:

```
procedure TForm1.btnComprimirClick(Sender: TObject);
var
  aOrigenFlujo, aDestinoFlujo: TFileStream;
begin
  aOrigenFlujo := TFileStream.Create (edOrigen.Text,
                       fmOpenRead);
  aDestinoFlujo := TFileStream.Create(edCompressed.
                      Text, fmCreate);
  try
    ComprimirFlujo (aOrigenFlujo, aDestinoFlujo);
    aDestinoFlujo.Position:= 0;
    Memo1.Lines.LoadFromStream (aDestinoFlujo);
  finally
    aOrigenFlujo.Free;
    aDestinoFlujo.Free;
  end;
end;

procedure TForm1.btnDescomprimirClick(Sender: TObject);
var
  aFlujoOrigen: TFileStream;
  aFlujoDestino: TMemoryStream;
begin
  aFlujoOrigen := TFileStream.Create (edCompressed.
                      Text, fmOpenRead);
  aFlujoDestino := TMemoryStream.Create;
  try
    DescomprimirFlujo(aFlujoOrigen, aFlujoDestino);
    aFlujoDestino.Position:= 0;
```

```
      Memo1.Lines.LoadFromStream(aFlujoDestino);
    finally
      aFlujoOrigen.Free;
      aFlujoDestino.Free;
    end;
  end;

  procedure TForm1.edOrigenAcceptFileName(Sender: TObject;
                            var Value: String);
  begin
   edOrigen.Text:= ExtractFilePath(Value) + ExtractFileName
                                                  (Value);
   edCompressed.Text := edOrigen.Text + '.zsc';
  end;
```

Cuando compile y ejecute la aplicación, elija preferiblemente un archivo de texto en el campo **Fichero original** y haga clic sobre el botón **Comprimir**. El componente **Memo** visualizará el resultado de la compresión del flujo de datos del archivo original.

La figura 13.6 muestra el resultado para el archivo comprimido del código fuente del formulario del programa actual (ZSCompF.pas).

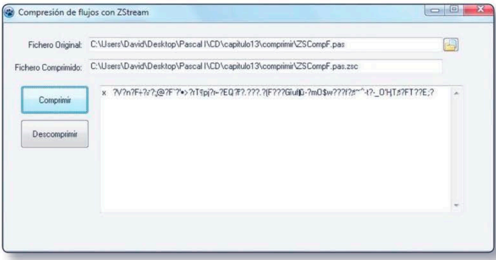



**Figura 13.6.** El ejemplo ZSComp puede comprimir un archivo con la unidad ZStream

Si pulsa el botón **Descomprimir**, se procederá a descomprimir el fichero ZSCompF.pas.zsc y copiarlo a un flujo de datos de memoria para mostrar el contenido en el campo de memo.

### 13.4.5 Criptografía de un flujo de datos con *Blowfish*

Para concluir este capítulo dedicado a las clases no visuales, hemos escrito una pequeña aplicación que permite cifrar un flujo de datos con el algoritmo de dominio público **Blowfish**. Para ello, vamos a emplear una instancia de la clase **TBlowFishEncryptStream** y otra de **TBlowFishDeCryptStream** para cifrar y descifrar, respectivamente, el flujo de datos.

El programa tiene dos campos de memo, un cuadro de edición y dos botones como controles fundamentales (Figura 13.7).

Cuando se hace clic sobre el botón **Cifrar**, la aplicación cifra el contenido del cuadro de texto con la clave elegida y le muestra el criptograma en el segundo campo de memo. Cuando se pulse el botón **Descifrar**, se lleva la cadena cifrada de vuelta a un flujo de datos que se descifra con la misma clave y muestra el texto plano en el primer componente de memo. Los controles y acciones definidos para la aplicación son los siguientes:

```
object Form1: TForm1
  Caption = 'Cifrador de bloques Blowfish'
  object edClave: TEdit
    TabOrder = 0
  end
  object btnCifrar: TButton
    Caption = 'Cifrar'
    OnClick = btnCifrarClick
    TabOrder = 2
  end
  object btnDescifrar: TButton
    Caption = 'Descifrar'
    OnClick = btnDescifrarClick
    TabOrder = 3
  end
  object mClaro: TMemo
    TabOrder = 1
  end
  object mCifrado: TMemo
    TabOrder = 4
  end
end
```

La figura 13.7 muestra el resultado de cifrar el contenido del campo **Texto claro** con la clave **Lazarus**:

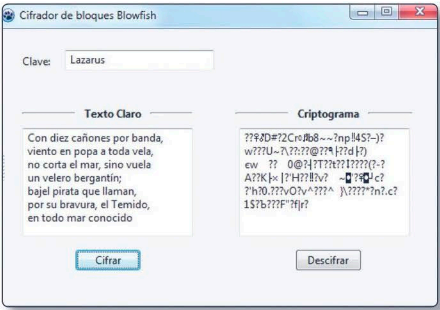



**Figura 13.7.** El ejemplo Cifrador puede comprimir flujos de datos con Blowfish

Una vez diseñado el formulario, añada la unidad **Blowfish** en la cláusula **uses** y tres campos en la sección privada del formulario:

```
private
    sOriginal, sCifrado: TStringStream;
    clave: string;
end;
```

Hemos declarado dos instancias de `TStringStream` y una cadena, que será la clave con la que el algoritmo **Blowfish** cifre y descifre el contenido del flujo de datos.

Ahora generemos los manejadores para los eventos `OnClick` de los botones `btnCifrar` y `btnDescifrar`:

```
procedure TForm1.btnCifrarClick(Sender: TObject);
var bfCif: TBlowFishEncryptStream;
begin
  if mClaro.Lines.Count = 0 then Exit;
  clave:= edClave.Text;
  // Creamos una instancia de TStringStream vacía
  sOriginal:= TStringStream.Create(EmptyStr);
  {Iniciamos el cifrado al vuelo escribiendo el
```

```
      texto claro en el flujo}
    bfCif:= TBlowFishEncryptStream.Create(clave, sOriginal);
    bfCif.WriteAnsiString(mClaro.Text);
    {Liberamos recursos y  mostramos el criptograma del
      flujo en el campo de memo}
    bfCif.Free;
    mCifrado.Text:= sOriginal.DataString;
    mClaro.Text:= '';
  end;

  procedure TForm1.btnDescifrarClick(Sender: TObject);
  var bfDes: TBlowFishDeCryptStream;
  begin
    // Creamos instancia con el contenido cifrado
    sCifrado:= TStringStream.Create(sOriginal.DataString);
    // y desciframos el flujo de datos
    bfDes:= TBlowFishDeCryptStream.Create(clave, sCifrado);
    // para mostrar su contenido en el memo
    mClaro.Text:= bfDes.ReadAnsiString;
    mCifrado.Text:='';
    // y liberamos toda la memoria
    bfDes.Free;
    sCifrado.Free;
    sOriginal.Free;
  end;
```

Si todo ha ido bien, debe ser capaz de compilar y ejecutar el programa para explorar sus características. El campo de `memo` que recibe el texto cifrado debe tomarse con cautela. Aunque todos los bytes se cifran y almacenan correctamente en la propiedad `Lines` del control, no se muestran correctamente, puesto que muchos de ellos no representan caracteres imprimibles.

# 14

# MANIPULACIÓN DE FICHEROS

Cuando el volumen de información que maneja una determinada aplicación aumenta considerablemente, es necesario emplear unas estructuras de datos especiales denominadas **archivos** o **ficheros**, que, al menos teóricamente, pueden almacenar una cantidad ilimitada de datos.

Son muchas las situaciones en las que es necesario manipular información que supera ampliamente la capacidad de almacenamiento de la memoria principal de la máquina, como la gestión de los datos fiscales de toda la población de un país, por ejemplo. En estos casos, es necesario utilizar estructuras de datos que puedan manipularse en soportes físicos no volátiles, como discos duros, discos ópticos, tarjetas de memoria, etc.

A lo largo de este capítulo aprenderá a gestionar y manipular con Lazarus ficheros de texto y binarios, tanto de la forma procedural clásica como con el estilo orientado a objetos propio de los flujos de datos ya estudiados en el capítulo anterior.

## 14.1 REGISTROS Y CAMPOS DE UN ARCHIVO

Un **fichero** puede definirse como una colección de **datos homogéneos** almacenados en un soporte físico, es decir que almacenan grupos de datos del mismo tipo, como en el caso de los arreglos. Cada uno de los elementos almacenados en el fichero se llama **registro**. Cada registro, a su vez, puede estar formado por un conjunto de **campos** que sí pueden ser **heterogéneos** (Figura 14.1).

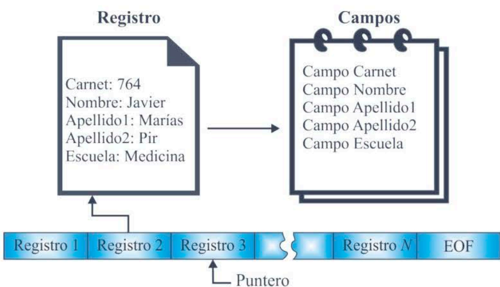



**Figura 14.1.** Relación entre registros y campos en un archivo

En cada operación de lectoescritura sobre un fichero solo podrá leerse o escribirse un único registro. La operación de acceso a esa información se lleva a cabo de dos formas distintas:

- **Acceso secuencial**. El acceso a un registro $j$ de un archivo supone la exploración previa y secuencial de los $j - 1$ registros anteriores. Así pues, el tiempo de acceso a un determinado dato vendrá definido por la posición del mismo dentro de la secuencia.

- **Acceso aleatorio**. El acceso al registro deseado se realiza directamente mediante la posición que ocupa en la secuencia, del mismo modo que se accedía a un dato almacenado en un arreglo mediante un índice. Aquí el tiempo medio de acceso a un registro es constante.

Cuando se realiza una operación de lectura/escritura sobre un fichero, existe un identificador de la posición del archivo sobre la que se está trabajando denominado **puntero de lectura/escritura** (puntero L/E).

Todo archivo dispone de un registro especial que indica su final. Este carácter, denominado **Fin de fichero** o **EOF** (*End Of File*) se incluye automáticamente por el sistema (Figura 14.2).

**Figura 14.2.** Registro especial fin de fichero y puntero L/E

## 14.2 OPERACIONES SOBRE FICHEROS

Las operaciones más habituales que Lazarus puede efectuar sobre los archivos se pueden clasificar de la siguiente manera:

- **Creación**. Esta operación permite relacionar un nombre lógico con el fichero físico. El nombre lógico será utilizado por el programa para acceder a la información física almacenada en memoria.
- **Apertura**. Una vez asignado el fichero, debe abrirse este para permitir el acceso a la información que tiene guardada.
- **Lectura/Escritura**. Son todas aquellas operaciones que permiten leer o escribir datos en el fichero.
- **Inserción/Borrado**. Permiten modificar los valores de alguno de los registros almacenados en el archivo, o borrarlos definitivamente.
- **Renombrado/Eliminación**. Permiten cambiar el nombre físico del fichero o eliminarlo completamente.
- **Desplazamiento**. Son aquellas operaciones que permiten mover el puntero L/E a lo largo de los diferentes registros que forman el archivo. En el caso de los ficheros de acceso secuencial no existen instrucciones que permitan saltar al registro deseado. En los ficheros de acceso aleatorio sí será posible situar el puntero en el registro buscado.
- **Cierre**. Es la operación que permite cerrar el archivo cuando se ha terminado de trabajar con él. Es esencial que todo archivo abierto se cierre antes de que finalice el programa, pues de otro modo el contenido del fichero podría quedar inutilizable.

En Pascal existe un conjunto de operaciones predefinidas que permiten trabajar de forma sencilla sobre la información contenida en los ficheros.

**IMPORTANTE**
Siempre que trabaje con un archivo deberá realizar las operaciones de creación, apertura, operaciones sobre el fichero (lectura/escritura, borrado/inserción, etc.) y cierre del mismo antes de la finalización del programa que maneja el fichero.

## 14.3 TIPOS DE FICHEROS

Existen distintas formas de clasificar los posibles tipos de archivos. En general pueden catalogarse en función de:

- **Tipo de acceso a la información**. Los archivos se clasifican, atendiendo a este parámetro, como hemos visto al principio, como de *acceso secuencial* o *aleatorio*.
- **Organización de los datos en memoria**. Los ficheros pueden catalogarse como de organización *secuencial*, si los registros se encuentran almacenados de forma consecutiva en memoria; de organización *aleatoria*, cuando el orden de los registros no coincide con el orden lógico en el que se han almacenado en memoria y, finalmente, de organización *indexada*, en el que deben utilizarse dos ficheros.
- **Tipo de información almacenada**. Según la información que almacenan, los ficheros pueden ser *de texto* y *binarios*. Los archivos *binarios* almacenan secuencias de dígitos binarios, mientras que los *de texto* guardan caracteres alfanuméricos en cualquier sistema de codificación y pueden leerse y/o modificarse mediante editores de texto.

## 14.4 EL TIPO FILE

Lazarus permite trabajar con dos tipos diferentes de ficheros, en función del tipo de datos que almacenan:

- **Archivos de texto**. Los ficheros de texto en Pascal almacenan caracteres alfanuméricos en un formato de codificación adecuado.
- **Archivos con tipo**. Los archivos con tipo en Pascal son de tipo binario y almacenan elementos de un tipo predefinido:
  - **Simple**. Cualquiera de los tipos simples estudiados: `Integer`, `Boolean`, `Char`, `Single`, etc.; o los definidos por el usuario, subrangos o enumerados.
  - **Estructurado**. Pueden emplearse los tipos `array`, `record`, `string` o `set`; pero no es posible utilizar el tipo `file`.
  - **Puntero**. Lazarus permite construir ficheros que almacenen datos de tipo puntero.

Dado que hay dos tipos principales de archivos en Lazarus (de texto y con tipo), existen dos formas distintas de declarar este tipo de estructuras, como vimos en el Capítulo 4:

```
type
  TFichero = File Of TipoDeDatos; // binario
  TFicheroDeTexto = Text;  // de texto

var
  NombreFichero1 : File Of TipoDatos;
  NombreFichero2 : Text;
```

El nombre es un identificador válido en Pascal que se utiliza como referencia lógica y permite acceder a la información almacenada en el fichero físico. Recuerde que puede crear archivos de cualquier tipo definible en Pascal, excepto de tipo fichero.

### 14.4.1 Asignación de un fichero

Para leer o escribir en un archivo es necesario declarar una variable conocida como **archivo lógico**, al que se le debe asignar su correspondiente **archivo físico**.

El proceso de asignación entre el fichero físico y el lógico se realiza en Lazarus con la palabra reservada **`AssignFile`**:

```
AssignFile (FicheroLogico, FicheroFisico);
```

Donde `FicheroLogico` es una variable de tipo archivo y `FicheroFisico` una cadena que representa el nombre del archivo con su ruta completa; por ejemplo: `'C:\Pascal\Proyectos\ejemplo.dat'`.

En caso de no especificarse ruta, el compilador buscará el fichero en la misma carpeta donde se encuentra el archivo ejecutable del programa.

Cuando realice la asignación de un archivo compruebe que el tipo de la variable lógica coincide con el tipo de datos almacenados en el fichero y que, para realizar operaciones de lectura o escritura de datos, el fichero debe existir previamente.

## 14.5 ARCHIVOS DE TEXTO

Un **archivo de texto** es una estructura de datos que permite guardar caracteres alfanuméricos en una codificación estándar en memoria, principal o secundaria. Es posible declarar archivos de texto de dos maneras distintas:

```
type
  TFicheroTexto1= File Of Char; // caracteres binarios
var
  NomFich1 : TFicheroTexto1;
  NomFich2 : TextFile;    // de texto
```

La primera declaración crea un **archivo binario** que almacena caracteres alfanuméricos byte a byte. La segunda utiliza la palabra reservada **`TextFile`**, que implementa directamente la estructura del fichero de texto. Ambas declaraciones crean archivos casi idénticos, pero existe una diferencia básica. Un archivo de texto creado como `TextFile` se divide en **líneas** y cada una termina en un carácter especial denominado **fin de línea** (*End Of LiNe* [**EOLN**]), por lo que los datos se verán tal cual se escribieron. Son mucho más funcionales que los ficheros de carácter en cuanto al uso de las distintas operaciones, por lo que se recomienda su uso casi en exclusiva.

### 14.5.1 Apertura de archivos de texto

Después de haber asignado el fichero con la instrucción `AssignFile` es necesario abrirlo para poder trabajar con los datos almacenados. Para ello, el lenguaje nos proporciona tres métodos predefinidos según el tipo de operación que se desee realizar sobre el archivo:

- **`Reset`**. Abre el archivo físico que corresponde al nombre lógico que previamente se le ha asignado y sitúa el **puntero** L/E en la **primera posición** del archivo en modo **lectura.**
- **`ReWrite`**. Abre el fichero en modo de **lectura** y **escritura** y sitúa el **puntero** L/E en la **posición inicial** del mismo. Si el archivo físico no existe, crea uno nuevo vacío. Si existe, se sobrescriben todos los datos existentes; es decir, si se utiliza esta operación sobre un archivo que contiene datos, estos se borrarán.

- **Append**. Cuando se dispone de un archivo de texto y se desean añadir nuevos datos sin borrar los anteriores, debe utilizarse esta operación. El **puntero** L/E se sitúa al **final** del mismo y el archivo se abre en **modo escritura**, por lo que se emplea para añadir datos al final del archivo de texto.

## 14.5.2 Lectura/Escritura de datos en archivos de texto

Las dos operaciones básicas que se pueden realizar sobre un fichero son la **lectura** y la **escritura**. Para ello, Lazarus permite utilizar los procedimientos `Read/ReadLn` y `Write/WriteLn`, respectivamente.

Para **leer** los datos del fichero es necesario haberlo abierto antes mediante una operación `Reset`. Existen dos procedimientos para la operación de lectura:

```
Read ( var NombreFichero : TextFile; var Valor {,var
      Valor2 ...};);
ReadLn ( var NombreFichero : TextFile; var Valor1 {,var
        Valor2 ...};);
```

Donde `NombreFichero` representa el identificador de tipo archivo que previamente se ha asignado al archivo físico. A continuación aparecen una o varias variables de tipo carácter o cadena.

Ambos procedimientos tienen un comportamiento diferente. Si se emplea `Read`, se lee desde la posición actual del puntero L/E un carácter y se mueve el puntero a la siguiente posición del archivo. Si se emplea `ReadLn`, se lee el conjunto de caracteres desde la posición que ocupa el puntero hasta que se encuentra el carácter **EOLN** o fin de línea y se almacena en la variable de salida. Si es de tipo **`String`** se devuelve la cadena; si es de tipo `Char` se devuelve el primer carácter leído y se obvia el resto.

Para poder **escribir** datos en un archivo de texto es necesario haberlo abierto antes con los procedimientos `ReWrite` o `Append`. Una vez abierto, pueden escribirse los datos empleando cualquiera de estos dos procedimientos:

```
Write (var NombreFichero : TextFile; Expresion1
      {opciones} {, Expresion2 {opciones} ...});
WriteLn (var NombreFichero : TextFile; Expresion1
        {opciones} {, Expresion2 {opciones} ...});
```

Donde `NombreFichero` representa el identificador de tipo archivo que previamente se ha asignado al archivo físico. A continuación aparecen una o varias expresiones que deben ser del mismo tipo que el archivo declarado, de carácter o cadena.

Al igual que sucedía con los procedimientos de lectura, los procedimientos de escritura tienen un comportamiento ligeramente diferente. El procedimiento `Write` escribe los parámetros que se le pasan en el archivo y deja el puntero después del último elemento escrito. Sin embargo, cuando se emplea `WriteLn` introduce un carácter **EOLN** después del último elemento introducido, por lo que el puntero de lectura/escritura se sitúa al comienzo de la línea siguiente.

### 14.5.3 Cierre de los ficheros de texto

Una vez acabadas las distintas operaciones efectuadas sobre un archivo siempre es necesario cerrarlo con el procedimiento predefinido **`CloseFile`** de la manera adecuada, pues, en caso contrario, se perderán todos los datos. La operación de cierre de un archivo sitúa el carácter **EOF** en el lugar donde se encuentra el puntero L/E. Si esta operación no se efectúa, el fichero quedará **corrupto** y, al no poder identificarse su final, los datos no se podrán visualizar.

En los siguientes ejemplos vamos a trabajar de forma práctica y completa con los procedimientos de creación, escritura y lectura de un archivo de texto. Abra un primer proyecto de nombre **EscrituraFichero** en modo consola. Añada la unidad `SysUtils` en la cláusula **uses** y complete su esqueleto para que el programa se parezca a esto:

```
program EscrituraFichero;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;

var
 Fichero: TextFile;
```

```
begin
  WriteLn ('Prueba de escritura', sLineBreak);
  AssignFile(Fichero, 'prueba.txt');
  {$I+} // usa excepciones
  try
    if FileExists('prueba.txt') then
      Append(Fichero)    // se abre para edición
    else
      ReWrite(Fichero);  // se crea el fichero

    Writeln(Fichero, 'Mi primer archivo de texto');
    CloseFile(Fichero); // y se cierra
  except
    on E: EInOutError do
    begin
     Writeln('Ha ocurrido un error. Detalles:
          '+E.ClassName+'/'+E.Message, sLineBreak);
    end;
  end;
  WriteLn('Ejemplo acabado. Pulse Enter para
                   finalizar.');
  {$IFDEF WINDOWS}
    ReadLn;
  {$ENDIF}
end.
```

Cuando compile y ejecute el programa obtendrá por pantalla la siguiente salida:

```
Prueba de escritura
Ejemplo acabado. Pulse Enter para finalizar.
```

Parece que no ha pasado nada, pero si observa el directorio donde ha compilado el programa, verá que se ha creado un fichero de texto de nombre `prueba.txt`. Si lo abre con el mismo bloc de notas del sistema operativo, podrá leer la cadena de texto que el programa ha escrito en aquel (Figura 14.3 A). ¿Qué ocurriría si por cualquier motivo el fichero no se ha podido cerrar de forma adecuada? Simplemente, que los datos se perderían. Repita el ejercicio eliminando el procedimiento `CloseFile(Fichero)` y observe lo que ocurre cuando abre el archivo `prueba.txt` con el bloc de notas (Figura 14.3 B).

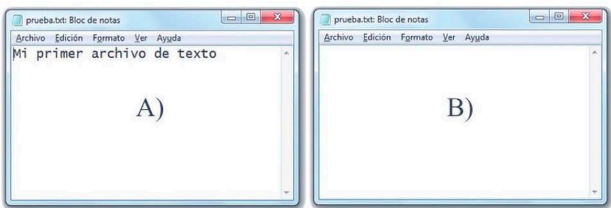



**Figura 14.3.** Prueba de escritura en un fichero. A) Cierre adecuado. B) El archivo queda abierto

Efectivamente, el archivo de prueba está vacío, se ha perdido toda la información como consecuencia de un cierre inadecuado tras efectuar cualquier operación sobre el fichero.

Estudiemos con un poco más de detalle el código fuente. Tras asignar el fichero lógico de nombre `Fichero` a su archivo físico `prueba.txt`, activamos la directiva de compilación **`{$I+}`** para verificar los errores de E/S, como veremos después.

Al comprobar la existencia del archivo `prueba.txt` con la llamada a la función `FileExists` dentro del bloque **`try`**/**`except`**, obligamos al compilador a comprobar en todo momento la excepción `EInOutError`. Si el fichero existe previamente, se abre para edición con el procedimiento `Append`; en caso contrario, se crea uno vacío con el procedimiento `ReWrite` y se escribe la línea `Mi primer archivo de texto` antes de cerrar el archivo de forma segura con el procedimiento `CloseFile`. Si aparece una excepción como consecuencia de un **error** de **Entrada/Salida**, se llamará a la función `Writeln`, con la que hemos construido un mensaje con los detalles del problema (Figura 14.4). El compilador se encargará, así mismo, de gestionar la excepción y liberarla, por lo que el programa podría seguir con la ejecución sin ningún problema.

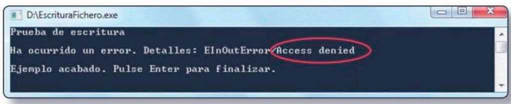



**Figura 14.4.** Error de Entrada/Salida por Acceso Denegado al fichero

Veamos ahora cómo preparar un archivo de texto para añadir nueva información a la ya existente. Abra un proyecto de nombre **EdicionFichero** en modo consola. Añada de nuevo la unidad `SysUtils` en la cláusula **uses** y complete su esqueleto del siguiente modo:

```
program EdicionFichero;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;

var
 Fichero: TextFile;

begin
  WriteLn ('Anexando una nueva linea...', sLineBreak);
  AssignFile (Fichero, 'prueba.txt');
  {$I+}
  try
    if FileExists('prueba.txt') then
      Append (Fichero)
    else
      Rewrite(Fichero);

    Writeln(Fichero, 'con una nueva línea...');
    CloseFile(Fichero);
  except
    on E: EInOutError do
    begin
     Writeln('Ha ocurrido un error. Detalles: ' +
          E.ClassName+'/'+E.Message, sLineBreak);
    end;
  end;
  WriteLn ('Ejemplo acabado. Pulse Enter para finali
                           zar.');
   {$IFDEF WINDOWS}
    ReadLn;
  {$ENDIF}
end.
```

Al ejecutar el programa obtendrá por pantalla la siguiente salida:

```
Anexando una nueva linea…
Ejemplo acabado. Pulse Enter para finalizar.
```

Efectivamente, el programa ha agregado una nueva línea al archivo `prueba.txt`. Si lo abre con el Bloc de notas podrá ver que ahora el fichero tiene dos líneas (Figura 14.5).

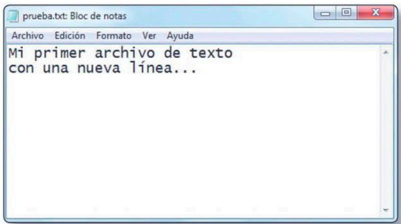



**Figura 14.5.** El fichero de texto con una nueva línea

De nuevo, comprobamos si existe o no el fichero tras la asignación para abrirlo con el método correspondiente. Tras insertar una nueva línea, cerramos el archivo con el método `CloseFile`.

Como en el ejemplo anterior, controlamos los errores de Entrada/Salida con la directiva de compilación ***{$I+}***.

Por último, vamos a emplear el procedimiento `Reset` para abrir un fichero en modo lectura y mostrar su contenido por pantalla. Abra un proyecto en modo consola con el nombre **`LecturaFichero`** y ajuste su estructura para que quede del modo siguiente:

```
program LecturaFichero;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;
```

```
var
 Fichero: TextFile;
 Cadena: String;

begin
  Writeln('Lectura de un fichero:', sLineBreak);
  AssignFile(Fichero, 'prueba.txt');
  {$I+}
  try
    Reset(Fichero); // Abre en modo lectura
    Repeat  // Repite hasta fin fichero
      Readln(Fichero, Cadena); // Lee toda la línea
      Writeln(Cadena); // Escribe la línea leída
    until(EOF(Fichero));
    CloseFile(Fichero);
  except
    on E: EInOutError do
    begin
     Writeln('Ha ocurrido un error. Detalles: ' +
          E.ClassName+'/'+E.Message, sLineBreak);
    end;
  end;
  WriteLn (sLineBreak, 'Ejemplo acabado. Pulse Enter
       para finalizar.');
   {$IFDEF WINDOWS}
    ReadLn;
  {$ENDIF}
end.
```

## 14.6 ARCHIVOS BINARIOS

Además de con ficheros de texto, Lazarus puede trabajar con archivos binarios. Un **archivo binario** es un fichero informático que contiene información de cualquier tipo simple, estructurado o puntero y codificada en binario con el propósito de almacenamiento y procesamiento en los ordenadores. Estos archivos con tipo son ficheros de acceso aleatorio, por lo que solo pueden asociarse con dispositivos de almacenamiento que permitan este tipo de acceso. Los archivos con tipo son como los arreglos, y se declaran con la palabra reservada **File** seguida del tipo de dato simple (menos File) que definirá cada bloque o elemento que tendrá el archivo:

```
var
 NombreFichero : File of TipoBase;
```

Para poder trabajar con ficheros binarios, como ocurre con los de texto, debe realizarse el mismo conjunto de operaciones: declaración, asignación, apertura, lectura/escritura y cierre.

Como ocurre con los archivos de texto, en Windows los nombres de los ficheros con tipo solo deben estar en ASCII o en una de sus variantes ANSI.

### 14.6.1 Apertura de archivos binarios

Después de haber asignado un nombre lógico al fichero físico con la instrucción `AssignFile`, se necesita abrir el fichero binario para poder trabajar con los datos almacenados. Para ello, Lazarus nos proporciona solo dos métodos predefinidos según el tipo de operación que se desee realizar sobre el archivo:

- **`Reset`**. Esta operación abre el archivo físico y sitúa el puntero L/E en la **primera** posición del archivo en modo Lectura/Escritura según el valor asignado previamente a la variable **`FileMode`**. El modo por defecto es Lectura/Escritura. Si se intenta abrir un fichero que no existe, se obtendrá un error.
- **`ReWrite`**. El fichero en modo de lectura y escritura. Si el archivo físico no existe, crea uno nuevo vacío. Si existe, se sobrescriben todos los datos existentes, es decir, si se utiliza esta operación sobre un archivo que contiene datos, estos se borrarán.

El procedimiento `Append` no se contempla con los archivos binarios, pues no es necesario, ya que con el uso de los procedimientos `Reset` y `Seek` se pueden ir añadiendo más bloques o elementos al final de un archivo con tipo.

### 14.6.2 Lectura/Escritura de datos en archivos binarios

Las dos operaciones básicas que pueden realizarse sobre estos ficheros son la **lectura** y la **escritura**. Para ello, Lazarus permite utilizar los procedimientos `Read` y `Write`, respectivamente.

Para **leer** los datos del fichero binario se utiliza el mismo procedimiento predefinido **`Read`** que se empleaba con los ficheros de texto, y con idéntica sintaxis.

Para poder **escribir** datos en un archivo con tipo se emplea el procedimiento **`Write`,** con idéntica estructura que en el caso de los archivos de texto.

En estos archivos con tipo no podrán usarse los procedimientos `ReadLn`/`WriteLn`, pues el concepto de línea no tiene sentido con ficheros binarios, que no almacenan texto.

### 14.6.3 Cierre de los ficheros

Una vez acabadas las distintas operaciones efectuadas sobre un archivo siempre es necesario cerrarlo con el procedimiento predefinido **CloseFile** de la manera adecuada, pues, en caso contrario, podrían perderse todos los datos.

Como hicimos con los ficheros de texto, vamos a mostrar ahora un par de ejemplos para trabajar de forma práctica en la creación, escritura y lectura de un fichero binario. Abra un primer proyecto de nombre **EscrituraAgenda** en modo consola, añada la unidad `SysUtils` en la cláusula **uses** y defina un tipo registro de la siguiente forma:

```
type
  TRegistro = record
  Nombre: String[32];
  Telefono: String[9];
  end;
```

Ahora declare las variables `Registro` y `Archivo` así:

```
var
  Registro: TRegistro;
  Archivo: File of TRegistro;
```

En el cuerpo del programa asigne el nombre `Agenda` al fichero físico agenda.dat y abra el archivo para escribir un primer registro de la siguiente forma, sin olvidar la directiva **{*$I+*}**:

```
try
   ReWrite(Archivo); // Abre Archivo posición inicial
   Registro.Nombre:= 'Jesus de la Rua ';
   Registro.Telefono:= '634876312';
   Write(Archivo,Registro); // y escribe el registro
   CloseFile(Archivo);
except
  on E: EInOutError do
   begin
        Writeln('Ha ocurrido un error. Detalles: '+
             E.ClassName+'/'+E.Message, sLineBreak);
   end;
end;
```

Cuando compile y ejecute el programa `EscrituraAgenda.lpr` del ejemplo, obtendrá por pantalla la siguiente salida:

```
Escritura de un registro:
Ejemplo acabado. Pulse Enter para finalizar.
```

Si no se ha producido ningún error, se habrá creado en el mismo directorio un archivo de nombre `agenda.dat` que solo contendrá el registro grabado. Para comprobarlo, crearemos ahora un programa de nombre **`LecturaAgenda`** para leer su contenido.

Ajuste el cuerpo del programa para que se parezca a esto:

```
AssignFile(Archivo,'agenda.dat');
  try
   FileMode:= 0;       // Abrir solo lectura
   Reset(Archivo);
     Repeat
      Read(Archivo, Registro); // Lee contenido
      With Registro do  // Escribe en pantalla
        Writeln(Nombre,' ',Telefono );
     until(EOF(Archivo));
    CloseFile(Archivo);
  except
    on E: EInOutError do
    begin
     Writeln('Ha ocurrido un error. Detalles: ' +
           E.ClassName+'/'+E.Message, sLineBreak);
    end;
  end;
```

El código es sencillo. Para asegurarnos de que no se produzca la pérdida de ningún dato, abrimos el archivo con el procedimiento `Reset` en modo lectura, lo que se hace con la variable `FileMode := 0`. A continuación, y hasta que se llegue al final del fichero (**EOF**), leemos cada registro almacenado y lo mostramos por pantalla. Cuando finalice, se cierra el fichero. Como siempre, englobamos el proceso en un bloque **`try/except`**, para obligar al compilador a comprobar en todo momento la excepción `EInOutError`.

## 14.6.4 Otras operaciones con archivos binarios

Existe todo un conjunto de operaciones predefinidas en Free Pascal que puede facilitar la manipulación procedimental de los ficheros con tipo.

### 14.6.4.1 OPERACIONES DE ACCESO DIRECTO

Estas operaciones permiten acceder de forma aleatoria a cualquier elemento de un archivo en función de la posición que ocupe en el mismo, cosa que no puede realizarse en un fichero de texto:

- **`Seek`**. Este procedimiento sitúa el puntero L/E del archivo en la posición especificada como argumento. La variable de posición debe ser de tipo `LongInt`. Tenga en cuenta que la primera posición de un registro en un archivo binario es siempre 0.

```
Seek ( var Fichero: File; Posicion: LongInt );
```

- **`FileSize`**. Devuelve un valor entero que se corresponde con el número de registros que posee un fichero.

```
FileSize ( var Fichero: File );
```

- **`FilePos`**. Esta función devuelve un valor entero con la posición que ocupa actualmente el puntero L/E en el archivo que se le pasa por parámetro.

```
FilePos ( var Fichero: File );
```

Aprovechando estas operaciones de acceso aleatorio, vamos a programar una aplicación que nos permita editar nuestra agenda. Abra un nuevo programa en modo consola con el nombre **`EditarAgenda`** y agregue en el bloque **`try`** el código necesario para actualizarla con un nuevo registro:

```
Reset(Archivo);
Seek(Archivo, FileSize (Archivo));
Registro.Nombre:= 'David Lopez';
Registro.Telefono:= '732201464';
Write(Archivo,Registro);
CloseFile(Archivo);
```

El código es realmente sencillo, pero es necesario comentar alguna peculiaridad. El procedimiento **`Reset`**, por defecto, abre un fichero con tipo en modo lectura/escritura y sitúa el puntero L/E en la primera posición del archivo. Si se intenta escribir algún registro en esa situación lo que se consigue es borrar el registro inicial, así que antes de escribir nueva información hay que colocar el puntero en la última posición. Esto es precisamente lo que hemos hecho con la orden:

```
Seek (Archivo, FileSize (Archivo));
```

Después ya podemos escribir el nuevo registro y cerrar el fichero. Si ahora lee el archivo `agenda.dat` con la aplicación **`LecturaAgenda`**, observará cómo hay, efectivamente, dos registros (Figura 14.6).

```
C:\Users\David\Desktop\Pascal I\CD\capitulo14\FicheroBinario\LecturaAgend...
Lectura de agenda.dat:

Jesus de la Rua  634876312
David Lopez 732201464

Ejemplo acabado. Pulse Enter para finalizar.
```

**Figura 14.6.** Contenido del fichero agenda.dat

### 14.6.4.2 OPERACIONES DE GESTIÓN

Estas operaciones permiten realizar distintas acciones sobre los archivos físicos binarios almacenados en una memoria secundaria. Los procedimientos predefinidos en Free Pascal son los siguientes:

- **`Erase`**. Este procedimiento borra el archivo físico asociado al archivo lógico. Equivale al comando **`rm`** utilizado en las distribuciones Linux.

```
Erase ( Fichero: File );
```

- **`Rename`**. Permite modificar desde el programa el nombre físico del fichero siempre que el archivo se encuentre previamente **cerrado**. Es equivalente al comando **`mv`** de Linux.

```
Rename ( var Fichero: File; NombreNuevo: String );
```

- **`Truncate`**. Tras la apertura de un fichero binario, este procedimiento coloca el carácter EOF en la posición actual del puntero L/E, por lo que todos los registros situados tras esta posición se perderán.

```
Truncate ( var Fichero: File );
```

Le dejamos como ejercicio un procedimiento que trunque el archivo `agenda.dat` tras el primer registro, muestre en pantalla el nuevo contenido y, por último, elimine automáticamente el fichero.

En la tabla siguiente le mostramos un resumen con las operaciones que la programación procedural clásica soporta para la gestión de ficheros.

| Operación | Archivos de texto | Archivos binarios |
|---|---|---|
| `Append` | Abre el archivo en modo escritura y posiciona el puntero al final | x |
| `Assign` | Asigna un archivo lógico a su fichero físico | |
| `Close` | Cierra el fichero | |
| `EOF` | Comprueba si el puntero está al final del fichero | |
| `EOLN` | Comprueba si el puntero está al final de línea | x |
| `FilePos` | x | Devuelve la posición del puntero |
| `FileSize` | x | Devuelve el número de registros |
| `Read` | Lee un archivo y asigna los valores a las variables que se le pasan como parámetro | |
| `ReadLn` | Lee el archivo hasta final de línea y asigna los valores a las variables | x |
| `Reset` | Abre el archivo en modo lectura y sitúa el puntero en la primera posición | Abre el archivo en modo lectura/escritura con el puntero en su primera posición |
| `Operación` | Archivos de texto | Archivos binarios |
| `ReWrite` | Si el fichero no existe, lo crea, y si tenía datos, los destruye | |
| `Seek` | x | Sitúa el puntero en la posición especificada |
| `Write` | Escribe en el fichero las variables que se pasan por parámetro | |
| `WriteLn` | Escribe en el fichero las variables que se le pasan e introduce un carácter final de línea | x |

**Tabla 14.1.** Operaciones soportadas por Lazarus sobre ficheros de texto y binarios

## 14.7 ARCHIVOS SIN TIPO

Existe en Lazarus un tercer tipo de variable archivo con la que el programador puede trabajar. Además de los **archivos de texto** y **binarios**, es posible programar también a bajo nivel mediante el empleo de variables de archivo sin tipo. Un **archivo sin tipo** es un canal de E/S de bajo nivel para acceder directamente a cualquier fichero en una unidad de memoria secundaria con independencia de su tipo y estructura. Esto es así porque trabajan con una serie de bloques de bytes que se manipulan independientemente del tipo de datos que contenga el archivo o de la estructura de este.

Los archivos sin tipo se utilizan para transferir de forma rápida grandes bloques de información. Se declaran con la palabra reservada **`File`** sin ninguna otra especificación:

```
var
  NombreFichero: File;
```

Al declarar un archivo sin tipo, automáticamente este quedará estructurado en *n* bloques de *T* bytes cada uno, por lo que en general el tamaño del último bloque puede ser distinto del tamaño definido para el bloque, *T*.

Si bien alguno de los métodos vistos anteriormente puede emplearse con este tipo de archivo, existen procedimientos específicos para su manipulación.

### 14.7.1 Procedimientos de apertura

El formato de los procedimientos de apertura de un archivo sin tipo con los procedimientos `Reset` y `ReWrite` es distinto al usado anteriormente. Sus cabeceras de declaración, en el caso de ser empleadas con variables archivo sin tipo, corresponden, respectivamente, a:

```
Reset (var Fichero: File; bloque: Word);
ReWrite (var Fichero: File; bloque: Word);
```

Donde `bloque` es un número entero que indica el tamaño en bytes o longitud de los bloques en los que queda estructurada la variable `Fichero`. Este segundo parámetro es opcional, si se omite, se configura por defecto un tamaño de bloque de 128 bytes.

## 14.7.2 Procedimientos de entrada y salida de datos

Para escribir o leer datos en una variable archivo sin tipo no se utilizan los procedimientos `Write/WriteLn` o `Read/ReadLn` vistos anteriormente, sino otros específicos para este tipo de archivos, pues no sabemos *a priori* qué datos contiene.

El procedimiento de escritura de datos **BlockWrite** escribe en una variable archivo sin tipo uno o más bloques de datos procedentes de una variable genérica auxiliar. Su cabecera es la siguiente:

```
BlockWrite (var Fichero: File; var buffer; m: Word);
```

El procedimiento de lectura de datos **BlockRead** lee uno o más bloques de datos de una variable archivo sin tipo y los almacena en una variable genérica auxiliar `buffer`. Su sintaxis es la siguiente:

```
BlockRead (var Fichero: File; var buffer; m: Word);
```

La variable auxiliar puede ser de cualquier tipo, aunque generalmente se suele emplear de tipo **array** con el tamaño adecuado al total de bloques manipulados; mientras que la variable `m` expresa el número de bloques que se manipulan en cada llamada al procedimiento.

La diferencia solo estriba en que el procedimiento `BlockWrite` escribe uno o más bloques almacenados en la variable genérica `buffer` en el archivo asociado a la variable `Fichero` y el procedimiento `BlockRead` lee los bloques de bytes del archivo y los almacena en la variable `buffer`. Por lo tanto es importante tener siempre presente el tamaño de esta y el número y tamaño de los bloques con los que se trabaja para evitar problemas en la escritura y lectura de los datos. Por ejemplo, el siguiente fragmento de código sería adecuado para la lectura que se plantea:

```
var
  Archivo: File;
    buffer: array[1..32] of char;
begin
  Assign (Archivo, 'datos.dat');
  Reset (Archivo, 16);
  BlockRead (Archivo, buffer, 2);
```

La sentencia `Reset (Archivo, 16)` estructura el fichero en bloques de 16 bytes y la ejecución de la sentencia `BlockRead (Archivo, buffer, 2)` almacena 2 bloques ($2 \times 16$ bytes) en la variable `buffer`, que previamente se ha

definido como una variable de tipo **array**[1..32] **of** char; es decir, con los 32 bytes necesarios para almacenar todo lo leído (Figura 14.7).

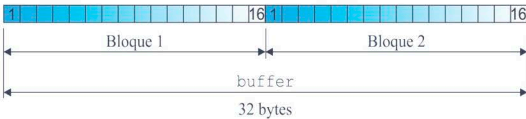



**Figura 14.7.** Archivo estructurado en bloques de 16 bytes y manipulado en grupos de dos

Como ejemplo de manipulación de archivos sin tipo, vamos a implementar un programa con el que podamos cifrar y descifrar ficheros de una forma bastante original para el usuario común, aunque débil para uso comercial. Como vamos a trabajar a bajo nivel, no importará ni qué datos tenga el archivo original ni su estructura.

Abra un nuevo proyecto en modo consola de nombre **CifradorBloques** y añada a la cláusula **uses** la unidad **md5**. Defina, a continuación, las siguientes variables:

```
fuente, destino: File;
nombre_fuente,  nombre_destino, Hash,
nombre_clave   : string;
cont :  byte;
buffer : Array [1..2048] of Byte;
RegLeidos, RegEscritos, Num : Word;
```

Ahora, en el cuerpo del programa, pidamos al usuario el nombre físico de los archivos de origen y destino, y el fichero que se empleará como clave de cifrado:

```
WriteLn ('Archivo Origen  : ');
Readln (nombre_fuente);
WriteLn ('Archivo Destino : ');
Readln (nombre_destino);
WriteLn ('Archivo de Clave: ');
Readln (nombre_clave);
```

Asignemos cada archivo físico con su nombre lógico y definamos el tamaño de los bloques que usaremos con los procedimientos Reset y ReWrite. En este ejemplo vamos a estructurar los archivos en bloques de un único byte, lo que va a facilitar la lectura de cualquier fichero, de cualquier contenido y con cualquier tamaño:

```
AssignFile (fuente, nombre_fuente);
AssignFile (destino, nombre_destino);
Reset (fuente, 1);    // cada bloque 1 byte
Rewrite (destino, 1);  // cada bloque 1 byte
```

Por último, hallamos la función resumen **md5** del fichero, que generará la cadena que emplearemos como clave:

```
Hash := MD5Print (MD5File (nombre_clave));
```

Ya estamos casi en condiciones de implementar el cifrado del archivo `fuente`. Primero leeremos el fichero con el procedimiento `BlockRead` en bloques de 2.048 bytes, `SizeOf (buffer)`, y guardaremos esa lectura en la variable `buffer`. La cantidad real de bytes leídos la almacenaremos en la variable `RegLeidos`.

Ahora, byte a byte, hacemos la operación lógica **xor** entre el archivo origen y la cadena **md5** del archivo clave hasta finalizar. Por último, usamos `Blockwrite` para escribir los bytes procedentes de la variable auxiliar `buffer` en la variable archivo `destino`.

```
Repeat
  BlockRead(fuente, buffer, SizeOf(buffer), RegLeidos);
  for Num := 1 to RegLeidos do
    begin
     buffer[Num] := buffer[Num] xor Ord(Hash[cont]);
     cont += 1;
     if cont > length(Hash) then cont := 1
    end;
  BlockWrite(destino,buffer,RegLeidos,RegEscritos);
Until (RegLeidos = 0) or (RegEscritos <> RegLeidos);
```

Observe que ambos procedimientos de lectura y escritura se encuentran dentro de un bucle `repeat`, que finalizará cuando `RegLeidos` tenga el valor 0, lo que indica que ya no se pudieron leer más bytes del archivo fuente, o cuando se produce un error de escritura, es decir, cuando los bytes leídos y escritos no coincidan. Finalmente, se cierran ambos archivos.

```
if RegEscritos <> RegLeidos
     then Writeln('Error. ¿Disco lleno?');
CloseFile(fuente);
CloseFile(destino);
```

Compile el código fuente del archivo `CifradorBloques.pas` que encontrará en la carpeta del ejemplo y pruebe su funcionamiento. Al archivo destino podrá darle cualquier nombre y extensión, o incluso ninguna. Para descifrar el fichero, tan solo debe emplear como archivo fuente el cifrado. Tenga la precaución de poner la extensión original que tenía y de emplear el mismo archivo de clave en ambos procesos.

En la carpeta le hemos dejado un fichero cifrado de nombre `imagen.jpg.cif.` Pruebe a obtener la imagen original dejando solo la extensión .jpg y utilizando como archivo de clave el fichero `.pas`:

```
Archivo Origen: imagen.jpg.cif
Archivo Destino  : imagen.jpg
Archivo de Clave : cifradorbloques.pas
```

## 14.8 LOS FICHEROS EN LA POO

Junto con la metodología procedural que acabamos de estudiar para la gestión de archivos, Lazarus es capaz de utilizar toda la potencia de la programación orientada a objetos para gestionar ficheros. Para ello usa el concepto de flujo de datos, ya conocido desde el capítulo anterior, como un nivel de abstracción superior que posibilita al programador la gestión de ficheros de una forma más rápida y eficaz.

Como es tradicional, comenzaremos con los ficheros de texto para finalizar con los binarios.

### 14.8.1 Archivos de texto

Para la gestión de ficheros de texto la POO emplea la clase **`TStringList`**, que soporta dos procedimientos esenciales: **`LoadFromFile`**, para **cargar** el fichero en memoria y acceder allí a su contenido, y **`SaveToFile`**, para **almacenar** los elementos de la lista en memoria en el archivo de texto físico.

La gestión de estos ficheros resulta ahora realmente sencilla en comparación con el método procedural tradicional. Si no, vea lo fácil que es crear un archivo de texto con una línea:

```
begin
  with TStringList.Create do
  try
    Add ('Mi primer archivo de texto');
    SaveToFile ('prueba.txt');
  finally
    Free;
  end;
end;
```

Veamos ahora cómo añadir nueva información a la ya existente en el archivo de texto:

```
begin
  with TStringList.Create do
  try
    LoadFromFile ('prueba.txt');
    Add ('con una nueva línea…');
    SaveToFile ('prueba.txt');
  finally
    Free;
  end;
end;
```

El fichero `prueba.txt` tiene ahora el aspecto mostrado en la siguiente figura:

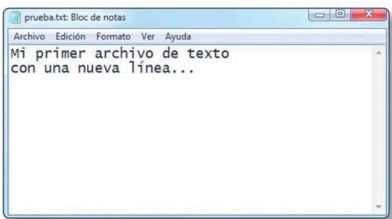



**Figura 14.8.** Aspecto del fichero prueba.txt

La flexibilidad que otorga el uso de listas de cadenas en comparación a los métodos procedurales es enorme, puesto que son de aplicación todos los métodos y propiedades que vimos en la sección 13.3 del capítulo anterior cuando estudiamos la clase `TStringList`. Por ejemplo, eliminar la última línea de nuestro fichero sería tan fácil como escribir:

```
begin
 with TStringList.Create do
  try
    LoadFromFile ('prueba.txt');
    Delete (Count - 1);  // borra la última línea
    SaveToFile ('prueba.txt');
  finally
    Free;
  end;
 end;
```

Por último veamos cómo obtener por pantalla una lectura ordenada de todos los registros de un fichero de texto llamado `provincias.txt`. Abra un proyecto en modo consola con el nombre **`provincias.lpr`** y ajuste su esqueleto para que quede del modo siguiente:

```
program provincias;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;
 var
   i: byte;
   listado: TStringList;
begin
  listado := TStringList.Create;
  WriteLn ('Listado de provincias.txt:', sLineBreak);
  try
    listado.LoadFromFile ('provincias.txt');
    listado.Sort; // se ordenan los registros
    for i := 0 to listado.Count - 1 do
    WriteLn (listado[i]); // y se muestran uno a uno
  finally
```

```
      listado.Free; // liberación del recurso
    end;
    Writeln (sLineBreak, 'Listado acabado. Pulse Enter');
   {$IFDEF WINDOWS}
     ReadLn;
   {$ENDIF}
  end.
```

Cuando compile y ejecute el programa, obtendrá por pantalla el listado, ordenado alfabéticamente, de las provincias recogidas en el fichero que se le adjunta como material descargable.

### 14.8.2 Archivos binarios

La gestión de ficheros binarios empleando el estilo orientado a objetos utiliza la clase **`TFileStream`**. Se trata de una encapsulación de los procedimientos `FileOpen`, `FileCreate`, `FileRead`, `FileWrite`, `FileSeek` y `FileClose`, que se encuentran en la unidad **`FileUtil`** de la LCL.

En primer lugar, antes de su utilización, se deberá crear la instancia de la clase `TFileStream` con la ruta de acceso y el modo de creación y los permisos de lectura y escritura del fichero dado:

```
TFileStream.Create(const NombreFich: string;  Modo: Word);
```

El nombre del fichero tiene que ser válido en Pascal y en el sistema operativo donde se ejecutará la aplicación. El modo indica cómo y con qué permisos se abre el archivo, como ya estudió en la sección 13.4.3 del capítulo anterior.

Creada la instancia, puede escribirse en el flujo de datos cualquier cantidad conocida `Cant` de bytes desde `Buffer` con el método **WriteBuffer()**:

```
TStream.WriteBuffer (const Buffer; Cant: LongInt);
```

O leer del flujo de datos `Cant` bytes en la variable `Buffer` con el método **`ReadBuffer()`**:

```
TStream.ReadBuffer (var Buffer; Cant: LongInt);
```

El primer parámetro de ambos procedimientos (`Buffer`) es una variable sin tipo, en el sentido de que puede contener cualquier dato. Por un lado, esto tiene muchas ventajas, pero puede ser un verdadero quebradero de cabeza si no se sabe usar.

Cuando se empleen datos de tamaño fijo y conocido, como `Integer`, `Byte`, `Char`, etc., tan solo deberá preocuparse por indicar como parámetro `Cant`, `SizeOf()`. Con variables de tamaño variable, como `String`, `PChar` o cualquier puntero, el método no funcionará.

#### 14.8.2.1 ESCRITURA DE ARCHIVOS BINARIOS

Comencemos creando un fichero binario de nombre `enteros.bin` que contenga cinco números enteros aleatorios entre 1 y 1000:

```
var
 i, Buffer: Integer;
begin
   with TFileStream.Create('enteros.bin',fmCreate) do
    try
     for i := 1 to 5 do
     begin
      Buffer := 1 + Random(1000);
      WriteBuffer (Buffer, SizeOf (Buffer));
     end;
    finally
     Free;
   end;
end;
```

Prácticamente, el código no necesita comentarse. Tras crear la instancia con la ruta, el nombre y el modo de apertura del fichero, hemos repetido cinco veces un bucle en el que se genera aleatoriamente un número entre 1 y 1000 que se almacena en la variable de tipo entero `Buffer`. Por último, escribimos en el flujo de datos desde esta los bytes correspondientes a cada registro. Encapsular las acciones en un bloque **`try..finally`** nos asegura que el objeto `Filestream` se liberará siempre, incluso aunque se encuentre cualquier error.

En el momento en que ejecute el programa, se generará en la misma carpeta un archivo binario de 20 bytes, de nombre `enteros.bin`, que contendrá cinco números enteros aleatorios.

Añadir nuevos datos a un archivo binario empleando la clase `TFileStream` es tan sencillo como irse al final del archivo y escribir la nueva información:

```
var
  i, Buffer: integer;
begin
   with TFileStream.Create('enteros.bin',fmOpenWrite)
do
     try
       Buffer:= Random (1000);
       Position:= Size;    // se sitúa al final
       WriteBuffer (Buffer, SizeOf (Buffer));
     finally
       Free;
     end;
end;
```

Observe cómo el código es parecido al anterior, tan solo hemos tenido que abrir el fichero en modo de escritura (`fmOpenWrite`) y situarnos al final del archivo (`Position:= Size`) para escribir un nuevo registro.

Si ha ejecutado el nuevo ejemplo, verá cómo ahora el archivo ocupará 24 bytes, debido a los 6 registros enteros que almacena.

#### 14.8.2.2 LECTURA DE FICHEROS BINARIOS

Para leer un archivo binario empleando la clase `TFileStream`, es necesario abrir el fichero en modo lectura, situarse al principio del mismo e ir leyendo en bloques de tamaño `SizeOf(Buffer)` hasta finalizar.

Veamos cómo implementarlo para leer los números que aleatoriamente se almacenaron en nuestro archivo `enteros.bin`. Abra un programa en modo consola en Lazarus con el nombre **`LecturaBinarios`** y modifique su esqueleto para que se parezca a esto:

```
program LecturaBinarios;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;
var
  BytesTotalesLeidos, BytesLeidos : Int64;
  Buffer, Contador  : Integer;
  Flujo   : TFileStream;
begin
 WriteLn ('Contenido de enteros.bin');
 WriteLn ('-----------------------', sLineBreak);
 try
  Flujo:= TFileStream.Create('enteros.bin', fmOpenRead);
  Flujo.Position := 0;
  while BytesTotalesLeidos < Flujo.Size do
   begin
     BytesLeidos := Flujo.Read(Buffer, SizeOf(Buffer));
     inc (BytesTotalesLeidos, BytesLeidos);
     inc (Contador, 1);
     WriteLn(' ', IntToStr(Buffer));
   end;
  WriteLn(sLineBreak, IntToStr(BytesTotalesLeidos) +
             'bytes leidos');
  WriteLn(IntToStr(Contador)+ ' registros');
 finally
  Flujo.Free;
 end;
{$IFDEF WINDOWS}
  ReadLn;
{$ENDIF}
end.
```

Pulse **[F9]** y obtendrá por pantalla una salida semejante a la mostrada en la figura 14.9.

```
C:\Users\David\Desktop\Pascal I\CD\capitulo14\Flujos\LecturaBin...
Contenido de enteros.bin
------------------------

 549
 593
 716
 845
 603
 548

24 bytes leidos
6 registros
```

**Figura 14.9.** Lectura de un fichero binario con el método ReadBuffer()

Fíjese en que el archivo almacena los seis números enteros (24 bytes) que generamos aleatoriamente en los dos momentos diferentes en los que editamos `enteros.bin`.

Estudiemos con un poco más de detalle el código fuente escrito. Tras abrir el archivo en modo lectura (`fmOpenRead`) nos aseguramos de situarnos al inicio del mismo (`TFileStream.Position := 0`). A continuación, comenzamos a leer del flujo de datos de 4 en 4 bytes en la variable `Buffer`. Recuerde que un entero ocupa 32 bits, 4 bytes. Cada vez que leemos un bloque, acumulamos el número de bytes leídos en la variable `BytesTotalesLeidos` y volcamos en pantalla el registro. ¿Hasta cuándo realizamos esta operación? En el momento en que el número de bytes leídos coincide con el tamaño del archivo (`TFileStream.Size`) salimos del bucle, mostramos el tamaño del fichero y liberamos el recurso.

La aproximación orientada a objetos admite aún más posibilidades para trabajar con ficheros. Es posible, por ejemplo, cargar directamente todo el archivo en la memoria principal y trabajar con él desde allí, lo cual acelera muchísimo el proceso al tratar con flujos de datos en la memoria RAM. Observe el siguiente fragmento de código:

```
with TMemoryStream.Create do
  try
    LoadFromFile ('enteros.bin');
    Position:= Size;
    Dato:= 777;
    Write (Dato, SizeOf (Dato));
    SaveToFile ('enteros.bin');
  finally
    Free;
  end;
```

TMemoryStream es un descendiente de TStream que almacena datos en memoria, con la ventaja de que no es necesario gestionar la memoria manualmente, ya que esta se reservará o liberará dinámicamente según las necesidades.

En el ejemplo de más arriba hemos usado el método LoadFromFile de la clase TMemoryStream para cargar el contenido del archivo en memoria. A continuación, nos situamos al final del mismo y escribimos en el flujo el número 777. Antes de liberar completamente la memoria, guardamos en un archivo el nuevo contenido que se encuentra en ella. Puede ser el mismo u otro distinto. Pruebe el código y vuelva a leer el contenido del fichero con el programa **LecturaBinarios**. Ahora tendrá 7 registros con un contenido de 28 bytes.

## 14.9 UN ÚLTIMO APUNTE: NOMBRES DE FICHEROS

Internamente, Lazarus trabaja con el formato de codificación de caracteres UTF-8. Algunos sistemas operativos, entre ellos Windows, lo hacen con codificaciones diferentes. Las rutinas de la biblioteca en tiempo de ejecución del compilador de Free Pascal emplean la codificación del propio sistema, por lo que es necesario algún método de conversión cuando se trabaja con las rutinas tradicionales de Pascal.

Imagine que tiene un fichero de texto llamado España.txt con el nombre de las 50 provincias y las dos ciudades autónomas. Para abrir este fichero, llamado EspTxt en su programa y volcar su contenido en un componente de memo, podría escribir lo siguiente:

```
procedure TForm1.CargarProvinciasEnMemo1;
var  EspTxt: TextFile;
     fNombre: string = 'España.txt';
     Provincia: string;
begin
  AssignFile (EspTxt, UTF8ToSys(fNombre));
  try
    Reset (EspTxt);
    while not Eof(EspTxt) do
    begin
      readln (EspTxt, Provincia);
      Memo1.Lines.Add (Provincia);
    end;
  except
    ShowMessageFmt('El fichero de texto %s no se encuen
                   tra', [fNombre]);
  end;
end;
```

Si se le olvida poner la función de conversión de codificación `UTF8ToSys (fNombre)`, la aplicación no encontrará el archivo, puesto que contiene una letra *ñ*, y la llamada a `Reset(EspTxt)` fallará lanzando una excepción (Figura 14.10).

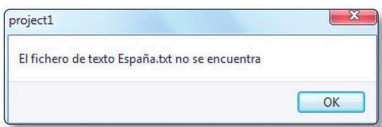



**Figura 14.10.** Excepción lanzada por un error en la codificación de caracteres

Aunque es una mala práctica en el desarrollo de software incrustar datos directamente en el código fuente del programa, en vez de obtenerlos de una fuente externa, sirve muy bien para ilustrar la necesidad de codificar correctamente los nombres de sistema.

En realidad, podríamos incluso simplificar mucho más el código eliminando el bucle **`while`** y aprovechar el método `LoadFromFile` de la propiedad `TMemo.Lines`. De modo que bastaría con escribir esto:

```
procedure TForm1.CargarProvinciasEnMemo1;
var  EspTxt: TextFile;
     fNombre: string = 'España.txt';
     Provincia: string;
begin
  AssignFile (EspTxt, UTF8ToSys(fNombre));
  try
    Memo1.Lines.LoadFromFile (UTF8ToSys (fNombre));
  except
    ShowMessageFmt('El fichero de texto %s no se encuen
                tra', [fNombre]);
  end;
end;
```

Como hemos comentado, es una mala práctica incrustar los nombres de archivos directamente en el código fuente del programa, en vez de obtenerlos de una fuente externa, por lo que la única posibilidad de hacerlo con total seguridad es a través de las ventanas de diálogo que Lazarus implementa en el propio entorno de desarrollo. Estos controles se encuentran en la pestaña *Dialogs* de la

Paleta de componentes y permitirán **Abrir** y **Guardar** ficheros de forma segura independientemente de la plataforma en la que se trabaje (Figura 14.11).

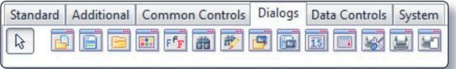



**Figura 14.11.** Pestaña Dialogs de la Paleta de componentes

Los distintos componentes de la ficha *Dialogs* comparten ciertas propiedades comunes, de las que las dos más importantes son **`DefaultExt`**, que define la extensión que se añade automáticamente al fichero del usuario, si este lo teclea manualmente, y **`FileName,`** que asigna un nombre por defecto al archivo. Junto a estas propiedades, también comparten el método booleano **`Execute`**, que devuelve `True` si el usuario ha seleccionado un archivo y pulsado el botón **Abrir**/**Guardar** de la ventana de diálogo.

Los componentes visuales **`TOpenDialog`** y **`TSaveDialog`** se complementan con `TOpenPictureDialog` y `TSavePictureDialog`, especialmente diseñados para trabajar con ficheros gráficos. La propiedad Filter para estos controles recoge los formatos más comunes de archivos gráficos que Lazarus soporta.

En Lazarus, a menudo, suele emplearse con más eficacia el control `TFileNameEdit` de la pestaña *Misc* de la Paleta de componentes, que combina los cuadros de diálogo **Abrir** y **Guardar** junto con un botón para invocarlos según el valor de la propiedad **`DialogKind.`** En estos controles, el programador puede completar a voluntad la propiedad **`Filter`** para que se muestren solo esos tipos de archivos en los cuadros de diálogo (Figura 14.12).

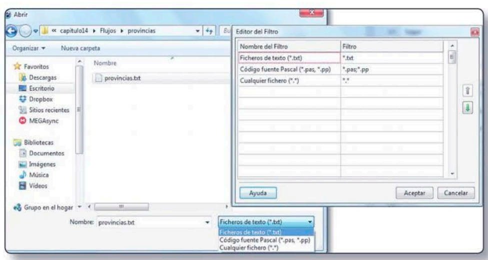



**Figura 14.12.** Editor del filtro de archivos del componente TFileNameEdit y cuadro de diálogo Abrir

# 15

# LOS LÍMITES DE LA PROGRAMACIÓN

Pascal es un lenguaje fuertemente tipado, lo que significa que para cada dato o variable se ha de especificar de qué tipo es antes de usarla, lo que implica conocer por adelantado el máximo tamaño que puede almacenar. Consecuentemente, ha de analizarse muy bien con qué tipos de datos va a trabajar la aplicación, pero, aun así, esto no evita que durante su funcionamiento puedan excederse los límites. ¿Qué ocurrirá entonces? Todo dependerá de si el programador ha previsto esa posibilidad y actuado en consecuencia para permitir que el programa pueda gestionar un desbordamiento semejante.

En este capítulo trabajaremos en los límites de la programación para aprender a gestionar casos en los que las variables se queden muy pequeñas para los datos que deben contener. Así mismo, introduciremos el concepto de hilo de ejecución, para que una aplicación pueda realizar varias tareas a la vez de forma concurrente. De este modo, un hilo podrá estar atento a la propia interfaz gráfica de la aplicación mientras otro realiza una pesada operación interna.

Lazarus, como Java, implementa características de diseño expresamente creadas para permitir a los programadores trabajar de forma nativa con hilos de ejecución sin necesidad de crearlos mediante llamadas a bibliotecas específicas que depeden del propio sistema operativo, como ocurre con C y C++, por ejemplo.

## 15.1 RECURSIVIDAD Y NÚMEROS FACTORIALES

El **factorial** de un número natural *N* se define como el producto de todos los enteros no negativos menores o iguales que *N*, asumiendo que $0! = 1! = 1$.

Ya en el Capítulo 6, cuando estudiamos las sentencias de control, realizamos un pequeño programa con un bucle **for** para hallar números factoriales. Ahora vamos a emplear la **recursividad** para evaluar el resultado mediante una rutina que se llame a sí misma el número necesario de veces. Es una técnica sencilla y poderosa, pero con un lado oscuro importante. El éxito de una rutina recursiva depende de que las sucesivas llamadas a la misma acaben en algún momento, pues, de otro modo, se entraría en un bucle sin fin con pocas soluciones posibles, salvo matar la ejecución del programa o apagar el ordenador.

El valor de $N!$ aumenta muy rápidamente. De hecho, el mayor valor que puede almacenarse en una variable de tipo entero de 32 bits es 12!, mientras que para un entero de 64 bits es 20!

Así pues, para nuestro siguiente ejemplo vamos a implementar una función recursiva de 64 bits. Abra una nueva aplicación y dele al proyecto el nombre **factorial**. En la sección privada del formulario añada el método `Factorial64`:

```
private
function Factorial64(aByte: byte): int64;
```

Emplee la compleción automática de código y escriba lo siguiente:

```
function TForm1.Factorial64(aByte: byte): int64;
begin
 case aByte of
     0, 1: Result:= 1;
 else Result:= aByte * Factorial64(aByte - 1);
  end;
end;
```

Observe que para los valores 0 y 1 la función devuelve el valor 1, mientras que para otros números naturales se llama sucesivamente a sí misma hasta que el valor que se le pasa como parámetro es 1. En ese momento, se acaban las llamadas y se devuelve el producto resultante.

Para probar su funcionamiento en modo gráfico, arrastre un botón a la parte superior del formulario, llámele `btnFactoriales` y póngale como leyenda **Hallar factoriales hasta**:

Al lado del botón arrastre un control `TSpinEdit` de nombre `seNumero` con su propiedad `Value` fijada en 20 y su valor máximo en 25.

Bajo ambos controles ponga un campo de `memo` de nombre `mMostrar` alineado al pie (`alBottom`) con barras de desplazamiento a ambos lados (`ssAutoBoth`). Arrastre la parte superior del campo para cubrir la mayor superficie del formulario.

Genere ahora un evento `OnClick` para el botón y complete el código como sigue:

```
procedure TForm1.btnFactorialesClick(Sender: TObject);
var i: integer;
begin
  mMostrar.Lines.Clear;
  for i := 1 to seNumero.Value do
    mMostrar.Lines.Add(Format('%d! es %s',[i,
            FormatFloat(',#', Factorial64(i))]));
end;
```

Compile y ejecute el programa y vea lo que ocurre por encima de 20! ¿Observa algo extraño? Si no se fija atentamente, puede que le pase inadvertido que a partir de 20! empiezan a aparecer valores negativos, lo cual es imposible en un factorial (Figura 15.1).

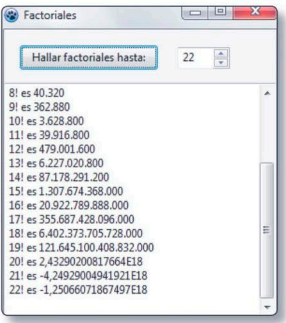



**Figura 15.1.** Salida del ejemplo Factoriales

Lo curioso de la salida del ejemplo es que el desbordamiento producido pasa inadvertido. ¿Podemos solucionarlo de algún modo? Existen posibles soluciones, desde luego. El remedio más sencillo es impedir que el programa acepte valores por encima de 20! con un entero de 64 bits. Bastaría con fijar el parámetro `MaxValue` del control `TSpinEdit` en 20. También podríamos usar variables que pudiesen almacenar números mayores, como `qword` o `extended`. Una tercera alternativa más compleja sería utilizar un bloque **`try...except`** para manejar el desbordamiento mediante la excepción `EIntOverflow`. Solo necesitamos modificar levemente el código y advertir al compilador sobre posibles desbordamientos. Para ello, pulse la combinación **[Ctrl] + [May] + [F11]**. En el visor de la izquierda, bajo el nodo **Opciones del Compilador**, haga clic en **Depurando información** y marque en la nueva hoja las casillas de verificación correspondientes a **Desbordamiento** y **Pila** (Figura 15.2). La pila es una estructura de datos en memoria que resulta esencial para trabajar con subrutinas, incluidas las recursivas, e interrupciones.

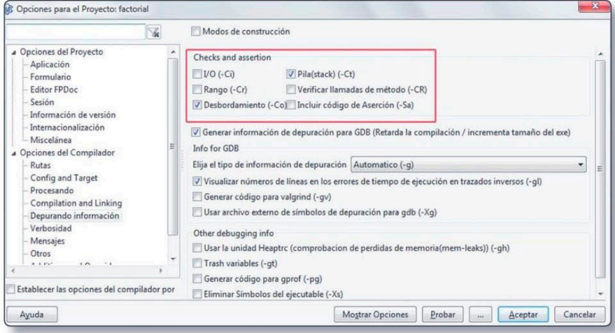



**Figura 15.2.** Opciones de la página Depurando información para el proyecto

Para ver ahora cómo tratar en el código la excepción que provoca el desbordamiento, añada un nuevo botón de nombre `btnFactorialesExc` con la leyenda **Hallar factoriales con excepciones**. Haga doble clic sobre él para generar un manejador `OnClick` y complete el código:

```
procedure TForm1.btnFactorialesExcClick(Sender: TObject);
  var i: integer;
begin
  mMostrar.Lines.Clear;
```

```
    for i := 1 to seNumero.Value do
      mMostrar.Lines.Add(Format('%d! es %s',[i, FormatFloat
                   (',#', Factorial64Exc(i))]));
  end;
```

El código es idéntico al del otro botón, salvo que llamamos a una función diferente. Ahora solo resta escribir el método `Factorial64Exc()`. Declárelo en la sección privada del formulario:

```
function Factorial64Exc(aByte: byte): int64;
```

Y use la compleción automática de código para completar su esqueleto de la siguiente forma:

```
function TForm1.Factorial64Exc(aByte: byte): int64;
begin
  Result:= -1;
  case aByte of
    0, 1: Result:= 1;
  else
    try
    Result:= aByte * Factorial64Exc(aByte - 1);
    except on e:EIntOverflow do
     MessageDlg ('Problema',
     Format('"%s"%s El número %d ha causado en la función
       Factorial64Exc' + ' el desbordamiento señalado ',
     [e.Message, LineEnding, aByte]), mtWarning, [mbClose]
                           , 0);
    end;
  end;
end;
```

Al encerrar la llamada a la función `Factorial64Exc()` dentro del bloque **`try/except`**, obligamos al compilador a comprobar en todo momento la excepción `EIntOverflow`. Cuando aparezca, llamará a la función `MessageDlg`, con la que hemos construido un mensaje con la explicación del desbordamiento. El compilador se encargará, así mismo, de gestionar la excepción y liberarla, por lo que el programa puede seguir con su ejecución sin ningún problema.

Observe cómo al principio del método se ha establecido como resultado −1. De modo que cuando se produzca la excepción este será el valor que devuelva la función, lo que nos alertará, además, de la invalidez del resultado. En otros ejemplos bastaría con indicar el error o un código de error. Cuando no se produzca el desbordamiento, el resultado se sobrescribe en la sección **`try`** de la función.

La salida del programa será ahora la mostrada en la figura siguiente:

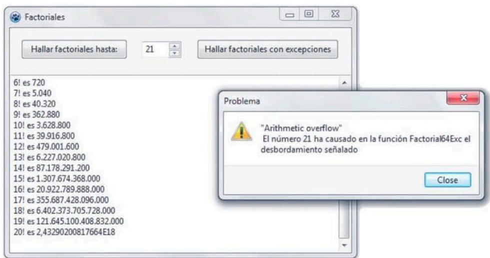



**Figura 15.3.** Desbordamiento aritmético para el número 21!

## 15.2 PERMUTACIONES

El factorial de un número *n* indica el número de formas diferentes de ordenar esos *n* elementos sin repetirlos, es decir, el número total de **permutaciones** para esos elementos. Cuando se permutan las letras de una palabra se obtienen otros términos que se denominan **anagramas**. Si tomamos la voz castellana **aro**, por ejemplo, sus posibles anagramas son seis, pues para estas tres letras hay 3! permutaciones posibles: aro, aor, rao, roa, oar y ora; aunque solo tres de ellas tengan significado en nuestro idioma.

Es bastante sencillo diseñar un procedimiento recursivo que nos devuelva los anagramas de una palabra dada. Básicamente, es tan fácil como mover cada letra a la primera posición y escribir las distintas permutaciones de las letras que quedan. Para ello, movemos otra letra a la segunda posición y permutamos el resto, y así sucesivamente. De modo que tenemos un algoritmo recursivo que puede extenderse, al menos teóricamente, a palabras de cualquier longitud.

Como vimos en el ejemplo anterior, el número de posibles permutaciones aumenta rapidísimamente con cada nuevo elemento, por lo que, a partir de unas 8 letras, las limitaciones de tiempo se hacen realmente importantes. Veamos cómo el

tiempo es, en efecto, el único factor limitante en un proceso que genera todos los anagramas posibles de una palabra dada por el usuario.

Abra un nuevo proyecto en Lazarus con el nombre **anagramas** y arrastre un componente `TLabelEdit`, de nombre `EPalabra`, hasta la parte superior del formulario y una etiqueta de nombre `LContarAnagramas` bajo aquella. Ajuste el resto de propiedades con los datos obtenidos de su fichero `.lfm`:

```
object Form1: TForm1
  Height = 450
  Width = 580
  Caption = 'Generador de anagramas'
  object EPalabra: TLabeledEdit
    Left = 194
    Height = 23
    Top = 16
    Width = 110
    EditLabel.Caption = 'Generar anagramas de la
                         palabra '
    LabelPosition = lpLeft
    MaxLength = 10
  end
  object LContarAnagramas: TLabel
    Caption = 'LContarAnagramas'
  end
```

Haga doble clic sobre **EPalabra** para generar un manejador del evento `OnChange` y complete el código siguiente:

```
procedure TForm1.EPalabraChange(Sender: TObject);
var long: integer;
    f   : int64;
    function Plural(x: int64): char;
    begin
         Result:= #0;
         if (x > 1) then Result := 's';
    end;
begin
 long:= EPalabra.GetTextLen;
 f:= Factorial(long);
 LContarAnagramas.Caption:= Format('%s generará %s
     anagrama%s', [EPalabra.Text, FormatFloat(',#',f),
                        Plural(f)]);
end;
```

Este procedimiento indica en tiempo real cuántos anagramas se generarán de acuerdo con el número de letras del término escrito. Además, se ha introducido una pequeña función para que el programa escriba correctamente "anagrama" o "anagramas" según el número de permutaciones indicado por la función `Factorial()`, que aún no se ha implementado y que escribiremos ahora mismo. En la sección privada del formulario declare esta función:

```
function Factorial(unEnt: integer): int64;
```

Genere su esqueleto y complételo con este código:

```
function TForm1.Factorial(unEnt: integer): int64;
begin
 result := -1;
 if (unEnt < 0) then Exit;
 case unEnt of
      0, 1: Result:= 1;
       else Result:= unEnt*Factorial(Pred(unEnt));
 end;
end;
```

El código es semejante al que escribimos en el punto anterior salvo por dos pequeños detalles introducidos para optimizar su rendimiento. El primero es que usamos enteros de 32 bits en vez de variables de tipo `Byte`, puesto que en los sistemas operativos de 32 bits esto hace que el proceso sea más rápido. El segundo detalle es que usamos en cada paso la función `Pred()` en vez de calcular `unEnt - 1`, lo que también mejora la velocidad.

Añada ahora dos etiquetas más, `LTiempoCalculo` y `LProgreso`, a la derecha de `EPalabra`. Bajo `LContarAnagramas` arrastre dos botones más. Al primero llámelo `BLento` y ponga como leyenda **Generar anagramas (lento)**, y al segundo, `BRapido`, con la leyenda **Generar anagramas (rápido)** (Figura 15.4).

Haga doble clic sobre cada uno de los botones para generar los manejadores de los eventos `OnClick` y complete sendos códigos así:

```
procedure TForm1.BLentoClick(Sender: TObject);
begin
  LBPalabras.Items.Clear;
  LBPalabras.Items.Add('[Lento]');
```

```
    numPalabras:= 0;
    inicio:= Now;
    LBPalabras.Items.BeginUpdate;
    GenerarAnagramasLento(EPalabra.Text,1); LBPalabras.
                         Items.EndUpdate;
    LTiempoCalculo.Caption:= 'Tiempo de cálculo: ' +
               FormatDateTime ('n"m" s,z"s"', Now-inicio);
end;

procedure TForm1.BRapidoClick(Sender: TObject);
begin
  LBPalabras.Items.Clear;
  numPalabras:= 0;
  ss := TStringStream.Create('[Rápido]'+LineEnding);
  ss.Seek(0, soFromEnd);
  inicio:= Now;
  LBPalabras.Items.BeginUpdate;
  try
     GenerarAnagramasRapido(EPalabra.Text, 1);
     LProgreso.Caption:= 'Cargando lista...';
     Application.ProcessMessages;
     ss.Seek(0, soFromBeginning);
     LBPalabras.Items.LoadFromStream(ss);
  finally
     ss.Free;
  end;
  LBPalabras.Items.EndUpdate;
  LTiempoCalculo.Caption:='Tiempo de cálculo: '+
                FormatDateTime('n"m" s,z"s"', Now-inicio);
end;
```

La rutina rápida trabaja con los anagramas en dos fases. Primero crea un flujo de datos con `TStringStream` y lo usa para almacenar los anagramas mientras se generan. Finalizado el proceso, rellena el campo de listas `LBPalabras` con la llamada `LBPalabras.Items.LoadFromStream()`. El trabajo con flujos de datos acelera muchísimo el proceso, como podrá comprobar en breve.

La rutina lenta, por el contrario, añade los anagramas al cuadro de listas con la propiedad `LBPalabras.Items.Text`. A medida que aumenta el número de caracteres de la palabra, el proceso se hace más y más lento.

El último control de la aplicación es un componente `TListBox` de nombre `LBPalabras` alineado al pie del formulario. Establezca su altura en 320, el número de columnas en 6 y su propiedad `ScrollWidth` en 580 (Figura 15.4).

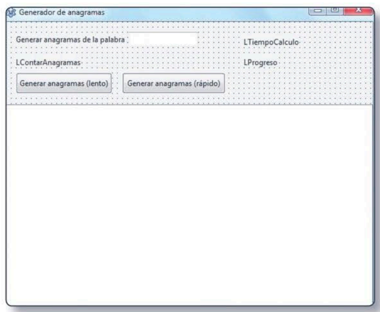



**Figura 15.4.** El proyecto Generador de anagramas en tiempo de diseño

Antes de que podamos ejecutar la aplicación, necesitamos añadir algunos campos y métodos más en la sección privada del formulario:

```
inicio: TDateTime;
numPalabras: int64;
ss: TStringStream;
procedure GenerarAnagramasLento(unaPalabra: string;
                                posCar: integer);
procedure GenerarAnagramasRapido(unaPalabra: string;
                                posCar: integer);
procedure Intercambio(var a, b: Char);
```

Invoque la compleción automática de código y complete los tres métodos de la siguiente forma:

```
procedure TForm1.GenerarAnagramasLento(unaPalabra:
string;
       posCar: integer);
var long, p: integer;
begin
  long := Length(unaPalabra);
  if (posCar >= long) then
```

```
      begin
        LBPalabras.Items.Text := Format('%s%s', [LBPalabras.
              Items.Text, unaPalabra]); inc(numPalabras);
         if (numPalabras mod 10) = 0 then
           LProgreso.Caption:= Format('%d anagramas generados
                                          ',[numPalabras]);
           Application.ProcessMessages;
         end
    else for p:= posCar to long do
       begin
        Intercambio(unaPalabra[p], unaPalabra[posCar]);
        GenerarAnagramasLento(unaPalabra, Succ(posCar));
        Intercambio(unaPalabra[p], unaPalabra[posCar]);
       end;
  end;
  procedure TForm1.GenerarAnagramasRapido(unaPalabra:
  string; posCar: integer);
  var long, p: integer;
  begin
    long := Length(unaPalabra);
    if (posCar >= long) then
      begin
       ss.WriteString(unaPalabra + LineEnding);
       inc(numPalabras);
       if (numPalabras mod 10) = 0 then
        LProgreso.Caption:= Format('%d anagramas generados'
                                              ,[numPalabras]);
        Application.ProcessMessages;
      end
    else for p:= posCar to long do
      begin
       Intercambio(unaPalabra[p], unaPalabra[posCar]);
       GenerarAnagramasRapido(unaPalabra, Succ(posCar));
       Intercambio(unaPalabra[p], unaPalabra[posCar]);
      end;
  end;
  procedure TForm1.Intercambio(var a, b: Char);
  var tmp: Char;
  begin
    tmp := a;
    a := b;
    b := tmp;
  end;
```

Compile y ejecute el programa. Introduzca una palabra y observe la diferencia de tiempos de cómputo entre ambas rutinas (Figura 15.5).

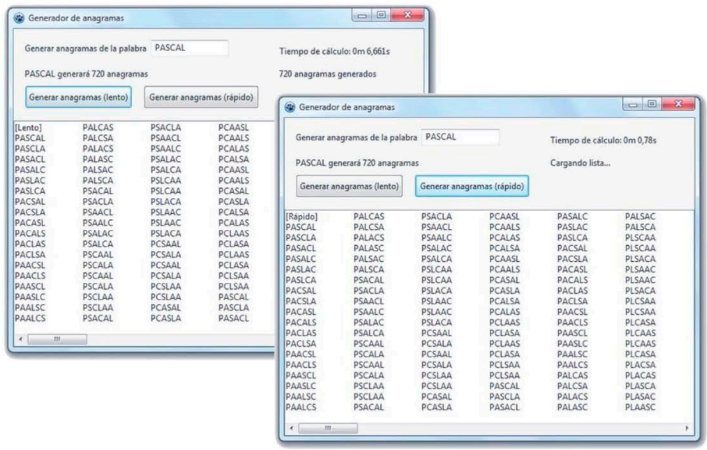



**Figura 15.5.** Generación de anagramas con las rutinas lenta y rápida

En el ejemplo de arriba hemos generado los 720 anagramas de la palabra *PASCAL* con ambas rutinas. Con la rutina lenta, el ordenador ha tardado 6,66 s, mientras que empleando un flujo de datos solo ha necesitado 0,78 s. ¡Sorprendente!

Pruebe ahora con alguna palabra de siete letras, 5.040 anagramas, y verá cómo la diferencia de tiempos es todavía mayor. Si tiene tiempo y quiere llevar al microprocesador al límite, intente generar anagramas de términos de ocho letras o más con la rutina lenta. Las dos funciones que hemos implementado en el código hacen lo mismo, son igualmente válidas y utilizan el mismo algoritmo recursivo para intercambiar los caracteres individuales; sin embargo, su modo de generar y almacenar los anagramas es distinto, y esa diferencia hace que el rendimiento sea muy dispar.

## 15.3 TIEMPO Y COMPUTACIÓN

Los ordenadores son extremadamente rápidos calculando y moviendo datos entre el microprocesador y la memoria principal, y relativamente lentos mostrándolos en pantalla, y más lentos aún cuando se trata de leer y escribir esa información en la memoria secundaria (discos y otros dispositivos). Como acabamos de ver, generar y mostrar todas las permutaciones de un término puede implicar invertir una gran cantidad de tiempo. Imagine cuánto tardaríamos en generar todas las contraseñas posibles de ocho caracteres que incluyan letras mayúsculas, minúsculas y los diez dígitos del sistema decimal. Existen $62^8 = 2{,}2 \cdot 10^{14}$ palabras. ¿Qué puede hacer el programador para enfrentarse a rutinas que deban realizar tareas tan pesadas? El microprocesador de la máquina estará prácticamente dedicado a semejante trabajo. El programa parecerá estar congelado y en algunos sistemas operativos se observará el mensaje "No responde" en la barra de título, aunque el programa siga trabajando. Los usuarios, en estos casos, no saben qué hacer, pues puede ser que la aplicación haya dejado de funcionar verdaderamente o lo siga haciendo. Para ello, lo mejor es que cuando nos enfrentemos a programas así podamos indicar al usuario el progreso del mismo mediante información visual en la pantalla. Esta situación, desde luego, es consecuencia de la propia naturaleza **monohilo** de los programas. Los microprocesadores multinúcleo, que están en el mercado desde hace ya muchos años, son capaces de ejecutar varios hilos simultáneamente. De modo que, a menos que creemos en Lazarus varios hilos en las rutinas, solo podremos conformarnos con el hilo principal que genera cualquier aplicación en nuestro entorno de desarrollo.

Si intentamos monopolizar el microprocesador con una pesada rutina que puede tardar incluso horas en completarse, la interfaz aparecerá congelada. Hasta que se complete, la actividad del ratón o del teclado se verá seriamente comprometida y puede que incluso ni respondan.

Los bucles en Lazarus se controlan por el objeto global `Application` que se crea con cada proyecto, como vimos brevemente en el Capítulo 10. El método `Run` de `TApplication` comienza mostrando el formulario principal. Después llama repetidamente al método `ProcessMessages`, que, en esencia, consulta si existe algún mensaje pendiente. Si es así, `ProcessMessages` envía este mensaje al destino de control apropiado. Esto significa que el programador puede hacer que una aplicación gráfica sea más receptiva llamando explícitamente a `Application.ProcessMessages` en la rutina, tal como hemos hecho en el ejemplo anterior.

Otra opción mejor es ejecutar las rutinas más pesadas en un **hilo** independiente para evitar saturar la aplicación. La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez, de forma concurrente.

Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

Un ejemplo de la utilización de hilos es tener un hilo atento a la interfaz gráfica (iconos, botones, ventanas), mientras otro hilo hace una larga operación internamente. De esta manera, el programa responde de manera más ágil a la interacción con el usuario.

Dado que este libro está dedicado a aprender las técnicas básicas de programación, no entraremos en detalle en la programación de hilos, que es una parte realmente compleja dentro del paradigma de la programación dirigida por eventos, pero daremos una visión global para el programador novel.

## 15.4 DE EVENTOS A HILOS

La idea básica de la programación guiada por eventos consiste en que hay determinados eventos que establecen el flujo de control de la aplicación. Los programas pasan la mayor parte del tiempo esperando a que ocurran dichos eventos para ofrecer el código correspondiente a cada uno de ellos. Cuando se pulsa el botón del ratón, por ejemplo, se produce un evento que envía un mensaje a la ventana que en ese momento está bajo el cursor del ratón. El código del programa que responde a ese evento lo recibe, procesa y responde según corresponda.

Cuando la aplicación termina de responder al evento, vuelve al estado de espera. Los eventos, por tanto, funcionan en serie, es decir que cada evento se controla cuando ha terminado el anterior. Cuando el programa está ejecutando código, los demás eventos de la aplicación han de esperar en una cola de mensajes, a menos, claro está, que el programador haya implementado distintos hilos de ejecución.

No obstante, aunque no haya hilos, cuando ha pasado un tiempo dado, el sistema interrumpe la aplicación actual y pasa automáticamente el control a la siguiente de la lista. Es lo que se denomina **multitarea** no cooperativa. Por tanto, una aplicación que realice una operación que ocupa mucho tiempo no evita que el sistema funcione correctamente, pero normalmente no puede ni pintar siquiera sus propias ventanas, lo que causa un efecto horrendo. Si se añade la llamada `Application.ProcessMessages` dentro de un bucle consumidor de tiempo, como el que teníamos en las rutinas de nuestro ejemplo de anagramas, la operación se ralentiza, pero, a cambio, se refresca más rápidamente. Este es el código que utiliza esta aproximación:

```
if (posCar >= long) then
 begin
   ss.WriteString(unaPalabra + LineEnding);
   inc(numPalabras);
     if (numPalabras mod 10) = 0 then
     LProgreso.Caption:= Format('%d anagramas generados
                                  ',[numPalabras]);
     Application.ProcessMessages;
 end
```

### 15.4.1 Multihilos en Lazarus

Cuando sea necesario realizar operaciones en segundo plano, o cualquier proceso no relacionado estrictamente con la interfaz gráfica, se puede utilizar la aproximación más correcta desde el punto de vista técnico: crear un **hilo de ejecución** separado dentro del propio proceso. Aunque es un tema complejo, no queda más remedio que tratarlo desde el punto de vista de sus fundamentos, pues en el mundo de la programación para Internet pocas cosas pueden hacerse sin hilos.

Lazarus posee una clase heredada directamente de `TObject`, **`TThread`**, que permite crear y controlar hilos. Esta clase no puede usarse directamente, pues es abstracta, sino que heredamos de ella y utilizamos las características de esta clase base.

Por lo general, solo hay que sobrescribir dos métodos: el **constructor** `Create` y el **método** `Execute`. El constructor requiere un único parámetro (`Suspended`) que permite elegir entre ejecutar el hilo inmediatamente o dejarlo en espera. Cuando el objeto hilo arranca automáticamente, o cuando se reanuda su ejecución, mantiene el método `Execute` en funcionamiento hasta el final.

La clase `TThread` incluye dos métodos básicos:

```
procedure Execute; virtual; abstract;
procedure Synchronize(Method: TThreadMethod);
```

El método `Execute` declarado como un procedimiento abstracto virtual, debe ser redefinido en cada hilo. Contiene el código principal del hilo, es decir, el código que habitualmente se utiliza en una funcion de hilo al usar las funciones del sistema. El método `Synchronize` recibe un parámetro, que consiste en un método de nuestra clase, que a su vez no tiene parámetros. Cuando se llama a este método a través de `Synchronize(@MiMetodo)`, la ejecución del hilo se detendrá, el código

de MiMetodo se ejecutará en el hilo principal y, a continuación, se reanudará la ejecución del hilo.

Para ver un ejemplo de hilo, vamos a estudiar el programa **primos**, que tiene a su disposición como material descargable. Produce un hilo secundario para hallar cuántos números primos existen hasta un número máximo introducido por un usuario. La clase hilo tiene el método Execute, un valor inicial pasado mediante una propiedad pública (Max) y dos valores internos (FTotal y FPosicion) empleados para sincronizar la salida de los métodos MostrarTotal y ActualizarProgreso. Esta es la declaración completa para el objeto hilo:

```
type
  { TAnadePrimo }
  TAnadePrimo = class (TThread)
    private
    FMax, FTotal, FPosicion: Integer;
  protected
    procedure Execute; override;
    procedure MostrarTotal;
    procedure ActualizarProgreso;
  public
    property Max: Integer read FMax write FMax;
end;
```

El método Execute va a ejecutar el algoritmo sincronizado con los métodos ActualizarProgreso y MostrarTotal, como puede comprobar:

```
procedure TAnadePrimo.Execute;
var
  I: Integer;
begin
  FMax := Form1.seNumero.Value;
  for I := 1 to FMax do
  begin
    if esPrimo (I) then
     begin
       FTotal := I;
    Synchronize (@MostrarTotal);
    end;
  end;
```

```
          if FMax mod I  = 0 then
            begin
              FPosicion := I * 100 div FMax;
              Synchronize (@ActualizarProgreso);
            end;
      end;

      procedure TAnadePrimo.MostrarTotal;
      begin
       Form1.LBPrimos.Items.Add (IntToStr(FTotal));
      end;

      procedure TAnadePrimo.ActualizarProgreso;
      begin
       Form1.ProgressBar1.Position := FPosicion;
      end;
```

El hilo se crea al hacer clic sobre el botón **Calcular primos** y se destruye automáticamente cuando termina el método `Execute`:

```
      procedure TForm1.CalculoClick(Sender: TObject);
       var
        AnadirHilo: TAnadePrimo;
      begin
        LBPrimos.Items.Clear;
        ProgressBar1.Position := 0;
        AnadirHilo := TAnadePrimo.Create (True);
        AnadirHilo.FreeOnTerminate: = True;
        AnadirHilo.Max := Form1.seNumero.Value;
        AnadirHilo.Start;
      end;
```

Si el hilo tiene un bucle, este debe terminar cuando `Terminated` es `True`, por tanto, es preciso comprobar el valor de `Terminated`, y si es cierto salir del método `Execute` tan pronto y limpiamente como sea posible.

Hay que tener en cuenta que el método `Terminate` no hace nada por defecto: el método `Execute` debe realizar explícitamente todas las tareas necesarias para finalizar el trabajo.

Como hemos explicado antes, el hilo no debe interactuar con los componentes visibles. Para mostrar algo al usuario hay que hacerlo en el hilo principal. Para hacer esto, `TThread` usa el método `Synchronize`. Cuando se llama a este método a través de `Synchronize(@MostrarTotal)` y `Synchronize (@ActualizarProgreso)`, la ejecución del hilo se detendrá, se ejecutará el código de los métodos `MostrarTotal` y `ActualizarProgreso` en el hilo principal y, a continuación, se reanudará la ejecución del hilo secundario.

Existe otra propiedad importante de `TThread` que hemos usado en el manejador del evento `OnClick` del botón: `FreeOnTerminate`. Si esta propiedad es `True`, el hilo se liberará automáticamente al finalizar la ejecución del método `Execute`.

En lugar de fijar el número máximo utilizando una propiedad, hubiera sido mejor pasar este valor como un parámetro adicional de un constructor hecho a medida, pero, para nuestro ejemplo, hemos preferido centrarnos en el uso del hilo.

# 16

# TÉCNICAS DE DEPURACIÓN

Un error de software –comúnmente conocido en el campo de la informática por su término en inglés, *bug*– es un fallo en un programa o sistema de software que provoca un resultado indeseado. La mayor parte de los errores los cometen las personas, es de humanos errar, bien a la hora de escribir el código fuente o previamente en su proceso de diseño. En otras ocasiones son los propios sistemas operativos en los que se ejecutan los programas los responsables de los fallos, y, en menor medida, puede culparse a los propios compiladores de generar código incorrecto.

Los **errores**, en general, pueden clasificarse en tres grandes grupos: **sintácticos**, **lógicos** y **de ejecución**. Los errores de sintaxis en el código son los más fáciles de corregir, pues el compilador informará de ello en el momento de compilar, como le habrá ocurrido más de una vez en el desarrollo de las aplicaciones propuestas en el libro. Los errores lógicos y de ejecución, por el contrario, solo se pondrán de manifiesto en tiempo de ejecución durante su uso normal, tras la perfecta compilación y enlazado del proyecto. En estas situaciones, solventar el problema no es tan sencillo y requiere mucho más trabajo que simplemente mirar por encima el código del programa.

No se trata solo de reconocer el error, sino de entender la causa que provoca una salida tan diferente a la esperada para poder resolverla. Algunos fallos tendrán fácil solución, otros serán difíciles de precisar y podrán provocar graves problemas. La historia está llena de ellos. Pueden ocurrir sin motivo aparente, sin un patrón que permita predecirlos, y, en ocasiones, ni siquiera se podrán solventar.

En este último capítulo aprenderá algunas técnicas básicas para detectar y corregir los pequeños errores de software que suele cometer un programador novel.

## 16.1 PREVENCIÓN DE ERRORES

El modo más sencillo de evitar errores es no escribir código desde cero, desde luego. Seguiríamos siendo programadores, no hay que menospreciarse, pero emplear rutinas ya escritas y probadas por miles de usuarios permite tener una mayor seguridad en cuanto a los posibles errores en ejecución. Es este el motivo por el que a lo largo de la obra le hemos repetido hasta la saciedad que si Lazarus ya implementa una determinada unidad, función o procedimiento, es recomendable no escribir otro para realizar la misma tarea. Todos los programadores que han intervenido en el proyecto Lazarus/Free Pascal tienen en conjunto una experiencia superior a la nuestra y son mejores profesionales. Además, cientos de miles de usuarios han colaborado probando una a una las distintas funcionalidades del programa, informando de los errores e incluso corrigiéndolos.

Sería muy simplista pensar que una primera ejecución de un programa nos dará todos los detalles de su funcionamiento. Es necesario someterlo al uso de muchos usuarios, llevar su funcionamiento al límite. A menudo, los errores en ejecución solo surgen bajo determinadas condiciones o en combinación con algunas circunstancias que se dan en las pruebas realizadas por los usuarios sobre la versión beta del software. Circunstancias que pueden no volverse a repetirse. Por ese motivo dijimos que con cierta frecuencia muchos de los fallos en ejecución no llegan a resolverse.

## 16.2 UNIDADES DE PRUEBA

La escritura de código para probar clases, rutinas y unidades debiera ser una etapa más del proceso de desarrollo de nuestro software. Poco a poco ha ido aumentando la popularidad de las distintas infraestructuras para pruebas de código. Java ya contaba con JUnit, Delphi con DUnit y Lazarus/Free Pascal incorporó en la versión 0.9.7 beta de su IDE el marco de trabajo **FPCUnit**. Cualquiera de estos marcos de prueba proporciona un entorno controlado para la realización de una batería de test que posibilite al programador aislar y corregir errores.

El corazón del sistema de pruebas **FPCUnit** se encuentra implementado en la unidad homónima. Contiene una serie de clases base que integran todo el sistema de pruebas:

- **`TTest`**. La clase padre para `TTestCase` y `TTestSuite`.
- **`EAssertionFailedError`**. Es la excepción básica. Indica que una prueba ha sido errónea.
- **`TAssert`**. Es la clase que implementa toda la batería de comandos de afirmación (`Assert`).
- **`TTestCase`**. Es la clase central de la unidad `fpcunit`. Para cada conjunto de pruebas se ha de implementar un descendiente de ella.
- **`TTestResult`**. Es una clase para ejecutar un test y almacenar los resultados de la prueba.
- **`TTestFailure`**. Es la clase que permite almacenar una prueba fallida.

De todas estas clases, las más importantes para el programador son, en realidad, `TTestCase` y `TAssert`. El resto de clases solo son necesarias para registrar las pruebas, ejecutarlas y comunicar los resultados.

### 16.2.1 Un ejemplo de prueba

Como hemos dicho, la clase central de FPCUnit es **`TTestCase`**, de la que debe escribirse un descendiente para cada batería de pruebas. Una prueba consiste en varios procedimientos que pueden ejecutarse de forma independiente. Cuando se ejecuten, los procedimientos se llamarán de forma automática y si se produce alguna excepción, el resultado se considerará fallido.

La única forma que tiene el entorno de pruebas de saber qué procedimientos debe llamar es a través de la información *RTTI* de la clase. Solo pueden ejecutarse aquellos que se encuentran definidos en la sección **`published`** de la clase. No pueden aceptar parámetros y no deben devolver resultados. Si la prueba falla, se lanzará una excepción.

Lazarus facilita dos estructuras listas para ejecutar pruebas en los programas, una para programas en modo consola y otra para aplicaciones gráficas. Seleccione **Proyecto | Nuevo Proyecto…** y elija la última opción de la ventana de diálogo **Crear un proyecto nuevo**, **FPCUnit Test Application** (Figura 16.1 A).

Por defecto, la nueva ventana de diálogo **TestCase Options** le muestra el proyecto de nombre `TTestCase1`. Cámbielo por `TWiFiTestCase` y pulse el botón **Create unit** (Figura 16.1 B).

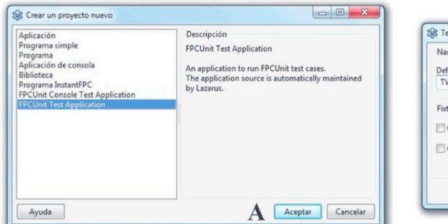


**Figura 16.1.** Creación de una unidad de pruebas con FPCUnit

En el menú **Proyecto** seleccione **Guardar proyecto como...** y dele el nombre `WiFiTest.lpi`, y a la unidad el nombre `wifitestcase.pas`. El esqueleto de código que Lazarus le muestra contiene las funcionalidades heredadas de la clase `TTestCase`:

```
type
  TWiFiTestCase = class(TTestCase)
  published
    procedure TestHookUp;
  end;
implementation
procedure TWiFiTestCase.TestHookUp;
begin
  Fail('Write your own test');
end;
```

Ahora hay que adaptar el método `TestHookUp` del descendiente de `TTestCase` para dar forma a las diferentes pruebas que ideemos.

Seleccione **Archivo | Nueva unidad** y guárdela como `wificases.pas`. Escriba en la sección de implementación de la unidad la función siguiente:

```
function AltaSeguridad():byte;
var
 letras : array[1..63] of char;
 i : integer;
 clave: string;
begin
 clave :='';
 randomize;
 for i:=1 to 63 do
  begin
    letras[i] := chr(32+random(94));  // ASCII 32 - 126
    clave += letras[i];
  end;
 Result := Length(clave);
end;
```

No se olvide de escribir su declaración en la interfaz de la unidad para poder exportarla. Regrese ahora a la unidad principal `WiFiTestCase` y pulse **[Alt] + [F11]** para abrir el cuadro de diálogo **Usar una unidad de este proyecto**. Seleccione el botón de radio **Interfaz** y haga doble clic sobre la única unidad que aparece: `wificases` (Figura 16.2). Lazarus añadirá automáticamente esta unidad a la cláusula **`uses`** de `WiFiTestCase`.

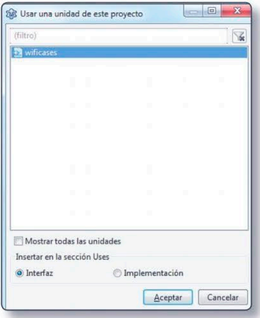



**Figura 16.2.** Creación de una unidad de pruebas con FPCUnit

Para completar el test tan solo necesitamos añadir las pruebas necesarias en el método `TestHookUp`. Sitúe el cursor en la palabra `TTestCase` de la declaración de clase y pulse **[Alt]** + ↑. Automáticamente, Lazarus saltará a su declaración. Observe que hereda de la clase **`TAssert`**. Esta posee cerca de cuarenta procedimientos

sobrecargados de nombres `AssertXXX`, que son los que pueden invocarse en las pruebas que escribamos.

Por ejemplo, podemos comprobar que, efectivamente, la función escrita `AltaSeguridad()` devuelve una cadena de 63 bytes, que es la longitud de una clave segura WPA. Así que nuestro test podría ser:

```
AssertEquals ('LongitudClaveLarga', 63, AltaSeguridad());
```

Donde el primer parámetro es una cadena que describe el objetivo de la prueba; el segundo, el valor esperado, 63 bytes; y el último, la llamada a la función que generará la clave.

Sitúe el cursor sobre el nombre del procedimiento `TestHookUp`, pulse **[F2]** y cambie su nombre a `WPALarga`. Reemplace ahora la línea

```
Fail('write your own test');
```

con:

```
AssertEquals ('LongitudClaveLarga', 63, AltaSeguridad());
```

Compile y ejecute el programa. Si todo ha ido bien, le aparecerá la ventana de diálogo de la aplicación **FPCUnit**. Cuando haga clic en el botón **Run**, Lazarus ejecutará la prueba y le informará de su resultado. Si el test se realiza con éxito, verá una serie de iconos de color verde, como en la figura 16.3.

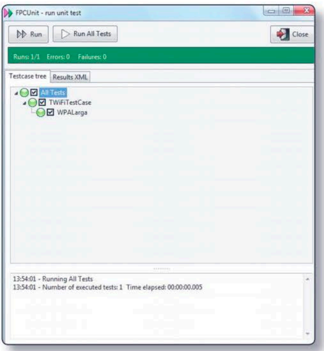



**Figura 16.3.** Ejecución exitosa de la prueba planificada

Si los iconos son violetas le indican que la salida obtenida y la esperada no coinciden, lo que lanzaría una excepción de tipo `EAssertionFailedError`.

La cuestión más importante es cómo saber qué tipo de pruebas programar. Evidentemente, no hay una respuesta que sea universalmente válida. Los test más asequibles son de tipo procedural. Al menos siempre es obligado examinar los valores extremos de las funciones, aunque es recomendable escribir tantas pruebas como se nos puedan ocurrir. Lógicamente, codificar estas pruebas es algo tedioso, pero muy recomendable. Sobre todo porque una vez escritas pueden ejecutarse rápidamente siempre que se altere la función original. Así mismo, aunque las pruebas se pasen con éxito, no garantizan que las funciones o procedimientos probados estén libres de errores, pero sí disminuyen las probabilidades de que ocurran.

## 16.3 MENSAJES DEL COMPILADOR

Algunos errores de codificación ocurren por ignorar las advertencias del compilador. Es importante tener presente que todas las notificaciones que el compilador nos muestra en la **Ventana de mensajes** merecen ser tenidas en cuenta, aunque algunas de ellas no sean útiles para localizar fuentes de posibles errores. Así, el mensaje

```
unit1.pas(8,42) Hint: Unit "math" not used in Unit1
```

probablemente no permita localizar errores, como todas las pistas `Hint` que nos proporciona el compilador de Free Pascal. Sin embargo, sí podemos reducir ligeramente el tamaño del código eliminando las declaraciones innecesarias. De este modo, también reducimos el número de mensajes que hay que revisar cada vez que se construya o compile el programa. En el caso del mensaje mostrado más arriba, hay varias maneras de eliminar la unidad no usada. Si pincha sobre el mensaje automáticamente, Lazarus le llevará a la línea 8, carácter 42 y podrá borrar la unidad `math` manualmente. También puede pulsar el botón derecho del ratón sobre el mensaje y elegir **Solución rápida**: **Eliminar unidad**. Lazarus eliminará la unidad `math` por usted.

**NOTA**
Si desea ver todos los mensajes lanzados por el compilador en el proceso de compilación, haga clic derecho sobre cualquier mensaje y elija **Copiar todos los mensajes mostrados y ocultos al portapapeles**. Luego puede pegarlos en cualquier editor de texto para estudiarlos.

Igualmente, puede hacer clic derecho en `Unit1` en el Editor y elegir **Refactorizar | Unidades sin usar…** Se abrirá el cuadro de diálogo homónimo con todas las unidades que realmente no sean necesarias, independientemente de los mensajes del compilador. La salida se mostrará en dos nodos en vista en árbol en la que se recogen de forma separada las unidades que no se emplean en las secciones **Interfaz** e **Implementación**. Aquí podrá elegir entre **Eliminar todas las unidades** o **borrar solo algunas** (Figura 16.4).

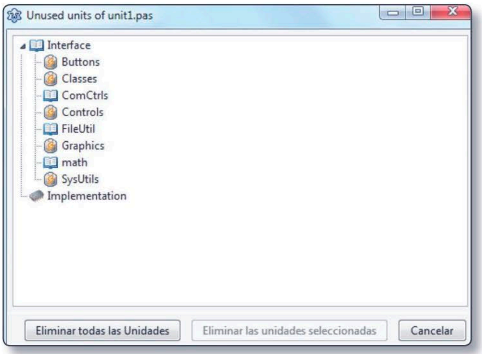



**Figura 16.4.** Cuadro de diálogo Unidades sin usar

Otro error de codificación demasiado habitual en los programadores noveles ocurre como consecuencia de olvidarse de declarar alguna variable. En estas situaciones, el compilador responde con un error de **identificador no encontrado**:

```
unit_main.pas(59,16) Error: Identifier not found "IVA"
```

Otras veces sí se declaran, pero se olvida **inicializarlas**. El compilador responde entonces con una advertencia (**`Warning`**):

```
fib.pas(35,10) Warning: Local variable "i" does not
seem to be initialized
```

Todo lo que debería hacerse sería añadir la declaración `i := valor inicial` inmediatamente antes de su utilización.

## 16.4 LAS ASERCIONES

En programación, una **aserción** es un predicado verdadero-falso que se incluye en un programa como indicación de que el programador está convencido de que dicha aseveración siempre debe cumplirse en ese punto del flujo de programa. Se trata de una herramienta de depuración muy útil, originaria de C y C++, para comprobar en tiempo de ejecución el valor de una variable.

El procedimiento **Assert** toma por argumentos una expresión booleana y una cadena opcional como mensaje:

```
Assert(Condición: Boolean; [ Mensaje: String ]);
```

Si la prueba lógica falla, el programa hace una parada lanzando una excepción EAssertionFailed, junto con el mensaje que identifica la línea de código que generó el fallo. Generalmente, las aserciones se emplean para detectar errores de programación y para comprobar asunciones que no siempre tienen que ser ciertas.

Las aserciones suelen codificarse en las primeras fases del desarrollo o prueba, pero pueden dejarse permanentemente en el código, pues el compilador de Free Pascal da un soporte completo a las mismas. Es posible generar el código necesario en el ejecutable y, una vez depurado, eliminar el código compilado de las aserciones mediante dos directivas globales de compilación:

```
{$ASSERTIONS ON/OFF}  (forma larga)
{$C +/-}  (forma corta)
```

Probablemente, como programador novel, es una herramienta que no va a emplear a menudo, sin embargo, las aserciones constituyen una manera elegante de comprobar un predicado lógico. Veamos con un ejemplo práctico cómo manejarlas.

Abra un nuevo proyecto en modo gráfico de nombre **asercion.lpi** con una unidad de formulario llamada unidad_ppal.pas. Añada la directiva ***{$ASSERTIONS ON}*** y arrastre un botón. Declare en la sección **type** de la unidad una clase TAlmacenar del siguiente modo:

```
TAlmacenar = class(TObject)
    FDatos: string;
    property Datos: string read FDatos write FDatos;
 end;
```

En la sección **implementation** defina un procedimiento de nombre `ModificarAlmacen` como sigue:

```
procedure ModificarAlmacen(AAlmacenar: TAlmacenar; const
                              s: string);
begin
  Assert(AAlmacenar <> Nil, 'Cadena no asignada.');
  AAlmacenar.Datos := s;
  ShowMessage( AAlmacenar.Datos );
end;
```

Y genere un manejador para el evento `OnClick` del botón así:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Almacenar: TAlmacenar;
begin
  Almacenar := TAlmacenar.Create;
  try
    ModificarAlmacen (Almacenar, 'Hola, mundo');
  finally
    Almacenar.Free;
  end;
  // Esta llamada fallará y lanzará una excepción
  ModificarAlmacen (Nil, 'Falla');
end;
```

Tras compilar y ejecutar el programa, pulse el botón y se abrirá una ventana con el mensaje `Hola, mundo`. Al hacer clic sobre **OK**, se lanzará una excepción `EAssertionFailed` (Figura 16.5).

Las líneas fundamentales del código son tres:

```
Assert(AAlmacenar <> Nil, 'Cadena no asignada.');
ModificarAlmacen (Almacenar, 'Hola, mundo');
ModificarAlmacen (Nil, 'Falla');
```

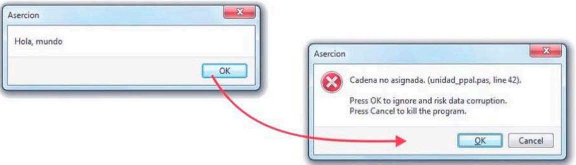



**Figura 16.5.** Excepción EAssertionFailed lanzada al fallar la prueba lógica establecida

En la primera línea definimos la aserción, que será cierta siempre que la variable se encuentre definida. Al pulsar el botón del formulario, se llama por primera vez al procedimiento `ModificarAlmacen` con la cadena `'Hola, mundo'`, que se muestra como mensaje en una ventana emergente. Al hacer clic sobre su botón, se liberan de memoria los recursos y se llama de nuevo al procedimiento con un puntero que no apunta a ningún sitio (**`Nil`**), lo que hace que falle la prueba lógica, al no estar definido, y se lance la excepción que se muestra en la figura anterior.

## 16.5 OBSERVADOR DE CÓDIGO

Lazarus posee una herramienta denominada **Observador de código**, por defecto inhabilitada, que puede ayudarnos a identificar procedimientos sobredimensionados y otras deficiencias habituales en los programas. Los largos procedimientos pueden significar una pobre encapsulación o una dependencia excesiva de una parte del código frente a otra. Cuanto más simples sean las rutinas y menor la dependencia de unas partes frente a otras, el código será más sencillo de estudiar y mantener.

El **Observador de código** se habilita desde el nodo **Categorías** de la rama *Explorador de código* del cuadro de diálogo **Opciones del IDE**, accesible a través de la combinación de teclas **[Ctrl] + [May] + [O]** (Figura 16.6).

A continuación vaya al nodo **Observador de código** para abrir la página en la que seleccionar los diferentes parámetros que quiera controlar. No tema elegir todos.

Como ya dijimos, resulta siempre muy interesante para mantener el código ordenar alfabéticamente los métodos y las variables, sobre todo en clases que contienen decenas de ellos. Una categoría también muy importante para evitar posibilidades de fallos es **Procedimientos grandes**, y 20 líneas son un buen indicador para empezar (Figura 16.7).

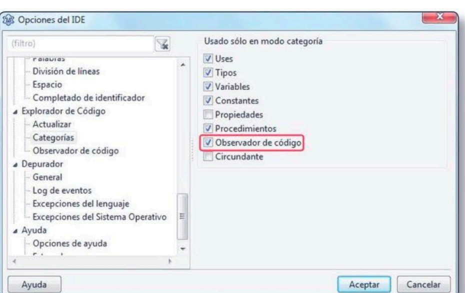



**Figura 16.6.** Hoja Categorías de Opciones del IDE

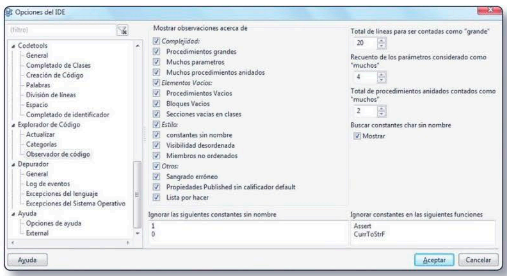



**Figura 16.7.** Página Observador de código de Opciones del IDE

Una vez establecidas las opciones que haya decidido, podrá ver las observaciones que le hace Lazarus acerca de un proyecto a través del menú **Ver | Explorador de código**.

Como ejemplo, abra el archivo de nombre **`addrbook.lpi`** que encontrará en la carpeta `address_book` y acceda al **Explorador de código** desde la unidad `frmmain`. En la vista en árbol, navegue hasta el nodo **Observador de código**. Verá una lista semejante a la de la figura 16.8.

Solo un método es especialmente largo. Si hace doble clic sobre su nombre, automáticamente saltará a su código fuente para que pueda examinarlo y decidir si debería refactorizarlo.

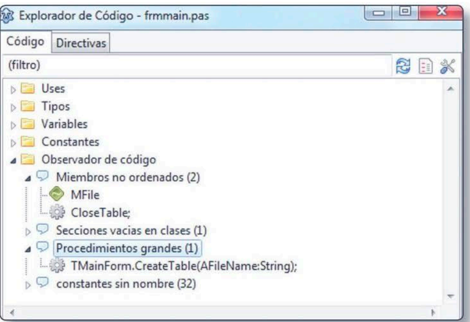



**Figura 16.8.** Observaciones de Lazarus sobre el código de un proyecto

Hay, así mismo, dos miembros no ordenados. Esto es muy fácil de resolver. Seleccione las líneas desordenadas y elija **Editar | Ordenar selección…** En la ventana de diálogo del mismo nombre, asegúrese de que en la sección **Dominio** está seleccionado el botón de radio **Líneas** y después haga clic en **Aceptar** (Figura 16.9). Automáticamente, Lazarus ordena los miembros o rutinas. Es posible que tras la ejecución deban ajustarse las sangrías.

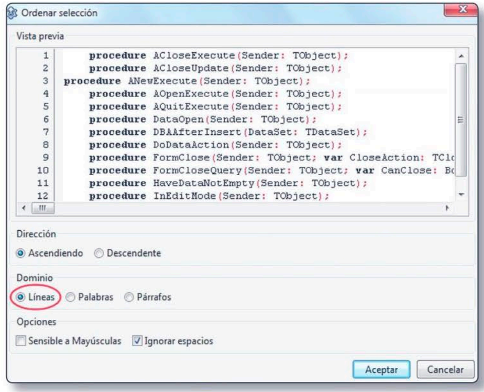



**Figura 16.9.** Ordenación alfabética de los procedimientos del ejemplo

También hay 32 constantes sin nombre. Si expande este nodo observará que muchas de ellas son numéricas. Si hace doble clic sobre alguna de ellas, saltará de nuevo a su código en el editor. La identificación de constantes sin nombre es muy útil si desea internacionalizar su aplicación traduciéndola a otros idiomas.

## 16.6 REFACTORIZACIÓN

En ingeniería del software, el término **refactorización** se usa para describir la reestructuración del código fuente, alterando su estructura interna sin cambiar su comportamiento externo, lo que se conoce en el argot por *limpiar el código*, para mejorar su consistencia interna y su claridad. Los test aseguran que la refactorización no cambia el comportamiento del código.

Debe tener en cuenta que la refactorización no arregla errores ni añade funcionalidad, tan solo intenta facilitar la comprensión del código y eliminar código muerto para mejorar su mantenimiento en el futuro.

Lazarus cuenta con un número importante de funciones para refactorizar el código, entre las que destacan **Renombrar identificador** y **Extraer procedimiento**. Ambas pueden ejecutarse desde el menú contextual **Refactorizar**. Para ver cómo funciona la refactorización, echemos un vistazo a la unidad `mainF.pas` del programa **`cifrador`** que estudiamos en el Capítulo 13 y que se encuentra en la carpeta BlowFish. Esta aplicación contiene un procedimiento llamado `btnCifrarClick` de tan solo 12 líneas, bastante corto para ser un primer candidato para refactorizar, pero suficiente para ilustrar el modo de trabajo.

Seleccione en el procedimiento `btnCifrarClick` las líneas resaltadas en gris en el código:

```
procedure TForm1.btnCifrarClick(Sender: TObject);
var bfCif: TBlowFishEncryptStream;
begin
  if mClaro.Lines.Count = 0 then Exit;
  clave:= edClave.Text;
  sOriginal:= TStringStream.Create(EmptyStr);
  bfCif:= TBlowFishEncryptStream.Create(clave, sOriginal);
  bfCif.WriteAnsiString(mClaro.Text);
  bfCif.Free;
  mCifrado.Text:= sOriginal.DataString;
  mClaro.Text:= '';
 end;
```

Haga clic derecho con el ratón y elija el menú contextual **Refactorizar | Extraer procedimiento…** En el cuadro de diálogo homónimo cambie el nombre del nuevo procedimiento de `NewProc` a `CifrarAlVuelo`, con la precaución de dejar el botón de radio de **Subprocedimiento** seleccionado (Figura 16.10).

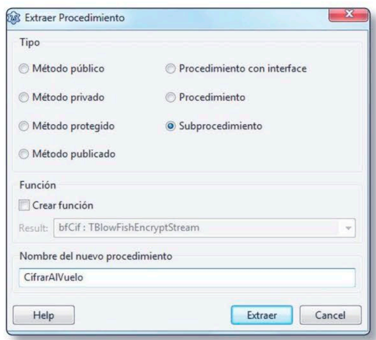



**Figura 16.10.** Cuadro de diálogo Extraer procedimiento del menú Refactorizar

Cuando haga clic sobre el botón **Extraer**, Lazarus insertará en el código del procedimiento `btnCifrarClick` un subprocedimiento, llamado `CifrarAlVuelo`. Ahora quedaría así:

```
procedure TForm1.btnCifrarClick(Sender: TObject);
var bfCif: TBlowFishEncryptStream;
  procedure CifrarAlVuelo;
  begin
    clave:= edClave.Text;
   sOriginal:= TStringStream.Create(EmptyStr);
   bfCif:= TBlowFishEncryptStream.Create(clave,sOriginal);
   bfCif.WriteAnsiString(mClaro.Text);
    bfCif.Free;
 end;
begin
  if mClaro.Lines.Count = 0 then Exit;
  CifrarAlVuelo;
  mCifrado.Text:= sOriginal.DataString;
  mClaro.Text:= '';
end;
```

En este caso concreto, la mejora, si existe, es mínima, pero le muestra cómo puede trabajar con la refactorización. Quizás con ganar algo de legibilidad ya merecería la pena, pues facilitará en un futuro el mantenimiento del código.

Otra opción bastante útil es **Renombrar identificador**, muy interesante cuando se decide cambiar el nombre de alguna variable o rutina. Basta con situar el cursor sobre el identificador deseado y pulsar **[F2]**. En el cuadro de diálogo **Buscar o Renombrar identificador** escriba el nuevo nombre, su alcance y pulse el botón **Renombrar todas las referencias** para que Lazarus realice el cambio.

## 16.7 SEGUIMIENTO DE LAS VARIABLES

A veces, una respuesta inesperada en la salida de una función es consecuencia de una variable errónea, por lo que es muy interesante disponer de procedimientos que permitan realizar un seguimiento exhaustivo del valor de las mismas en tiempo de ejecución. Evidentemente, como estas pruebas se efectúan solo durante el proceso de desarrollo, deben permanecer invisibles en la versión final, por lo que disponer de código que permita hacer este seguimiento, o inhabilitarlo a voluntad, es muy útil. En las siguientes secciones veremos algunas posibilidades para seguir el valor de las variables, funciones y propiedades allá donde se obtenga una salida inesperada.

### 16.7.1 La directiva *{$DEFINE DEBUG}*

El compilador de Free Pascal usa la directiva `{$DEFINE DEBUG}` para activar código de depuración encapsulado en un bloque `{$IFDEF ...}` … `{$ENDIF}`. Lazarus permite envolver el código seleccionado en el editor en las directivas seleccionadas mediante el cuadro de diálogo **Defines condicionales**, accesible mediante la combinación de teclas **[Ctrl] + [May] + [D]** (Figura 16.11).

`Debug` es una de las opciones del cuadro de diálogo, también accesible a través del menú **Fuente | Encerrar en $IFDEF…**

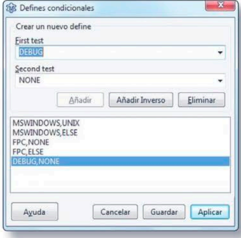



**Figura 16.11.** Cuadro de diálogo Defines condicionales

Suponga que ha escrito una función para invertir una cadena:

```
function InvertirCadena(const s: string): string;
var long, p: integer;
begin
  long:= Length(s);
  SetLength(Result, long);
  for p in [1..long] do
    Result[long - p] := s[p];
end;
```

Para depurarla, vamos a construir una aplicación gráfica de nombre **invertir_cadena** con un cuadro de edición y una etiqueta. El cuadro de edición poseerá un manejador del evento `OnChange`, de modo que la etiqueta recoja la cadena invertida según se escribe.

Para poder usar el procedimiento **DebugLn()**, debemos añadir a la cláusula **uses** la unidad **LazLogger**. El procedimiento `DebugLn` escribe en la salida estándar, de modo que en Windows debemos tener la precaución de incluir la directiva **{$*apptype console*}** para indicar que es una aplicación de consola, o no se verá la salida de depuración.

El fichero **invertir_cadena.lpr** sería así:

```
program invertir_cadena;
{$mode objfpc}{$H+}
{$IFDEF WINDOWS}
  {$apptype console}
{$ENDIF}
  uses Interfaces, Forms, main;
{$R *.res}
begin
  RequireDerivedFormResource := True;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Y la unidad principal, `main.pas` debe parecerse a esto:

```
unit main;
{$mode objfpc}{$H+}
{$DEFINE DEBUG}
interface
uses SysUtils, Forms, StdCtrls, LazLogger, Classes;
type
    { TForm1 }
    TForm1 = class(TForm)
      Edit1: TEdit;
      Label1: TLabel;
      procedure Edit1Change(Sender: TObject);
    private
      function InvertirCadena(const s: string): string;
    end;
var
Form1: TForm1;
implementation
{$R *.lfm}
procedure TForm1.Edit1Change(Sender: TObject);
begin
Label1.Caption:= InvertirCadena(Edit1.Text);
end;

function TForm1.InvertirCadena(const s: string):
string;
var
   long, p: integer;
begin
     long:= Length(s);
     SetLength(Result, long);
     for p in [1..long] do
       begin
      {$IFDEF DEBUG}
      DebugLn(['longitud: ',long,'; p: ',p,'; longitud -
                                 p: ',long-p]);
      {$ENDIF}
      Result[long - p] := s[p];
      end;
end;
end.
```

Cuando compile y ejecute el programa verá por consola una salida así,

```
longitud: 1; p: 1; longitud - p: 0
```

justo antes de que el programa lance una excepción como consecuencia de que `[long - p]` adopta como valor 0, que es un índice inválido. Así pues, debemos modificar la línea

```
Result [long - p] := s[p];
```

del siguiente modo:

```
Result [long - p + 1] := s[p];
```

Si ahora vuelve a comprobar el funcionamiento, observará que ya no se produce ninguna excepción con los índices, así que podemos desactivar la directiva **{$DEFINE DEBUG}**, lo que puede hacerse de dos modos:

```
{ $DEFINE DEBUG} // dejando espacio entre { y $
{$UNDEF debug}  // desactivando la directiva
```

Si se necesita volver al código con alguna operación de mantenimiento, el soporte de depuración se encuentra ya presente y tan solo habría que volver a activar la directiva.

### 16.7.2 Funciones de depuración

La unidad `LazLogger` provee un gran número de funciones para mostrar cadenas en la salida estándar junto a una aplicación gráfica. Recuerde que en Windows se requiere siempre la directiva **{$apptype console}**.

Todas las funciones definidas en la unidad están sobrecargadas, lo que significa que `DebugLn()` puede aceptar una gran cantidad de parámetros:

- Hasta 18 cadenas separadas por comas:
  `DebugLn('uno','dos',...);`

- Un arreglo de constantes:
  `DebugLn(['varUno= ', varUno,', varDos= ', varDos]);`

- O una cadena formateada, entre otros:
  `DebugLn('intVar= %d, strgVar= %s',[intVar, strVar]);`

Hay, así mismo, un gran número de funciones de conversión de cadenas para transformar tipos especiales en cadenas que permitan mostrar su representación en la salida estándar. Tres de las funciones disponibles son:

- **`DbgStr()`**. Devuelve una cadena normal, con la excepción de que si dispone de caracteres no imprimibles, estos se convertirán a su formato hexadecimal `#nnn`.
- **`DbgS()`**. Convierte tipos ordinales en cadenas. Así mismo, transforma punteros y reales tipo `extended` para su impresión por pantalla.
- **`DbgSName()`**. Devuelve el nombre del componente o el nombre de la clase.

Aprovechando la función `InvertirCadena()`, podríamos ahora depurarla así:

```
function TForm1.InvertirCadena(const s: string):
string;
var long, p: integer;
begin
  {$IFDEF debug}
  DebugLnEnter ('Parametro recibido "%s"', [s]);
  {$ENDIF}
  long:= Length(s);
  SetLength(Result, long);
  for p in [1..long] do
  begin
   {$IFDEF debug}
   DebugLn(['longitud: ',long,'; p: ',p,'; longitud - p:
                ', long-p]);
   {$ENDIF}
   Result[long + 1 - p] := s[p];
  end;
  {$IFDEF debug}
  DebugLnExit('Saliendo de InvertirCadena, valor final de p
                  %d', [p]);
  {$ENDIF}
end;
```

De modo que al ejecutar la aplicación, obtendríamos la salida mostrada en la figura siguiente:

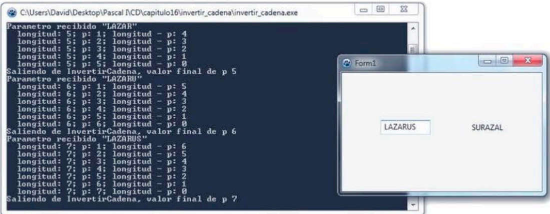



**Figura 16.12.** Salida de depuración de la función InvertirCadena

Hasta ahora solo habíamos hablado de la función `DebugLn`, pero aquí hemos hecho uso de otras dos funciones relacionadas: `DebugLnEnter` y `DebugLnExit`. El comportamiento de ambas es idéntico al de `DebugLn`, con la salvedad de que `DebugLnEnter` sangra automáticamente la línea siguiente que se escribe en la salida estándar, mientras que `DebugLnExit` no.

### 16.7.3 Interrupciones

En algunas situaciones es muy útil interrumpir en mitad de una rutina el flujo de un programa para revisar una determinada salida. Esto suele hacerse de una forma muy rápida, aunque poco formal, con la orden `ShowMessage()`. Mientras el cuadro de diálogo esté abierto, el flujo del programa queda detenido. Una vez depurada la rutina, basta con comentar o eliminar la línea. No se olvide de añadir la unidad `Dialogs` en la cláusula **uses** si piensa usar el procedimiento `ShowMessage`.

Por ejemplo, en nuestra manida función `InvertirCadena`, podríamos haber optado por añadir una interrupción así:

```
function TForm1.InvertirCadena(const s: string):
string;
var long, p: integer;
begin
  long:= Length(s);
  SetLength(Result, long);
  for p in [1..long] do
  begin
  ShowMessageFmt('[long - p] = %d', [long - p]);
```

```
    Result[long - p] := s[p];
    end;
end;
```

Al ejecutar el programa, se provocará una interrupción para mostrar un cuadro de diálogo con el valor `[long - p] = 0`, justo antes de que se lance una excepción como consecuencia del índice erróneo.

### 16.7.4 El servidor de depuración

Lazarus se descarga e instala con una herramienta muy interesante que se encuentra, en Windows, en el directorio `\lazarus\tools\debugserver` lista para construirse. Tan solo debe abrir el proyecto **`debugserver.lpi`** y compilarlo.

Abra el proyecto `invertircadena.lpi` y añada en su cláusula **`uses dbugintf`**. Esta es una unidad completamente transparente en su uso y proporciona once procedimientos `SendXXX` que funcionan de modo semejante a la función `DebugLn`. No obstante, no manda los mensajes de depuración a la salida estándar, sino al servidor de depuración, que los muestra en pantalla en su propia aplicación visual. Es realmente útil para rastrear secuencias de acontecimientos, pues el servidor muestra el tiempo en que ocurre cada mensaje de depuración recibido. La aplicación, además, permite almacenar un fichero de registro.

Una vez que haya puesto la unidad `dbugintf` en la cláusula **`uses`** del programa que va a depurar, ejecute el servidor de depuración, `debugserver.exe` en Windows, y, en su menú **View**, seleccione **Always on top**. Modifique entonces la función `InvertirCadena()` como sigue:

```
function TForm1.InvertirCadena(const s: string):
string;
var
  long, p: integer;
begin
  SendMethodEnter ('InvertirCadena');
  long:= Length(s);
  SetLength(Result, long);
  for p in [1..long] do
    Result[long + 1 - p] := s[p];
    SendDebugFmt ('El resultado es "%s"',[Result]);
end;
```

Compile y ejecute la aplicación. Observe cómo ahora la información de depuración aparece en la ventana del servidor (Figura 16.13).

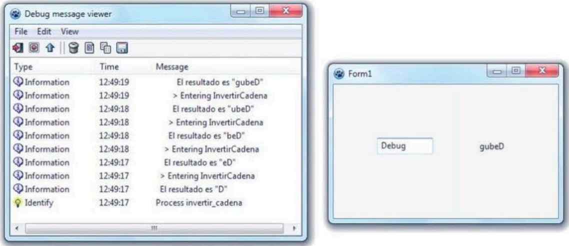



**Figura 16.13.** Salidas de depuración en la ventana del servidor

## 16.8 LA UNIDAD *HEAPTRC*

El cuadro de diálogo **Opciones para el proyecto** (**[Ctrl]** + **[May]** + **[F11]**) incluye varias alternativas relacionadas con la depuración de las aplicaciones. Entre ellas cabe mencionar la unidad **`Heaptrc`**, que se recoge como una casilla de verificación en la sección **Depurando información** de la rama *Opciones del compilador* (Figura 16.14).

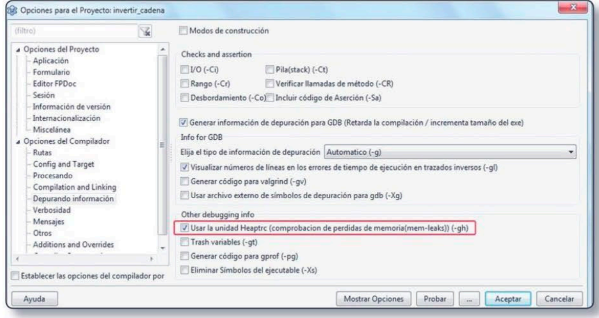



**Figura 16.14.** Selección de la unidad Heaptrc para comprobar pérdidas de memoria

Se trata de una herramienta vital para los desarrolladores, ya que permite identificar pérdidas de memoria en una aplicación. Las pérdidas se producen, por ejemplo, como consecuencia de un código erróneo que no libera la memoria previamente reservada; también apunta a errores de lógica en el programa, o a una pobre manipulación de punteros, por ejemplo. Este es el motivo por el que resulta tan interesante activar la unidad `heaptrc` para identificar estos errores.

La unidad redacta un informe –en una consola adicional en un programa en modo consola, o en un cuadro de diálogo aparte en los programas en entorno gráfico– cuando finaliza el programa. Si no se encuentran errores de pérdidas de memoria, el informe incluirá la declaración:

```
0 unfreed memory blocks: 0
```

Como alternativa, se puede volcar la información en un fichero, lo que suele ser recomendable cuando existen varios errores de pérdida de memoria. Para ello, es suficiente con añadirle al programa `.lpr` principal la siguiente línea:

```
SetHeapTraceOutput('/directorio/heaptrace.trc');
```

Veamos un ejemplo práctico muy sencillo en el que vamos a provocar intencionadamente un error de pérdida de memoria al no liberar un recurso previamente reservado. Abra un nuevo proyecto en modo gráfico de nombre **`perdida_memoria`** y añada la unidad **`StdCtrls`** en la cláusula **`uses`**. Seleccione ahora la casilla de verificación **Usar la unidad Heaptrc**, como mostramos más arriba, y genere un manejador del evento `OnCreate` para el formulario. Complete el procedimiento de la siguiente forma:

```
procedure TForm1.FormCreate(Sender: TObject);
var lbl: TLabel;
begin
  lbl:= TLabel.Create(nil);
  lbl.Caption:= 'Esto es una etiqueta dinámica';
  lbl.Parent:= Self;
end;
```

Ejecute el programa. Observará una ventana con una etiqueta de leyenda `Esto es una etiqueta dinámica` alineada en la parte superior izquierda del formulario. Cuando lo cierre, se abrirá una docena de cuadros de diálogo, de los que el primero será como el mostrado en la figura siguiente:

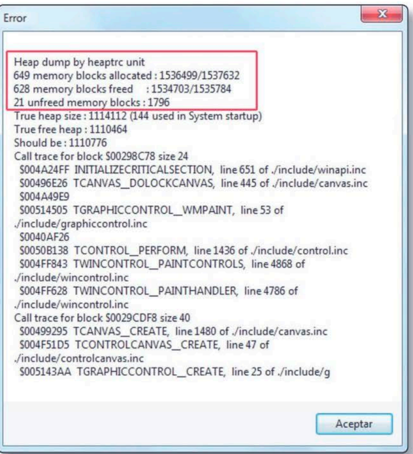



**Figura 16.15.** Resultado de la salida de la unidad Heaptrc

El problema de pérdida de memoria se produce porque se ha creado un componente `lbl` de tipo `TLabel` al generar el formulario, pero nunca se ha liberado, lo que conlleva dejar huérfanos 21 bloques con 1.796 bytes de memoria al finalizar el programa.

Pruebe ahora a ejecutar otra vez el programa añadiendo la llamada `SetHeapTraceOutput ('d:\heaptrace.trc')` en el archivo `perdida_memoria.lpr`. Una vez finalizada la ejecución, cuando cierre el programa, se habrá generado el archivo `d:\heaptrace.trc`, que podrá examinar desde el menú **Herramientas | Leak View**.

## 16.9 EL DEPURADOR GDB

**Gdb** es un potente depurador de código que se instala junto a Lazarus como herramienta independiente. Aunque está diseñado específicamente para C y C++ y no para Pascal orientado a objetos, muestra un comportamiento bastante aceptable para los programadores de Lazarus. Gdb está en continuo desarrollo y cambia tan rápidamente como lo hace Lazarus.

Aunque se dan algunas desventajas al usar esta herramienta orientada a C para depurar ejecutables escritos en Pascal, le será muy útil y podrá, como programador en Lazarus, seguir el valor de las variables, ver salidas de depuración, establecer puntos de ruptura y navegar línea a línea por el código. Su nivel de integración con Lazarus, considerando que es un proyecto independiente, es muy aceptable.

En el cuadro de diálogo **Opciones para el proyecto**, en el nodo **Opciones del compilador**, seleccione la hoja **Depurando información** y marque la casilla **Generar información de depuración para GDB** (Figura 16.14). Cerciórese, así mismo, de que la última casilla, **Eliminar símbolos del ejecutable (-Xs)**, se encuentra deshabilitada.

Vaya a la hoja **Compilación y enlazado** y desactive, si no lo está ya, la casilla **Enlazado inteligente (-XX)**. Elija, además, un nivel de optimización 0 o 1 y acepte los cambios.

El depurador dispone de varias ventanas, de las que las más útiles para los programadores noveles son **Lista de puntos de observación** y **Locales**, a las que se puede acceder desde el menú **Ver | Ventanas de depuración** o mediante la combinación de teclas **[Ctrl] + [Alt] +[W]** y **[Ctrl] + [Alt] +[L]**, respectivamente.

Para ejecutar un proyecto bajo el depurador, generalmente se establece primero un **punto de interrupción** o se usa la opción **Ejecutar | Ejecutar hasta el cursor**. Para establecer un **punto de interrupción**, haga clic sobre el número de línea en el editor de código o pulse **[F5]**. Lazarus le mostrará un icono rojo junto al número de línea y le resaltará la misma en rojo (Figura 16.16).

```
  .   function TForm1.InvertirCadena(const s: string): string;
  .   var long, p: integer;
  .   begin
 40       long:= Length(s);
  .       SetLength(Result, long);
 42       for p in [1..long] do
  .         Result[long - p+1] := s[p];
  .   end;
 45
  .   end.
 47
```

**Figura 16.16.** Punto de interrupción en la línea 42 del código fuente

Si regresamos a nuestra función original `InvertirCadena` y establecemos el punto de interrupción justo donde comienza el bucle **`for`**, la ejecución se parará en esa línea cuando ejecutemos el programa (**[F9]**). Ese es el momento de abrir los

cuadros de diálogo **Locales**, **Lista de puntos de observación** y **Pila de llamadas** (**[Ctrl]** + **[Alt]** + **[S]**). Añada a la lista de puntos de observación las expresiones `long` y `Result[long-p+1]`. Si ahora se mueve paso a paso por el código de la función pulsando **[F8]**, verá cómo los valores de las expresiones irán cambiando (Figura 16.17).

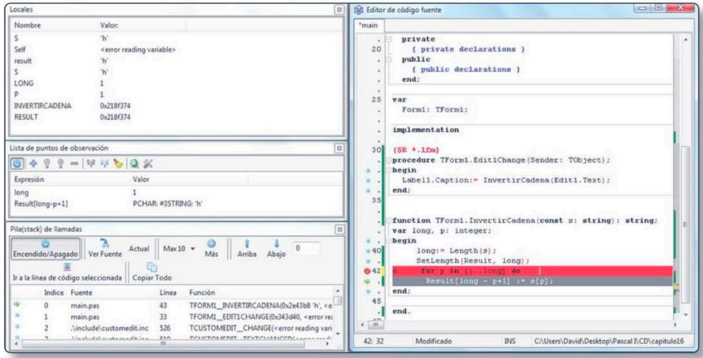
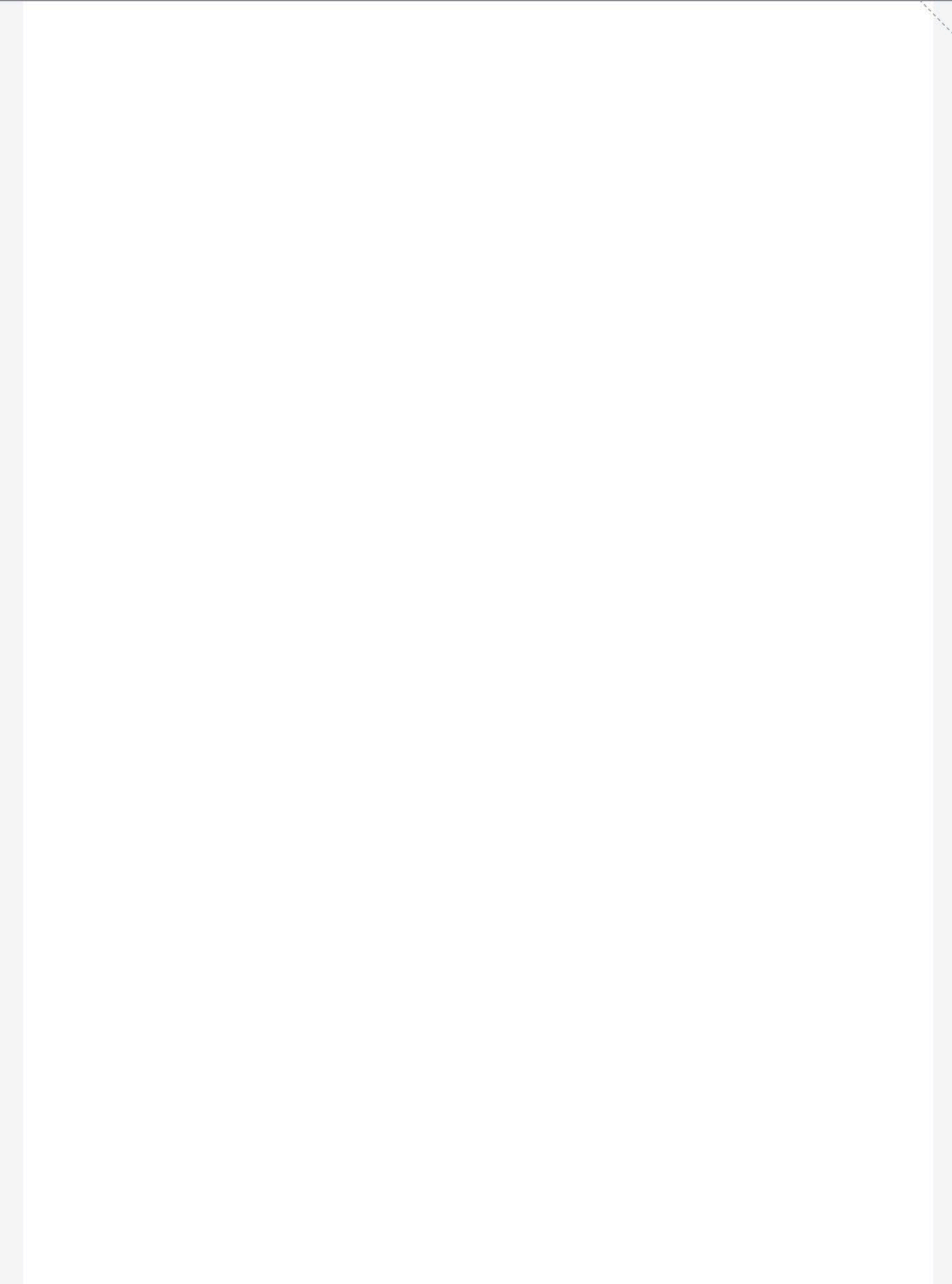



**Figura 16.17.** Ventanas de depuración en una ejecución paso a paso tras un punto de interrupción



# MATERIAL ADICIONAL

El material adicional de este libro puede descargarlo en nuestro portal web: *http://www.ra-ma.es*.

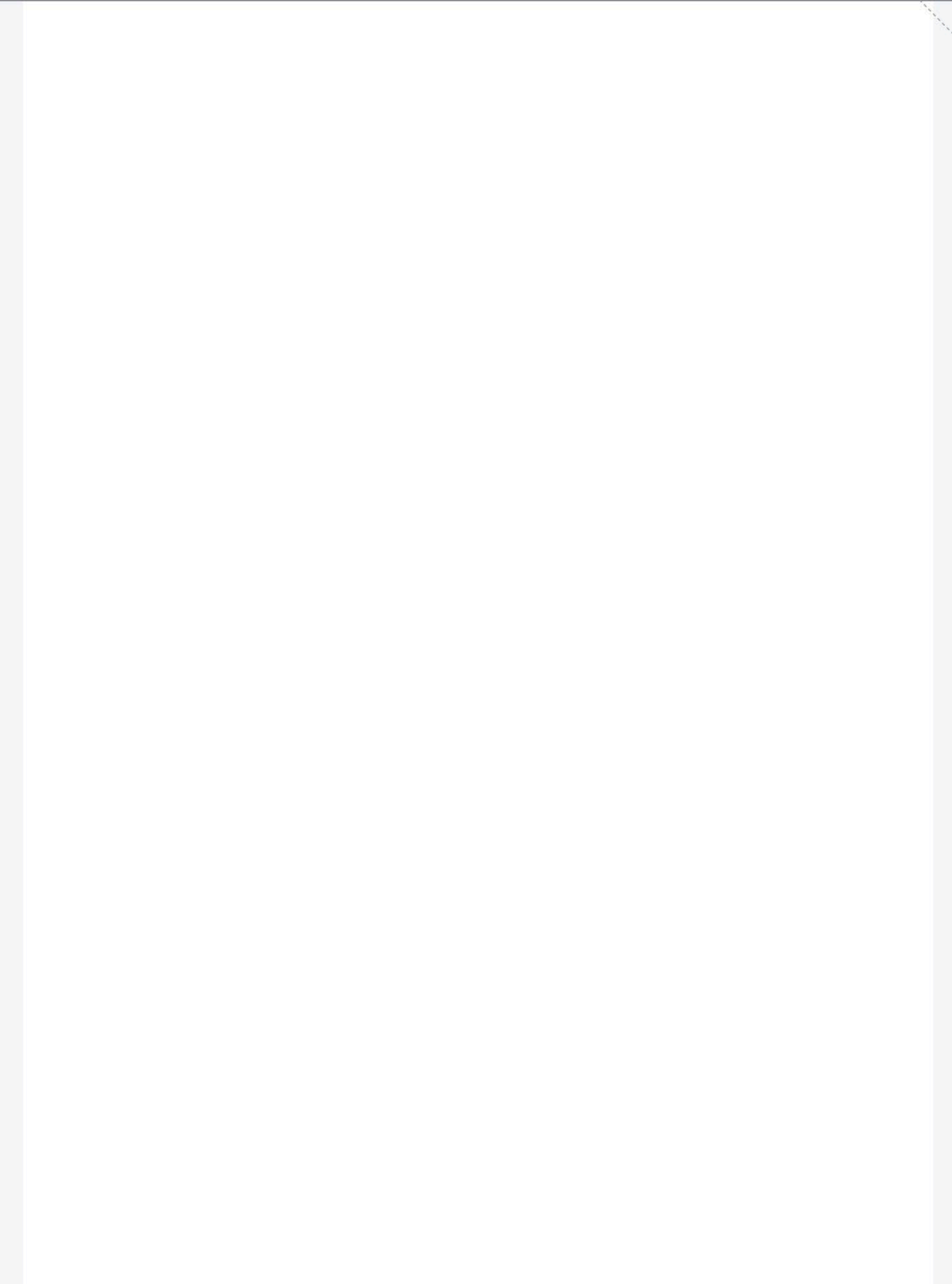
Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

Podrá localizar el número de ISBN en la página IV (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

## INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.



# ÍNDICE ALFABÉTICO


**Símbolos**

&, 168
<, 78
<=, 78
<>, 78
=, 78
>, 78
>=, 78
$apptype, 311
$ASSERTIONS, 303
$C, 303
$ENDIF, 36
$H–, 36
$H+, 36
$IFDEF, 36
$UNITPATH, 130

**A**

Abstract, 143
Acciones, 195, 207, 209, 210
ActionLink, 208
AddStrings, 228
Advertencias, 302
Aleatorio, acceso, 244
Alignment, 168
Alto nivel, lenguaje, 20
Anagramas, 282
Anchors, 197
And, 76
AnsiChar, 42
AnsiString, 42, 61
Append, 249
Application.ProcessMessages, 285, 289
Argumentos, 105, 109
Aritméticos, operadores, 74
Array, 58
Arreglo, 57
    Dinámico, 60
    Estático, 58
ASCII, código, 18
Aserciones, 303
Asignación
    Operador de, 48
    Sentencias de, 49
Asintentes, 201
Assert, 303
Assign, 218, 224
AssignFile, 69, 247
Autocheck, 208
AutoComplete, 190
AutoSize, 159, 168

**B**

BackPages, 202
Backup, carpeta, 32

Bajo nivel, lenguaje, 19
Beep, 175
begin, 23, 32
biMaximize, 159
biMinimize, 159
Binario
  Operador, 73
  Sistema, 18
Binarios, archivos, 255
  Apertura, 256
  Cierre, 257
  En la POO, 269
  Lectura/Escritura, 256
Bisel, 171
Bit, 17
BlockRead, 70, 263
BlockWrite, 70, 263
Blowfish, 240, 241
Boolean, 41
Booleano, tipo, 40
BorderIcons, 159
BorderStyle, 170
Borland, 21
BottomLine, 171
Box, 171
Break, 94, 95
Bucle, 86
Bug, 295
Byte, 39
ByteBool, 41

C

Cabecera, 32, 105, 109
Cadena, tipo, 42, 61
CamelCase, tipografÌa, 35
Campo, indicador, 64
Caption, 158, 168
Carácter, tipo, 41
Case of, 84
Category, 208, 209
Char, 41
CharCase, 175
Chr, 45, 77
Circunflejo, acento, 51
Clase, 115, 116
Class, 132
Classes, unidad, 33
ClassName, 137
ClassParent, 137
C, lenguaje, 23, 32
C++, lenguaje, 23
Close, 70
CloseFile, 250
Cláusula, 32
Colecciones, 218
Colores, 34
Columns, 184
Comentarios, 34
CommaText, 227
Compilación, 20, 21
Compilador
  Directivas, 34
Compilador, directivas, 36
Componente, 161
  Buscador, 161, 162
Conjuntos, 67
  Operaciones, 68
Consola, modo, 26, 28
Const, 37, 50, 107
Constantes, 37, 50
Continue, 96
Controles, 165
CopyFrom, 232
Count, 223
Create, 119
Cthreads, unidad, 33
Cuadros combinados, 190
Código, completar, 153
Cuerpo, 32
Currency, tipo, 40
Cursor, 204

D

DateToStr, 45
DbgS, 314
DbgSName, 314

DbgStr, 314
Dbugintf, unidad, 316
Debug, 310, 312
DebugLn, 311, 313
DebugLnEnter, 314, 315
DebugLnExit, 314, 315
Declaración, buscar, 152
DefaultExt, 276
DelimitedText, 227
Delphi, 21
Descendiente, objeto, 132
Destroy, 119
DialogKind, 276
Dialogs
  Pestaña, 275
  Unidad, 315
Dinámico, método, 142
Directivas
  Condicionales, 36
  Conmutadores, 36
  Parámetros, 36
División, 74
Do, 66
DoMenuClick, 206
Doublé, tipo, 40
Downto, 87
DropdownMenu, 207
DupeString, 65
Dynamic, 142

E

EAssertionFailed, 303, 304
EAssertionFailedError, 297
EchoMode, 175
EditMask, 176
Editor, 26, 27, 147, 151
  Navegación rápida, 152
EInOutError, 252
EIntOverflow, 280
Embarcadero, 21
Enabled, 166
Encapsulación, 25, 115
Encapsulado, 120, 124
End, 23, 32
Enlazado, 21
Ensamblador, lenguaje, 19
Entero, tipo, 38
Enumerado, tipo, 43
EOF, 70, 244
EOLN, 248, 249
Erase, 260
Escape, 41
Estado, barra de, 171
Eventos, 167
Excepciones, 97
  Clases, 100
Except, 98, 99
Exception, 98
Execute, 276, 291
Exit, 111
Exponenciación, 74
Expresión, 73
ExtendedSelect, 188
Extended, tipo, 40
External, clase, 118

F

Factorial, 87, 277
Fibonacci, sucesión, 112
Fichero, 69, 243
  Binario, 69
  De texto, 69,70
  Operaciones, 69, 245
  Sin tipo, 262
  Tipos de, 246
FileCreate, 231
FileMode, 258
FileMode, variable, 69
FileName, 276
FilePos, 259
FileSize, 259
File, tipo, 246
FileUtil, unidad, 269
Filter, 276
finally, 98, 99
Flat, 204

Flujo, 98
fmOpenRead, 231
fmOpenReadWrite, 231
fmOpenWrite, 231
fmShareCompat, 231
fmShareDenyNone, 231
fmShareDenyRead, 231
fmShareDenyWrite, 231
fmShareExclusive, 231
Foco, 174
FocusControl, 168, 169
Focused, 174
For, 86
For in, 88
Format, 72
FormCreate, 199
Formulario, 26, 27
FPCUnit, 296
Frame, 171
Frecuency, 191
Free, 120
FreeOnTerminate, 293, 294
Free Pascal, 20, 24
Funciones, 103, 104
Function, 104

## G

Gdb, depurador, 319
Gráfico, modo, 26, 28
Grids, 220
GUI, 145

## H

Hardware, 17
Heaptrc, unidad, 317, 318
Height, 164
Herencia, 25, 115, 131, 133
Hide, 166
Hilo, 277, 289, 291
Hint, 207, 301

## I

Identificador
 Completado, 153
 Renombrar, 310
If, 82
ImageIndex, 205
Images, 204
Implementation, 128
In, 68
Inc, 53
IndexOf, 223, 224, 228
IndexOfName, 224
Índice, 58
Inherited, 126, 133, 141
Instancia, 116
Int, 45
Int64, 39
Integer, 39
Interface, 127
Interpretación, 20
Interrupciones, 315
Interrupción, punto de, 320
IntToStr, 45
ItemIndex, 186, 187
Items, 183, 187

## J

Java, lenguaje, 23, 32

## L

Layout, 168
Lazarus, 21, 22, 24
 IDE, 25
LazLogger, unidad, 311, 313
LCL, 145, 146
Lectura/Escritura, puntero, 244
Left, 164
LeftLine, 171
Lib, carpeta, 32
LIFO, acceso, 119
LineSize, 191
Listas, 187, 223
LoadFromFile, 223, 266

Longbool, 41
Longint, 39
Longword, 39
LowerCase, 62, 175
Lógicos, operadores, 75

**M**

Máquina, código, 18, 19
Máscaras, 176, 177
  Editor de, 177
Math, unidad, 108
Matriz, 58
Max, 191
MaxLength, 174
MaxValue, 178
MD5, 264, 265
MD5File, 265
MD5Print, 265
Memoria, dirección, 50
Mensajes, ventana de, 26
MessageDlg, 281
Método, 116
Min, 191
MinValue, 178
Monohilo, 289
Módulo, 74, 103
Multihilo, 291
MultiSelect, 187
Multitarea, 290

**N**

Name, 166
NET, 145
Nil, 53, 61
Nombres, 35
Not, 79

**O**

Object Pascal, 25
Objeto, 115, 116, 117
Objeto, módulo, 21
Objetos, inspector de, 158
Observador de Código, 305
OnAcceptFileName, 233, 238
OnActivate, 182
OnChange, 180, 185
OnClick, 167
OnCloseUp, 190
OnCreate, 199
OnDestroy, 222
OnExecute, 208
OnKeyPress, 174
OnMouseMove, 172
OnSelect, 190
OnUpdate, 210
Operador, 73
  Precedencia, 78
  Sobrecarga, 73
Or, 76
Ord, 45, 77
Ordinal, tipo, 39
Orientation, 192
Out, 107
Overload, 108, 142
Override, 126, 127, 138, 141

**P**

PageCount, 199
Pages, 199
PageSize, 191
Paleta, 26, 27, 155, 160
Panels, 172
Paquete, 151
Parent, 164
Parámetros, 106
Pascal, 20
Password, 175
PasswordChar, 175
PerformTab, 174
Permanencia, 218
Permutaciones, 282
Pila, 280
Polimorfismo, 25, 115, 131, 138, 139, 143
POO, 115
Position, 191, 229

Private, directiva, 121
Procedimiento, 65
  Declaración, 109
  Definición, 108
Procedimientos, 103, 108
Procedure, 65, 108
Producto, 74
Program, 32
Programación, lenguajes, 19
Programa, estructura, 32
Property, 124
Propiedades, encapsulado, 124
Protected, directiva, 121
Proyecto, inspector de, 150
Public, directiva, 121
Puntero, 50
  General, 51
  Tipado, 51

**R**

Radio, botones de, 183
Raise, 98
Rangos, 191
Read, 70, 124, 249
ReadBuffer, 229, 269
ReadLn, 249
ReadOnly, 175
Real, tipo, 40
Record, unidad, 63
Recursividad, 112, 278
Refactorización, 308
Refactorizar, men˙, 302, 308
Registro, fichero, 243
Registro, tipo, 63
  Campos, 63
Relacionales, operadores, 78
RemoveFocus, 174
Rename, 260
Repeat, 92
Repetitivas, estructuras, 86
Reset, 70, 248, 254, 256
Resta, 74
Restringido, pestaña, 159
Result, 104
Retroceso, 41
ReverseString, 62
ReWrite, 70, 248, 256
RightLine, 171
Round, 45
RTL, 145
Rutinas, 103

**S**

SaveToFile, 223, 266
Secuencial, acceso, 244
Seek, 256, 259
Selección, estructuras, 82
SelectNext, 174
SelectNextPage, 198
SelEnd, 191
Self, 136, 137
SelStart, 191
SendDebugFmt, 316
Sender, 207
SendMethodEnter, 316
SetFocus, 174
SetHeapTraceOutput, 318, 319
SetLength, 42
Set of, 67
Shape, 171
ShellExecute, 203
Shl, 76, 77
Shortint, 39
ShortString, 61
ShortString, 42
ShowAccelChar, 168, 169
ShowHint, 207
Shr, 76, 77
SimplePanel, 172
Single, tipo, 40
Size, 229
sLineBreak, 55
Sobrecarga, 108
Software, 17
Sort, 228
Spacer, 171

State, 183
StdCtrls, unidad, 318
Step, 193
StepIt, 193
Stile, 171
Streaming, 228
String, 42, 61
StrToInt, 45
Strutils, unidad, 62
Style, 190, 204
Subclase, 132
Subrango, tipo, 44
Suma, 74
Superclase, 132
Suspended, 291
Synchronize, 291, 294
System, unidad, 33
Sysutils, unidad, 62

**T**

TabOrder, 169, 174
TabStop, 169, 174
Tabulador, 41
TActionList, 207, 208
TApplication, 215
TAssert, 297, 299
TBevel, 171
TBlowFishDeCryptStream, 240
TBlowFishEncryptStream, 240
TCheckBox, 162, 183
TCollection, 218, 219
TCollectionItem, 218, 219
TComboBox, 190
TCom ponent, 164
TComponent, 161, 165, 218
TCompressionStream, 236
TControl, 163, 164
TCustomMemoryStream, 230
TDecompressionStream, 236
TDividerBevel, 171
TEdit, 174
Terminate, 293
Text, 174
TextFile, 71, 248
Texto, archivos de, 248
  Apertura, 248
  Cierre, 250
  En la POO, 266
  Lectura/Escritura, 249
TFileNameEdit, 232
TFileStream, 230, 269
TFloatSpinEdit, 178
TGroupBox, 183
THandleStream, 230
TickMarks, 192
TImage, 199
TImageList, 197, 204
Tipos, 37, 38
  Cambio de, 44
  Forzado de, 44, 46
TLabel, 168
TLabeledEdit, 174
Tlist, 224
TList, 223
TlistBox, 187
TMainMenu, 212
TMaskEdit, 176, 177
TMemo, 200
TMemoryStream, 230
TObject, 133, 137
Top, 164
TOpenDialog, 276
TopLine, 171
TPageControl, 196
TPersistent, 217
TPopupMenu, 204, 207
TProgressBar, 193
TRadioButton, 183
TRadioGroup, 183
Traducción, 21
Transparent, 169
Trunc, 45
Truncate, 261
Try, 98, 99
TSaveDialog, 276
TSpinEdit, 178, 279

TStaticText, 170
TStatusBar, 171
TStream, 228
TStringList, 223, 224, 266
TStrings, 218, 223
TStringStream, 230
TTabControl, 196
TTabSheet, 196, 197
TTest, 297
TTestCase, 297, 298
TTestResult, 297
TThread, 291
TTIGrid, 219
TTogleBox, 175, 176
TToolBar, 204, 205
TToolButton, 204
TTrackBar, 191
TWinControl, 164
Type, 37

**U**

Unario, operador, 73
Unidad, 32, 127
Unit, 127
UnitName, 137
Until, 93
UpDown, 199
UpperCase, 62, 175
Uses, 32, 130
UTF-8, 62, 63, 274
UTF8ToSys, 274
UTF-16, 42

**V**

Val, 45, 77
ValueFromIndex, 225
Values, 224
Var, 37, 106
Variables, 37, 47
Variant, 45
Vector, 58
Verificación, casilla de, 162, 183
Virtual, 138, 141, 142
Visible, 166

**W**

While, 90
WideChar, 42
Width, 164
Wirth, Niklaus, 20
With, 66
Word, 39
WordBool, 41
WordWrap, 169
Write, 70, 124, 249
WriteBuffer, 229, 269
WriteLn, 23, 31, 249

**X**

xor, 76

**Z**

ZStream, 236

# Aprenda a programar con
# Lazarus

Lazarus es un entorno de desarrollo integrado basado en una extensión orientada a objetos del lenguaje de programación Pascal. Pascal es un lenguaje de programación de propósito general y de alto nivel desarrollado por el profesor suizo Niklaus Wirth entre los años 1968 y 1969. Su objetivo era crear un lenguaje que facilitara el aprendizaje de programación a sus alumnos y que con el tiempo se ha convertido en una herramienta fabulosa para la creación de aplicaciones de todo tipo.

Mientras Borland introducía la programación orientada a objetos en Pascal con su compilador de Turbo Pascal 5.5 (que terminó dando lugar a Delphi), Cliff Baeseman, Shane Miller y Michael Hess, comenzaban a gestar el proyecto Lazarus/Free Pascal que, recogiendo las mismas características que Delphi, incluía tres aspectos diferenciales: ser de código abierto, completamente gratuito y verdaderamente multiplataforma.

Con este libro aprenderá:

- Los principios básicos de la programación estructurada.
- A configurar el IDE de Lazarus.
- A escribir funciones y procedimientos para mejorar la claridad, calidad y tiempo de desarrollo de un programa.
- Las técnicas esenciales de la programación orientada a objetos (POO).
- A crear potentes interfaces gráficas de usuario (GUI).
- A manipular ficheros de texto y binarios como extensión de los programas.
- Las principales técnicas de depuración para corregir los habituales errores de programación.

El libro contiene material adicional que podrá descargar accediendo a la ficha del libro en **www.ra-ma.es**