Marco Cantù

La guía de Delphi

© Danysoft 2009

Versión en castellano

DERECHOS RESERVADOS

El contenido de esta publicación tiene todos los derechos reservados, por lo que no se puede reproducir, transcribir, transmitir, almacenar en un sistema de recuperación o traducir a otro idioma de ninguna forma o por ningún medio mecánico, manual, electrónico, magnético, químico, óptico, o de otro modo. La persecución de una reproducción no autorizada tiene como consecuencia la cárcel y/o multas.

LIMITACIÓN DE LA RESPONSABILIDAD

Tantos el autor como en Danysoft hemos revisado el texto para evitar cualquier tipo de error, pero no podemos prometerle que el libro esté siempre libre de errores. Por ello le rogamos nos remita por e-mail a sus comentarios sobre el libro en attcliente@danysoft.com

DESCUENTOS ESPECIALES

Recuerde que Danysoft ofrece descuentos especiales a centros de formación y en adquisiciones por volumen. Para más detalles, consulte con Danysoft.

MARCAS REGISTRADAS

Todos los productos y marcas se mencionan únicamente con fines de identificación y están registrados por sus respectivas compañias.

Autor: Marco Cantù | www.marcocantu.com
Publicado en castellano por Danysoft
Avda. de la Industria, 4 Edif. 1 3º
28108 Alcobendas, Madrid. España.
902 123146 | www.danysoft.com

Si no es ese usuario rogamos nos lo comente a attcliente @ danysoft.com .

IMPRESO EN ESPAÑA

ISBN : 978-84-932720-9-8

© Danysoft | Madrid, Noviembre-2009 | versión en castellano.

Prólogo

A mi mujer Lella, con amor, ánimo, pasión, dedicación y paciencia

Este libro trata sobre Delphi 2009 y versiones superiores.

No encontrarás en él una introducción a la programación en Delphi, su lenguaje Object Pascal o su Visual Component Library en él. Ya que este libro puedes leer sobre *nuevas funcionalidades de Delphi 2009 para Win32* en cada una de estas áreas.

El libro cubre el soporte Unicode de Delphi 2009, las nuevas funcionalidades del lenguaje (como son los genéricos y los métodos anónimos), las mejoras del IDE, las nuevas clases de la Run Time Library, los nuevos componentes de la VCL (incluyendo el control Ribbon), y la extensión para la arquitectura de bases de datos y la tecnología de múltiples capas de DataSnap.

Como es habitual en mis libros, cubro la teoría pero también te muestro docenas de ejemplos, que puedes descargar y utilizar en tu ordenador. Si aún no tienes Delphi 2009, puedes descargar una versión de prueba y también ver el programa verdadero en acción en una serie de videos enlazados desde la página web:

`http://www.danysoft.com/embarcadero`

Este libro es una secuela de Delphi 2007 Handbook, así que no repito su contenido en absoluto. Si estás interesado en las nuevas funcionalidades de Delphi 2009 desde Delphi 7 (o una vieja versión similar), también puedes contactar con tu comercial Danysoft para adquirir las dos ediciones.

Si estás buscando una introducción a Delphi, sin embargo, puedes dirigirte a mi "Essential Pascal" para los fundamentos del lenguaje y a los libros de la serie "Mastering Delphi" (en particular tanto "Mastering Delphi 7" como "Mastering Delphi 2005").

Puedes encontrar más detalles sobre todos mis libros en mi sitio web personal:

`http://www.marcocantu.com`

Como es habitual, escribir este libro supuso un esfuerzo, y tengo que dar las gracias a muchos desarrolladores de la comunidad de Delphi que me apoyaron de diferentes maneras, comenzando por los revisores técnicos y directores de producto Delphi y los miembros del equipo de I+D. Un gran agradecimiento va para mi mujer e hijos por su paciencia y ánimo.

Espero que disfrutes del resultado, como yo he disfrutado escribiéndolo. Y espero que te guste Delphi 2009, una de las mejores versiones de Delphi que ha habido, como yo lo he hecho.

Tabla De Contenidos

Prólogo..**3**

Tabla de Contenidos..**5**

Introducción...**13**
Delphi...14
Por Qué Importa Win32...14
Este Libro...16
El Autor...17
Información de Contacto...17

Apartado I:
Unicode...**19**

Capítulo 1: ¿Qué es Unicode?...**21**
Caracteres del pasado: desde ASCII a la codificación ISO..................................22
Unicode: Un alfabeto para todo el mundo..24
Desde Puntos de Código a Bytes..27
Puntos de Código Unicode y Diagramas..28

Transformación de Formatos en Unicode (UTF)....29
Estudiando UTF-16....30
Descripciones de Puntos de Código Unicode....33
Byte Order Mark....35
Unicode en Win32....35
Velocidad de Llamada a la API Unicode....38
Parámetros UnicodeString en llamadas al API....40
Unicode y Fuentes y APIs....40
Unicode Antes de Delphi 2009....42
A continuación....43

Capítulo 2: El Tipo Unicode String....45
De AnsiChar a WideChar....46
Char como un Tipo Ordinal....46
Conversiones con Chr....48
Caracteres de 32-bit....48
La Nueva Unidad Character....49
Sobre String y UnicodeString....51
La estructura interna de los Strings....52
UnicodeString y Unicode....55
El Tipo UCS4String....57
Los múltiples tipos de cadenas....57
El Nuevo Tipo AnsiString....58
Creando una cadena de tipo personalizado....59
Gestión de Cadenas UTF-8....63
Conversión de Cadenas....65
Conversiones Ralentizando Nuestro Código....66
Las Llamadas Aseguradas....68
Tenga cuidado con Literales en Concatenación....70
Usando RawByteString....70
Nuevas Funciones de Conversión UTF-8....74
Cadenas y Caracteres Literales....75
Streams y Codificaciones....77
Streaming Listas de Cadenas....78
Definiendo una codificación personalizada....81
Unicode y la VCL....84
¿Un núcleo creciente en la RTL?....85
Unicode en archivos DFM....86
Localizando la VCL....87
A continuación....87

Capítulo 3: Exportando a Unicode....89
Operaciones Char Erróneas....90
Cuidado con Set of Char....90
Evite FillChar para Caracteres....92

Operaciones de Cadena Que Fallan o Ralentizan....94
Atendiendo a Todas las Alertas de Conversión de Cadenas....94
No Mueva Datos String....96
Leyendo y Escribiendo Búfers....97
Añadiendo y Concatenando Cadenas....99
Strings son... Strings (no Bookmarks)....100
Molestas “Importaciones” Actuales....100
InliningTest que usaba AnsiString....101
Utilizando Funciones Ansi-predeterminadas....102
Cadenas Unicode y Win32....105
Aplicaciones de consola Win32....106
PChar y el Puntero Matemático....107
El Problema Con PChar....107
De PChar a PByte....109
PInteger y la Directiva POINTERMATH....109
No utilice PChar para Pointer Math....110
Parámetros Variants y Open Arrays....111
A continuación....111

Apartado II:
Delphi 2009 y Su Compilador....113

Capítulo 4: Nuevas Características del IDE....115
Instalación y Ejecución....116
No Es Necesario el SDK .NET....116
Instalar el limpiador de Windows....117
El Flag -idecaption....118
Gestión de Proyectos en Delphi....118
Actualizando Ficheros de Configuración de Proyecto....119
Rediseñado Diálogo De Opciones De Proyecto....121
Nuevas Opciones de Proyecto para el compilador....122
Otras Nuevas Opciones de Proyecto....125
Ubicación Predeterminada de Proyectos....125
El Project Manager....126
Vistas del Project Manager....127
Construcción de Configuraciones y Opciones de Configuración....129
Administrador de Configuración de Proyecto....132
Gestión de Recursos en el IDE....133
Un "Nuevo" Compilador de Recursos....136
La Clase Delphi Explorer....137
Otras nuevas características....140
Paleta de Herramientas del cuadro de búsqueda....140
Actualización de Asistentes de Componentes....141
¿Algo nuevo en el Editor?....143
Depurador....144

Indice...421

Introducción

Presentado por primera vez por Borland el 14 de Febrero de 1995, Delphi tiene una larga y gloriosa historia de éxitos en las áreas del desarrollo Windows y cliente/servidor. Con millones de aplicaciones escritas en su lenguaje Object Pascal, Delphi ha generado un completo ecosistema de componentes, herramientas, revistas, libros, y (por supuesto) sitios web y recursos online.

Delphi está ahora en su 12ª versión, la 20ª si cuentas todas incluyendo las de su predecesor Turbo Pascal[1], el cual se lanzó por primera vez hace 25 años. ¡Lo que es nuevo en esta versión de Delphi es la compañía propietaria del producto!

Con la adquisición de la división CodeGear de Borland el 1 de Julio de 2008, Delphi se convierte en subsidiario de Embarcadero Technologies. Este cambio de propiedad ocurre bastante tarde en el ciclo de desarrollo de Delphi 2009, por lo que el único efecto práctico de los cambios es la inclusión de ER/Studio en la versión Architect del producto. Desde el comienzo de la división CodeGear dentro de Borland, sin embargo, ha

---

1 La versión actual del compilador de Delphi, de hecho, es la 20.00. Esto es destacado por el valor que la `VER200` define, mencionado en la sección "Versión del Compilador" al comienzo del Capítulo 7.

habido un interés renovado (e inversión en términos de I+D, QA y Documentación) en Delphi, específicamente en sus versiones Win32. Por esto es que es relevante fijarse por un segundo en los temas *políticos* de más alto nivel.

Delphi

Como ya he mencionado, la creación de la división CodeGear y luego la adquisición de esta división por Embarcadero Technologies está proporcionando unos nuevos cimientos para Delphi, y nuevas inversiones en el producto. Incluso si no tiene un marketing agresivo, y queda fuera del radar de la mayoría de publicaciones, aún así Delphi tiene millones de usuarios activos, tanto en el sector de ISV (donde su simplicidad en el despliegue gana sobre las versiones basadas en plataformas) como en los entornos cliente/servidor, donde la estabilidad de una inversión es más valorada que el *estar en la onda* de una plataforma.

Es verdad que la comunidad de Delphi es más pequeña de lo que fue hace unos pocos años, y que parte de ella aguanta con versiones más viejas del producto, pero sigue siendo el líder en muchos países y realmente ha vuelto a estar de buen humor en el último año.

Por Qué Importa Win32

Si lees la mayoría de las revistas de TI, sigues los blog, o asistes a conferencias, parece que solo las últimas tecnologías (y las últimas manías) se valoran para trabajar y todo lo demás está muerto o muriendo. Este es un comentario muy lejano de la realidad.

Desde el desarrollo en COBOL a mainframes, desde ordenadores AS/400 a bases de datos DBF, hay toneladas de tecnología heredada que no solo se mantiene sino en las que se ven nuevas inversiones. Puede ser por razones de compatibilidad, pero también porque las empresas prefieren tener tecnología probada y fiable para el núcleo de su negocio, que arriesgar sus negocios en la tecnología más recientemente promocionada.

Esto no significa, por supuesto, que seguir las tendencias, proporcionar mayor calidad, y potenciar a los usuarios no sea importante. Justo lo opuesto. Si te puedes mantener entregando valor adicional sobre fundamentos sólidos, tienes una ventaja doble. Mirándolo desde el punto de vista de Windows, por ejemplo, Microsoft ha incrementado valor con su creciente conjunto de librerías y arquitecturas basadas en la plataforma .NET. Por otro lado es verdad que, a pesar de la robustez y estabilidad del núcleo, apuntar a las últimas y mejores tecnologías .NET es como fijarse en un objetivo que se mueve rápido, lo que no es exactamente lo mejor cuando necesitas construir tu aplicación cliente/servidor que llevará un par de años crear y esperas no tocarla en los próximos diez años o así.

En el otro extremo están los Micro ISV, pequeños fabricantes de herramientas, desarrolladores de productos shareware, y constructores de utilidades para Internet. Ellos están en una situación de productos que abarcan una vida corta y podrían beneficiarse ciertamente de estar a la última... pero incluso ellos no pueden depender de una plataforma grande y cambiante para desplegar sus aplicaciones. Necesitan algo que funcione en todas y cada una de las versiones de Windows. Esta es una situación en la que Delphi brilla en comparación con la mayoría de soluciones. La única excepción real es Visual C++, pero (si nunca has intentado hacerlo) desarrollar en él no es una experiencia RAD y POO como lo son desarrollar en .NET y VCL.

La biblioteca MFC de Visual C++ es solo una fina capa sobre el API de Windows, mientras que Delphi proporciona lo que se llama un plataforma, con gestión de memoria y servicios de ejecución, una biblioteca de clases bastante grande con muchas ideas en lo referido a la creación de interfaces de usuario, soporte de Internet, y conectividad a base de datos, por nombrar solo las áreas más notables del producto.

Delphi hace tan buenos trabajos produciendo aplicaciones nativas con aspecto Windows, como Skype, que es raro que haya algún signo visible de que una aplicación ha sido desarrollada con Delphi.

Este Libro

Habiendo introducido la situación de Delphi, es el momento de hablar sobre este libro. Como mi reciente "Delphi 2007 Handbook" este no es un manual que abarque todas las funcionalidades de Delphi, lo que requeriría probablemente cerca de 4.000 páginas[2].

A pesar del tamaño, el foco de este libro se centra en las nuevas funcionalidades encontradas en Delphi 2009, o al menos añadidas a Delphi desde que se lanzó Delphi 2007 (como BlackFish SQL y algunas extensiones de metadatos del dbExpress que fueron incluidas en las actualizaciones de Delphi 2007).

No necesita decirse que da un papel central a Unicode y a los cambios en el núcleo del lenguaje (como los genéricos y los métodos anónimos), pero hay también material sobre las actualizaciones para la RTL y la VCL, las mejoras en el soporte para Vista, los últimos controles de interfaz de usuario, y un análisis en profundidad de dbExpress y el nuevo DataSnap 2009 con capacidades multi-hilo del producto.

Como en mis libros anteriores hay mucha teoría y material de introducción mezclado con incontables ejemplos, disponibles online en:

```
http://www.marcocantu.com/dh2009
```

Como mencioné en el "Prologo", también he creado unos videos cortos de demostración (cubriendo cómo funciona el programa, no cómo se escribió) para la mayoría de los ejemplos del libro, disponibles online en:

```
http://www.marcocantu.com/dh2009/videos.html
```

Al publicar este libro, le he dado la forma que más me gusta, contando con la ayuda de editores y revisores en los que confío, y (espero) que este valor se refleje en el resultado. Cuando publiqué "Delphi 2007 Handbook", fue mi primera experiencia de libros electrónicos. Ahora he aprendido de los errores, afinando algunas operaciones, y he reducido alguno de los dolores de cabeza de la publicación para centrarme completamente en escribir durante bastante tiempo. ¡Espero que encuentres que este esfuerzo ha tenido valor!

---

2 Esta suposición (4.000 páginas) es mi estimación de la cantidad de material que he escrito sobre Delphi en los últimos 13 años. Esto es, sin considerar los capítulos que fueron incluidos en las ediciones subsiguientes de mí serie Mastering Delphi.

El Autor

Para aquellos que tienen en sus manos por primera vez uno de mis libros, y para aquellos que no han leído recientemente uno, mi nombre es Marco Cantù, y he estado en el negocio de "escritor de libros de Delphi" desde la primera versión del producto, cuando lancé el "Mastering Delphi" original (un corpulento tomo de 1.500 páginas). Esta no fue mi primera experiencia escribiendo, ya que antes había escrito trabajos sobre Borland C++ y la Object Windows Library.

En los últimos años, junto a mi continuo compromiso con la comunidad de Delphi, también he dedicado mucho tiempo a las tecnologías relacionadas con XML, y a XSLT, con servicios web (incluyendo las implementaciones SOAP y REST), JavaScript y AJAX, y otras tecnologías Web 2.0. Después de un descanso, he vuelto a escribir para auto-publicar mis libros, y no solo sobre Delphi, ya que he finalizado también un volumen sobre redes sociales.

Junto a la escritura, me mantengo ocupado con consultorías (la mayoría sobre arquitectura de aplicaciones), ayudando a vender Delphi en Italia, haciendo revisiones de código, enseñando Delphi, y realizando consultorías generales para desarrolladores.

Con frecuencia también soy ponente en conferencias sobre Delphi y desarrollo en general, incluyendo las nuevas conferencias online de Embarcadero. Si estás interesado en invitarme a hablar en un evento público o a dar una sesión de formación (sobre Delphi o cualquier tema avanzado) en tu empresa, siéntete libre de contactarme vía Danysoft.

Información de Contacto

Finalmente, aquí hay alguna información de contacto:

```
http://blog.marcocantu.com
http://www.marcocantu.com
```

Mi sitio web personal hospeda una página específica para el libro, incluyendo actualizaciones, descarga de código fuente, y otras informaciones:

```
http://www.marcocantu.com/dh2009
```

Tengo una lista de mailing online basada en un grupo de Google donde puedes apuntarte desde mi sitio web. También llevo una newsgroup online con una sección dedicada a discutir mis libros y sus contenidos, disponible en la web (en la sección llamada “marco cantu”) en:

```
http://delphi.newswhat.com
```

Finalmente, en Danysoft traductores y editores de la versión en castellano del libro, han creado una página sobre el mismo en:

```
http://www.danysoft.com/bol/dh2009
```

Apartado I: Unicode

La primera parte de este libro se centra en Unicode, el carácter de codificación internacional estándar que soporta por primera vez Delphi 2009. Los tres capítulos de este apartado introducen este tema, describen la implementación actual, y le guiarán en las tareas de importación y compatibilidad, respectivamente.

- **Capítulo 1: ¿Qué es Unicode?**
- **Capítulo 2: El tipo de cadena Unicode**
- **Capítulo 3: Actualización a Unicode**

Capítulo 1: ¿Qué Es Unicode?

Unicode es el nombre del conjunto de caracteres internacionales que engloba los símbolos de todos los alfabetos escritos del mundo, de hoy y del pasado, y algo más[3]. El estándar Unicode (formalmente denominado "ISO/IEC 10646") está definido y documentado por el Consorcio de Unicode, y contiene más de 100.000 caracteres. Su sitio web principal es:

```
http://www.unicode.org
```

Como la adopción de Unicode es un elemento central de Delphi 2009 y son muchas las cuestiones a abordar, este capítulo se centra sólo en la teoría entre Unicode y las otras codificaciones de caracteres, mientras que el siguiente se centrará en los elementos clave de su aplicación en Delphi.

---

3 Unicode incluye también símbolos técnicos, signos de puntuación, y muchos otros caracteres utilizados en la escritura de texto, aunque no formen parte de ningún alfabeto.

Caracteres del pasado: desde ASCII a la codificación ISO

El Estándar Americano de Código para Intercambio de Información (ASCII) se desarrolló a principios de los años 60 como un estándar de codificación de caracteres, abarca las 26 letras del alfabeto Inglés, tanto en minúsculas como en mayúsculas, los números, símbolos comunes de puntuación, y numerosos caracteres[4] de control.

ASCII utiliza un sistema de codificación de 7 bits para representar 128 caracteres diferentes. Sólo los caracteres entre el # 32 (espacio) y el # 126 (Tilde) tienen una representación visual, tal y como se muestra en la siguiente tabla:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Aunque ASCII fue sin duda el principio (con su conjunto básico de 128 caracteres que siguen siendo parte de la base de Unicode), pronto fue sustituido por versiones extendidas que utilizan notación de 8 bits para añadir nuevos conjuntos de 128 caracteres al conjunto.

Ahora, el problema es que con tantos idiomas en todo el mundo, no hay una forma simple de averiguar qué otros caracteres debemos incluir en el conjunto (a veces mencionado como ASCII-8). Abreviando esta historia, Windows adopta un conjunto diferente de caracteres, llamado *code page* (página de código), cuyo conjunto de caracteres depende de su configuración local y de la versión de Windows. Además de las páginas de

4 Si bien la mayoría de los caracteres de control han perdido su sentido (como el separador de archivo o el tabulador vertical), algunos como el retorno de carro (# 13), salto de línea (# 10), Tab (# 9), y Retroceso (# 8) se usan aún cotidianamente.

código de Windows hay muchos otros convenios basados en enfoques similares de *paginación*.

El más relevante es el estándar ISO 8859, que define conjuntos de caracteres regionales. El más usado (bueno, el más usado en los países occidentales, para ser más preciso) es el conjunto Latino, denominado ISO 8859-1. Aunque parcialmente es similar, la página de códigos Windows 1252 no se ajusta plenamente a la serie ISO 8859-1. Windows agrega caracteres extra como el símbolo €, tal como veremos más adelante.

Si sigo la impresión de todos los caracteres de 8-bits, en mi equipo (que utiliza Windows 1252 como página de códigos por defecto) obtengo la siguiente salida (la suya puede ser diferente)[5]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
128	€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž	
144		‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ
160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

¿Cómo he conseguido tanto esta como la anterior imagen? Utilizando un sencillo programa en Delphi 2009 (llamado FromAsciiToUnicode) que muestra los caracteres en un componente StringGrid, inicialmente con el número de columna y línea pintados en las correspondientes cabeceras. El

5 Si el sistema por defecto utilizase una página de códigos multi-byte, el código de este programa se convierte en un sin sentido, porque la mayoría de los caracteres entre #$80 y #$FF son bytes perdidos, que no se puede mostrar por si mismos.

programa fuerza algunas conversiones tipo del tipo[6] `AnsiChar` para poder gestionar caracteres *tradicionales de 8-bits* (ampliaremos esta información en el siguiente capítulo):

```
procedure TForm30.btnAscii8Click(Sender: TObject);
var
  I: Integer;
begin
  ClearGrid;
  for I := 32 to 255 do
  begin
    StringGrid1.Cells [I mod 16 + 1,
      I div 16 + 1] := AnsiChar (I);
  end;
end;
```

En las versiones anteriores de Delphi usted podría obtener el mismo resultado escribiendo la siguiente versión más simple (que utiliza `Char` en vez de `AnsiChar` para la conversión):

```
  for I := 32 to 255 do
  begin
    StringGrid1.Cells [I mod 16 + 1,
      I div 16 + 1] := Char (I);
  end;
```

Pienso que no es realmente necesario que le diga lo confuso de esta situación con las diferentes codificaciones ISO 8859 (hay 16 de ellas, y todavía no pueden cubrir los alfabetos más complejos), la página de códigos de Windows, de caracteres multi byte soporta el chino y otros idiomas. Con Unicode, todo esto queda desfasado, a pesar de que la nueva norma tiene su propia complejidad y problemas potenciales.

Unicode: Un alfabeto para todo el mundo

Como ya he mencionado, todo esto cambió con la introducción de Unicode. La idea que hay detrás de Unicode (que es lo que hace que sea simple) es

6 Como veremos en detalle en el próximo capítulo, en Delphi 2009 el tipo 'Char' ha cambiado y el antiguo tipo Char, desde Delphi 1 a Delphi 2007, se llama ahora `AnsiChar`.

que todos y cada uno de los caracteres tiene su propio número único (o *punto de código*, para usar el término correcto de Unicode). No quiero ahondar en la teoría completa de Unicode (aunque si usted quiere, puede consultar el libro Unicode (Unicode book) con la norma[7] completa), pero sí poner de relieve sus puntos clave.

En cualquier caso, voy a empezar por ampliar el programa FromAsciiToUnicode, que tiene un tercer botón que muestra esos mismos 256 caracteres (256 menos los 32 caracteres de control iniciales y el carácter de espacio). Esto es lo que obtendrá (y esto no depende de su localización o de la página de código de Windows):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
128																
144																
160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Puede esperar ver exactamente la misma secuencia de caracteres, ya que como todo el mundo sabe, la porción inicial del conjunto de caracteres Unicode mapea la secuencia ASCII, ¿verdad? De hecho, esto es ¡totalmente erróneo! Sólo el conjunto original ASCII-7 tiene su equivalente perfecto en Unicode, y la mayoría del resto de caracteres extendidos también coinciden, pero no todos ellos. De hecho, el tramo entre 128 y 160 es diferente (aunque para ser más precisos, es diferente desde la propia interpretación de Microsoft del código de la página Latín 1). Si usted observa en la imagen[8] anterior, podrá comprobar una colección de símbolos utilizados rara vez ...

---

7 Más información del libro "The Unicode Standard" que puede usted encontrar en: http://www.unicode.org/book/aboutbook.html.

pero hay uno que (por lo menos en mi zona del mundo) es muy importante, el símbolo de moneda €. Para seguir el procedimiento generado, he añadido el código siguiente al mismo programa, de nuevo utilizando dos diferentes tipos de caracteres, AnsiChar y Char:

```
procedure TForm30.btnEuroClick(Sender: TObject);
var
  aChar: AnsiChar;
  uChar: Char;
begin
  aChar := '€';
  uChar := '€';
  ShowMessage ('€ for AnsiChar is ' +
    IntToStr (Ord (aChar)));
  ShowMessage ('€ for UnicodeChar is ' +
    IntToStr (Ord (uChar)));
end;
```

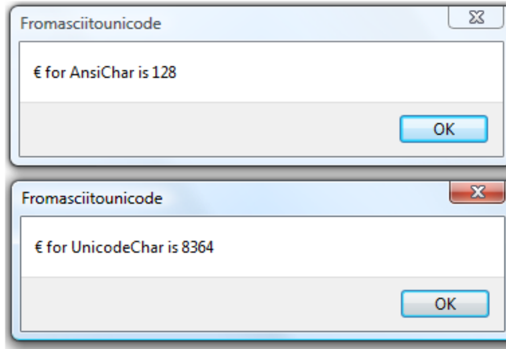
Tenga en cuenta que la forma en que este fragmento de código se compila depende de la opción `--codepage` del compilador, que (si no se especifica) por defecto se refiere a la propia página de códigos[9] del sistema operativo. Así que si recompila el mismo código en un zona diferente del mundo, sin proporcionar una página de códigos explícita, usted obtendrá un programa compilado distinto (no sólo una salida diferente).

Una vez más, los resultados que obtendrá pueden depender de su configuración y parecer algo extraño ... pero tendremos que aprender a vivir con ello en el mundo Unicode. Esto es lo que yo obtengo [ver imagen en página siguiente]:

---

8 Para una demostración de este ejemplo, vea el vídeo de YouTube "Delphi does Unicode ", que realicé en agosto del 2008, durante el período en que se nos permitió a los probadores experimentales de "Tiburón" anunciar las nuevas características del producto. Con estos vídeos quedan cubiertos los otros ejemplos de este capítulo. El vínculo es el siguiente: http://www.youtube.com/watch?v=BJMakOY8qbw

9 La página de códigos utilizada para compilar el programa sólo afecta a la forma en que se gestiona el carácter `AnsiChar`, y no el `Char` Unicode. De hecho, los caracteres Unicode y strings ignoran el código de la página por completo (¡lo que es una gran razón para usarlos!)



Desde Puntos de Código a Bytes

La confusión oculta de Unicode (lo que hace que sea complejo) es que hay muchas formas de representar el mismo punto de código (o valor numérico de caracteres Unicode) en términos de almacenamiento real, en bytes físicos. Si la única manera para representar todos los puntos de código Unicode de un modo simple y uniforme fue utilizar cuatro bytes por cada uno de los puntos de código[10] la mayoría de los desarrolladores percibirían como demasiado caro en términos de memoria y procesamiento.

Una de las opciones es usar pequeñas representaciones con diferente número de bytes (por lo menos 1 ó 2, pero posiblemente hasta 4) para los distintos puntos de código de todo el conjunto Unicode. También se denomina representación de *longitud-variable*. Estas codificaciones tienen nombres de los que probablemente haya oído hablar, como UTF-8 y UTF-16, y de las que examinaremos los detalles técnicos en la siguiente sección.

Existe un error común en creer que UTF-16 puede directamente direccionar todos los puntos de código con dos bytes, pero como Unicode define más de 100.000 puntos de código usted puede fácilmente adivinar que no van a caber. Es cierto, que sin embargo, a veces los desarrolladores utilizan sólo un subconjunto de Unicode, para hacer que encaje en una representación de longitud-fija-2-bytes-por-carácter. En sus primeros días, este subconjunto

---

10 En Delphi los Puntos de Código Unicode están representados utilizando el tipo de datos `UCS4Char`, que se tratan en la sección " Caracteres de 32-bit" del capítulo 2.

de Unicode se llamó UCS-2[11] , ahora lo puede encontrar a menudo como referencia Basic Multilingual Plane (BMP). Sin embargo, esto es sólo un subconjunto de Unicode (uno de sus muchos *Planos*).

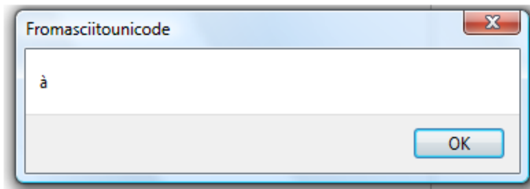
Puntos de Código Unicode y Diagramas

Para ser realmente preciso, debería incluir algún concepto más allá de los puntos de código. A veces, de hecho, múltiples puntos de código pueden ser utilizados para representar un único *grafema* (un carácter visual). Este no es generalmente una letra, sino una combinación de letras o letras y símbolos. Por ejemplo, si usted tiene una secuencia de los puntos de código que representan la letra latina a (#$0061) seguido por el punto de código que representa el acento grave (#$0300), esto se debería mostrar como un único carácter acentuado.

En términos de codificación Delphi, si escribe lo siguiente (disponible de nuevo en el ejemplo FromAsciiToUnicode):

```
var
  str: String;
begin
  str := #$0061 + #$0300;
  ShowMessage (str);
```

El mensaje tendrá un único carácter acentuado:



En este caso tenemos dos Chars, representando dos puntos de código, pero sólo un grafema. El hecho es que mientras que en el alfabeto latino puede usar un punto de código Unicode específico para representar este grafema (*letra con acento grave*), en otros alfabetos la combinación de puntos de

11 El Conjunto universal de caracteres 2-byte (UCS-2) se considera ahora una codificación de caracteres obsoleta. Sin embargo, ambos UTF-16 y UCS-2, mapean los puntos de código contenidos de la misma forma, dentro del BMP, excluyendo el subrogado especial punto de código 2048.

código Unicode es la única manera de obtener un determinado grafema (y su correcta representación).

Transformación de Formatos en Unicode (UTF)

Pocas personas saben que el término común "UTF" es el acrónimo de Unicode Transformation Format. Estos son algoritmos de asignación, *mapeados*, parte del Unicode estándar, que ubica cada punto de código en su mapa (la representación numérica absoluta de un carácter) a una secuencia única de bytes que representan un determinado carácter. Observe que estas asignaciones pueden ser utilizadas en ambas direcciones, convirtiendo diferentes representaciones como de ida y vuelta.

La norma define tres de estos formatos, dependiendo de cuántos bits se utilizan para representar la parte inicial de la serie (los primeros 128 caracteres): 8, 16, ó 32. Es interesante observar que las tres formas de codificación necesitan al menos 4 bytes de datos por cada punto de código.

- **UTF-8** transforma los caracteres en una longitud variable de codificación de 1 a 4 bytes. UTF-8 es muy popular para HTML y protocolos similares, porque es muy compacto y la mayoría de los caracteres (como en etiquetas HTML) entran en el subconjunto ASCII[12].
- **UTF-16** es popular en muchos sistemas operativos (incluyendo Windows) y entornos de desarrollo. Es muy ventajoso ya que la mayoría de caracteres caben en dos bytes, razonablemente compacto y rápido de procesar.
- **UTF-32** otorga mucho sentido a su proceso (todos los puntos de código tienen la misma longitud), pero es memoria que se consume y limita su uso en la práctica.

Un problema relacionado con representaciones multi-byte (UTF-16 y UTF-32) es ¿cuál de los bytes es lo primero? Según el standard, todos los modos

12 Originalmente UTF-8 se representaba de 1 a 6 bytes, para representar a cualquier supuesto futuro punto de código Unicode, pero fue más tarde limitado a usar únicamente la definición formal Unicode hasta el punto de código 10FFFF. Más información, incluyendo un mapa de las diferentes longitudes de puntos de código en UTF-8, en http://en.wikipedia.org/wiki/Utf-8.

están permitidos, por lo que puede tener un UTF-16 BE (big-endian[13]) o LE (little-endian), y lo mismo con UTF-32.

Estudiando UTF-16

¿Cómo crear una tabla de caracteres Unicode como la que he mostrado anteriormente sólo para los ASCII? Podemos empezar por ver los puntos de código en el plano básico multilingüe sobre el 32 (los caracteres habituales de control) y excluir los llamados pares subrogados. No todos los valores numéricos son verdaderos puntos de código UTF-16, ya que hay algunos valores numéricos no válidos para caracteres (llamados *subrogados*) utilizados para crear *códigos pareados* y representar puntos de código por encima de 65535.

Como mostrar una tabla de 256 * 256 es bastante duro, he mantenido la rejilla tal como es y añadido un control TreeView en el costado que le permiten elegir un bloque arbitrario de 256 puntos a mostrar. He usado un TreeView, ya que hay 256 secciones (incluidos los subrogados), por lo que he decidido agruparlos en dos niveles:

Cuando el programa se inicia, se rellena el TreeView con 16 grupos de nivel superior, cada uno contiene 16 subgrupos de segundo nivel, proporcionando así 256 opciones, cada una de las cuales puede mostrar una tabla con 256 caracteres, por un total de 64K de puntos de código (de nuevo, sin tener en cuenta los excluidos):

```
procedure TForm30.FormCreate(Sender: TObject);
var
  nTag: Integer;
  I: Integer;
```

13 La codificación big-endian tiene el byte más significativo en primer lugar (acaba en grande), la codificación little-endian tiene el byte menos significativo primero (acaba en pequeño). Como veremos enseguida, estas series de bytes se marcan a menudo en sus archivos de cabecera con un llamado Byte Orden Mark, Orden de Marca de Byte, (BOM).

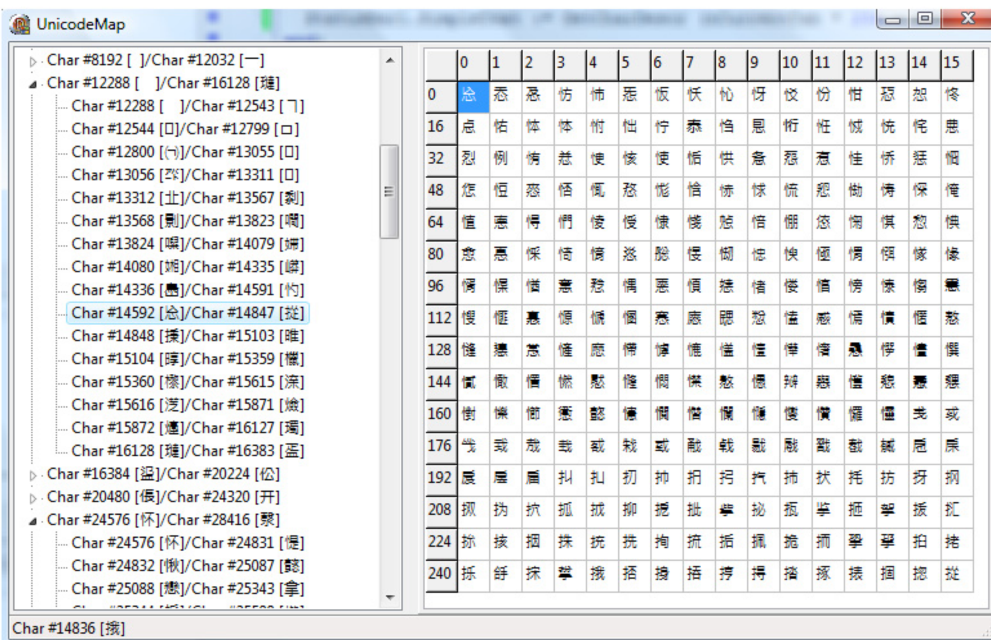
```
  J: Integer;
  topNode: TTreeNode;
begin
  for I := 0 to 15 do
  begin
    nTag := I * 16;
    topNode := TreeView1.Items.Add (nil,
      GetCharDescr (nTag * 256) + '/' +
      GetCharDescr ((nTag + 15)* 256));
    for J := nTag to nTag + 15 do
    begin
      if (J < 216) or (J > 223) then
      begin
        TreeView1.Items.AddChildObject (
          topNode,
          GetCharDescr(J*256) + '/' +
            GetCharDescr(J*256+255),
          Pointer (J));
      end
      else
      begin
        TreeView1.Items.AddChildObject (
          topNode,
          'Puntos de Código Subrogado',
          Pointer (J));
      end;
    end;
  end;
end;

// Función de Ayuda
function GetCharDescr (nChar: Integer): string;
begin
  if nChar < 32 then
    Result := 'Char #' + IntToStr (nChar) + ' [  ]'
  else
    Result := 'Char #' + IntToStr (nChar) +
      ' [' + Char (nChar) + ']';
end;
```

Como puede ver en el código anterior, cada nodo del TreeView obtiene un número con su número de página que es la posición inicial de su campo de datos (generalmente un puntero). Este se utiliza cuando se selecciona un segundo nivel en el elemento TreeView (que es un nodo que tiene un nodo padre) para calcular el punto inicial de la tabla:

```
procedure TForm30.TreeView1Click(Sender: TObject);
var
  I, nStart: Integer;
begin
```

```
    if (TreeView1.Selected.Parent <> nil) then
    begin
      // un segundo nivel de nodo
      nCurrentTab := Integer(TreeView1.Selected.Data);
      nStart := nCurrentTab * 256;
      for I := 0 to 255 do
      begin
        StringGrid1.Cells [I mod 16 + 1, I div 16 + 1] :=
          IfThen (I + nStart >= 32, Char (I + nStart), '');
      end;
    end;
  end;
```



Compruebe el uso de la función de `IfThen` para sustituir opcionalmente los 32 caracteres iniciales por una cadena vacía. El punto de partida del TreeView actual se mantiene en el campo `nCurrentTab` del formulario. Esta información es necesaria para mostrar el punto de código y su valor cuando el usuario mueve el ratón sobre las celdillas de la tabla:

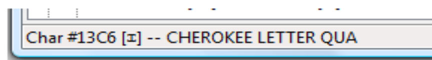
```
procedure TForm30.StringGrid1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
  gc: TGridCoord;
  nChar: Integer;
begin
  gc := StringGrid1.MouseCoord(X, Y);
```

```
  nChar := (gc.Y - 1) * 16 + (gc.X - 1);
  StatusBar1.SimpleText :=
    GetCharDescr (nCurrentTab * 256 + nChar);
end;
```

Cuando utilice el programa y navegue por las diferentes páginas de código de diferentes alfabetos, observará a menudo caracteres que no se muestran correctamente. Lo más probable es que se deba a la fuente que esté utilizando, ya que no todas las fuentes proporcionan una representación adecuada para todo el conjunto de caracteres Unicode. Esta es la razón por la que he añadido al programa UnicodeMap la capacidad de seleccionar una fuente distinta (se logra con un doble clic sobre la tabla). Puede encontrar más información acerca de este problema en la sección "Unicode y Fuentes y APIs" más adelante en este capítulo.

Descripciones de Puntos de Código Unicode

En el sitio web del Consorcio de Unicode, usted podrá encontrar gran cantidad de información, incluso un archivo de texto con una descripción escrita de un gran número de puntos de código (la mayoría de ellos con exclusión de los ideogramas unificados para el chino, japonés, y coreano). He usado este archivo para crear una versión ampliada del programa UnicodeMap, llamado UnicodeData. La interfaz de usuario se basa en la misma estructura, pero el programa lee y analiza el archivo UnicodeData.txt[14], y añade la descripción de caracteres disponibles en la barra de estado cuando nos desplacemos sobre la rejilla:



Analizar este archivo no es francamente simple, ya que no están todos los símbolos Unicode representados. He recurrido a la creación de un StringList con información en formato *charnumber=description* extraída del archivo. El archivo original utiliza un punto-y-coma para separar los campos y un carácter de retorno de carro (solo, sin combinarlo con salto de línea) para

14 La URL de este archivo es: http://unicode.org/Public/UNIDATA/UnicodeData.txt. Existe un segundo archivo mucho mayor (que no he utilizado en la demo) para los ideogramas unificados disponible en: http://www.unicode.org/Public/UNIDATA/Unihan.zip.

cada registro. Después de cargar el archivo entero en una cadena, utilizo el código siguiente para analizar y mover las dos descripciones a la sección de información (ya que a veces sólo una u otra descripción es relevante):

```
nPos := 1;
// ahora analizamos el archivo Unicode
while nPos < Length (strData) - 2 do
begin
  strSingleLine := ReadToNewLine (strData, nPos);
  nLinePos := 1;
  strNumber := ReadToSemicolon (
    strSingleLine, nLinePos);
  strDescr1 := ReadToSemicolon (
    strSingleLine, nLinePos);
  Skip8Semi (strSingleLine, nLinePos);-
  strDescr2 := ReadToSemicolon (
     strSingleLine, nLinePos);

  sUnicodeDescr.Add(strNumber + '=' +
  strDescr1 + ' ' + strDescr2);
end;
```

Este código puede ser ejecutado en el mensaje de un manejador de mensajes `wm_user` colocado en el evento `OnCreate` del formulario principal, para permitir al sistema iniciar el formulario principal antes de realizar esta larga operación. La barra de estado se actualiza en el bucle para informar a los usuarios del progreso actual. El bucle tiene otras sentencias de finalización, para saltarse los caracteres por encima de `$FFFF`.

La información almacenada en la lista de cadenas se extrae cuando tiene que mostrar la descripción de un carácter, con este código adicional del método `StringGrid1MouseMove`:

```
  if Assigned (sUnicodeDescr) then
  begin
    strChar := IntToHex (nChar, 4);
    nIndex := sUnicodeDescr.IndexOfName(strChar);
    if nIndex >= 0 then
      StatusBar1.SimpleText := StatusBar1.SimpleText +
        ' -- ' + sUnicodeDescr.ValueFromIndex [nIndex];
  end;
```

Obteniendo la información acerca de los puntos de código, el programa puede también crear un elemento más lógico en el árbol. Esto no es demasiado difícil para la mayoría de los alfabetos, ya que la mayoría de sus símbolos tienen un nombre genérico sin indicar que pertenecen a un grupo determinado. Una vez hayamos obtenido nuestra propia agrupación de todos los puntos de código Unicode, será posible la lectura de distintos

documentos[15], sin tener que analizar de nuevo el archivo `UnicodeData.txt`.

Byte Order Mark

Los archivos de almacenamiento de caracteres Unicode a menudo utilizan una cabecera de inicio, llamada Byte Orden Mark (BOM) como una marca que indica el formato Unicode utilizado y la ordenación de sus bytes (BE o LE). La siguiente tabla proporciona un resumen de los distintos BOM, que pueden ser de 2, 3, ó 4 bytes de longitud:

`00 00 FE FF`	UTF-32, big-endian
`FF FE 00 00`	UTF-32, little-endian
`FE FF`	UTF-16, big-endian
`FF FE`	UTF-16, little-endian
`EF BB BF`	UTF-8

Veremos en el próximo capítulo cómo Delphi gestiona el BOM dentro de sus clases streaming. El BOM aparece en el comienzo de un archivo con los datos Unicode inmediatamente después de este. Por lo tanto, un archivo UTF-8 con el contenido *AB* contiene cinco valores hexadecimales (3 para el BOM, 2 para las letras):

```
EF BB BF 41 42
```

Unicode en Win32

Desde sus primeros días, la API de Win32 (que se remonta a Windows NT) ha incluido el soporte de los caracteres Unicode. La mayoría de funciones de la API de Windows tienen dos versiones disponibles, una versión Ansi marcada con la letra A y una versión WideString marcada con la letra W.

---

15 Como veremos en el próximo capítulo, la nueva unidad de código Characters incluye métodos para el reconocimiento de un punto de código Unicode, dependiendo si es un símbolo, una marca de puntuación, un espacio ...

Como ejemplo, vea las siguientes líneas de código de Windows.pas en Delphi 2007:

```
function GetWindowText(hWnd: HWND; lpString: PChar;
  nMaxCount: Integer): Integer; stdcall;
function GetWindowTextA(hWnd: HWND; lpString: PAnsiChar;
  nMaxCount: Integer): Integer; stdcall;
function GetWindowTextW(hWnd: HWND; lpString: PWideChar;
  nMaxCount: Integer): Integer; stdcall;

function GetWindowText; external user32
  name 'GetWindowTextA';
function GetWindowTextA; external user32
  name 'GetWindowTextA';
function GetWindowTextW; external user32
  name 'GetWindowTextW';
```

Las declaraciones son idénticas, pero ambas utilizan `PAnsiChar` o `PWideChar` para el paso del string. Observe que la versión simple sin indicación de formato de cadena es sólo un marcador de posición para uno de ellos, invariablemente utiliza la 'A' en la versiones anteriores de Delphi (de dónde se ha tomado el código), mientras que en Delphi 2009 (como veremos) la opción por defecto se convierte en la versión "W". Básicamente, cada función del API que tiene parámetros strings tiene dos versiones separadas, mientras que todas las funciones que no utilizan strings tienen una sola, por supuesto.

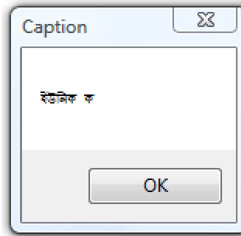
Windows 95 (y las siguientes versiones de Windows 98 y ME) aplicaban las funciones A y proveían las funciones W como alias que convertían Wide en Ansi. Esto significa que estas funciones generalmente no soportan Unicode, con algunas excepciones como `TextOutW` (que se implementa como una verdadera función Unicode en Windows 95/98/ME). Por otra parte, Windows NT y las siguientes versiones basadas en él (Windows 2000, XP y Vista) aplican funciones W, y proveen las funciones A como un alias de conversión de Ansi a Wide (a veces, ralentizando las operaciones).

Incluso en versiones anteriores de Delphi usted podría pasar un valor WideString a una API "W", llamándola explícitamente. Por ejemplo, el programa UnicodeWinApi (que puede ser compilado tanto en Delphi2007 como en Delphi 2009), el código es:

```
procedure TForm30.btnMessageAClick(Sender: TObject);
begin
  MessageBoxA (Handle, 'ইউনিকোড        কী  ','Caption ',   MB_OK);
end;

procedure TForm30.btnMessageWClick(Sender: TObject);
var
  ইউন  : WideString;
begin
  ইউন   :=  'ইউনিক      ক ';
  MessageBoxW (Handle, PWChar (ইউন  ), 'Caption',    MB_OK);
end;
```

La primera llamada la versión ANSI de `MessageBox` muestra un mensaje con una secuencia de símbolos de interrogación, mientras que la segunda (que se muestra aquí) tiene la salida correcta[16]:

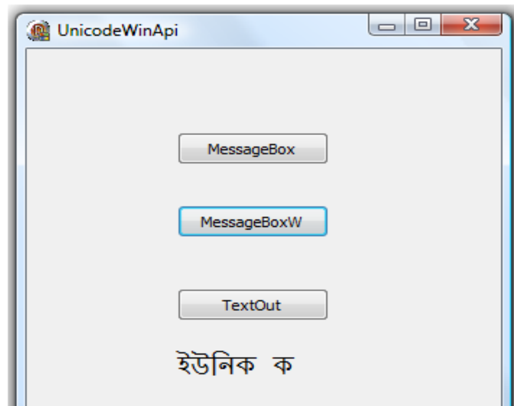


Observe el uso de caracteres Unicode en cadenas del código fuente y también en el nombre de una variable local. Esto se soportaba ya en Delphi 2006 y 2007 permitiéndole además guardar el archivo de código fuente en formato Unicode (UTF-16 o UTF-8). Una nueva llamada dibuja un texto en un raro alfabeto sobre el formulario, con la salida mostrada a continuación. Este es el código fuente, tomado del editor de texto:

```
procedure TForm30.btnTextOutClick(Sender: TObject);
var
  ws: WideString;
begin
  ws := 'ইউনিক        ক ';
  Canvas.Font.Size := 20;
  TextOutW(Canvas.Handle, 104, 224, PWChar(ws), Length (ws));
end;
```

---

16 La frase está escrita en bengalí y significa "¿Qué es Unicode?", por lo menos según http://www.unicode.org/standard/translations/bangla.html. Para el nombre de la variable he utilizado parte de la frase ... que probablemente (o afortunadamente) ¡no signifique nada!



Hay dos hechos importantes más a tener en cuenta sobre las cadenas en el API de Win32. El primero es que algunos de los antiguos sistemas operativos (como Windows 95) ofrecen sólo una aplicación parcial de las funciones del API. El segundo es que COM utiliza un enfoque diferente, con cadenas en el formato BSTR, asignadas en Delphi al tipo[17] non-reference-counted WideString.

Velocidad de Llamada a la API Unicode

Usted puede preguntarse si utilizar Unicode llamando a la API de Windows es más lento o más rápido que utilizando la sencilla API "A". Huelga decir que he tenido la misma duda, como Unicode causa algunas cargas extras en memoria, me preguntaba si esta iniciativa realmente tiene sentido para programas que no necesiten Unicode.

En teoría, como la única implementación de la API de Windows Vista y XP es la basada en Unicode, deberíamos esperar una velocidad mayor de ejecución, así como que el código salte alguna conversión de cadena durante su llamada. De modo que lo he intentado con el siguiente código, que es parte del ejemplo UniApiSpeed:

```
procedure TForm30.btnUserNameClick(Sender: TObject);
var
```

17 Más información sobre el tipo WideString y su relación con COM está disponible en la sección “Unicode Antes de Delphi 2009” al final de este capítulo. El soporte COM todavía se basa en el tipo WideString Delphi en 2009, muy poco ha cambiado en esa área del producto

```
  I:Integer;
  nSize: DWORD;
  t1: TDateTime;
  str: string;
  pch: PChar;
begin
  nSize := 100;
  SetLength (str, nSize);
  pch := PChar (str);
  t1 := Now;
  for I := 1 to 10000 do
  begin
    GetUserName (pch, nSize);
  end;
  t1 := Now - t1;
  Memo1.Lines.Add ((Sender as TButton).Caption + ' ' +
    FormatDateTime ('ss.zzz', t1));
```

He compilado el mismo programa en Delphi 7 y en Delphi 2009 y observé que los resultados fueron casi idénticos. He intentado un bucle similar basado en la llamada a la API `SetWindowText` y en este caso he notado un efecto muy extraño. Si ejecuto la aplicación bajo el depurador, tarda un 15% menos tiempo que su contrapartida en Delphi 7, si se ejecuta independientemente, lo hace de forma mucho más lenta. El problema, sin embargo, es que el programa consume un tiempo dibujando, una y otra vez, el título, por lo que estos resultados están totalmente alterados.

Estas dos pruebas puede que no sean muy pertinentes. Debería haber intentado con muchas otras llamadas a la API para poder tener una conclusión definitiva, pero esto demuestra que pasando a Unicode, puede tener una velocidad similar o ligeramente mejorada en llamadas a la API. Usted no ganará demasiado (al menos mientras no utilice caracteres Unicode), pero no incurrirá en ningún gasto extra por lo general[18].

---

18 Como veremos en el próximo capítulo, en efecto, hay importantes gastos generales potenciales relacionados con las conversiones implícitas de cadena realizada por el compilador de Delphi. Este es uno de los temas cubiertos en la sección "Conversiones lentas dentro del Código" del capítulo 2. También notará que algunas de las funciones ANSI de cadena relacionadas se llevaron a cabo en Delphi 2007 llamando a rutinas del API de Windows que requieren de conversiones y, a continuación volver a Unicode. Las llamadas a estas rutinas en Delphi 2009 deberían ser mucho más rápidas, como ha detallado Jan Goyvaerts en este blog: http://www.micro-isv.asia/2008/09/speed-benefits-of-using-the-native-win32-string-type.

Parámetros UnicodeString en llamadas al API

Aunque la mayoría de funciones de la API de Windows que tienen como parámetro una cadena están declaradas en la unidad Windows con un parámetro `PChar`, hay algunas excepciones a esta regla.

Las declaraciones de la API `GetTextExtentPoint32`, `ExtTextOut`, `LoadKeyboardLayout`, `DrawText`, `SetWindowText`, `LoadCursor`, `WinHelp`, y `HtmlHelp` tienen versiones overloaded (e inlined) tomando un parámetro UnicodeString. Supongo que esto podría ayudar a aplicar la conversión correcta en caso que usted esté pasando una cadena, cualquier cadena, a estas funciones (usted podrá comprenderlo mejor después de leer sobre los diferentes tipos de cadenas en Delphi 2009, en el próximo capítulo).

No está claro por qué estas funciones tengan este trato especial comparadas con las docenas de otras funciones de la API de Windows que tienen parámetros `PChar`. Una de las razones podría ser el aumento de la compatibilidad entre la VCL para Win32 y la VCL para. NET si el tipo de cadena que se utiliza preferiblemente es un `Pchar`.

Unicode y Fuentes y APIs

Otro hecho importante a tener en cuenta es que mientras Windows soporta realmente Unicode, esto lo hace de formas diferentes en sus distintas versiones. Las series Windows 9x (una sigla que quiere decir 95, 98, o ME – Mileninium Edition) tienen un soporte limitado de Unicode. Si usted tiene el Windows 2000, Windows XP, o algunos de los sistemas operativos para versiones servidor, puede usted tener ventaja por el suplemento de soporte a idiomas. Este puede ser instalado en el Idioma regional y opciones de idiomas adicionales del panel de control. Este apoyo extra proviene principalmente del modo fuentes extendidas (o Unicode-habilitado). Vista ha ampliado el soporte a Unicode por defecto.

Cuando Windows XP y Vista necesitan mostrar un punto de código Unicode, y este punto de código no está disponible en la fuente, a veces llevan a cabo una “sustitución de fuentes”, es decir, muestran el punto de código en una

fuente distinta[19]. Esto depende de la API que se llame para la visualización de texto (`DrawText`, `TextOut`, `ExtTextOut` se comportan distinto), sobre la fuente que esté usando, y en el punto de código dado. Esta es la razón por la que utilizar una fuente completa Unicode es una buena idea[20].

Si está usted interesado en más detalles, puedes echar un vistazo al ejemplo UniFont - Subst, que, básicamente dibuja un conjunto de cadenas con diferentes fuentes y diferentes llamadas a la API en el formulario. El programa utiliza tres tipos de letra estándar (Times New Roman, Tahoma, y Arial) que se muestran de arriba a abajo, y la tres llamadas a la API mencionadas anteriormente que se muestran de izquierda a derecha. Esta es una de las tres partes (hay una para cada fuente), del código que las dibuja:

```
Canvas.Font.Name := 'Times New Roman';
aRect := Rect(10, 60, 250, 100);
DrawText(Canvas.Handle, PChar (str1), Length (str1),
  aRect, DT_LEFT or DT_SINGLELINE or DT_EXPANDTABS
  or DT_NOPREFIX);
TextOut (Canvas.Handle, 260, 60, PChar (str1),
  Length (str1));
aRect := Rect(510, 60, 750, 100);
ExtTextOut (Canvas.Handle, 510, 60, 0, aRect,
  PChar (str1), Length (str1), nil);
```

La cadena se define mediante una secuencia de puntos de código Unicode consecutivos a partir de una posición al azar:

```
var
  str1: string;
  nPoint: Word;
  I: Integer;
begin
  nPoint := 32 + random (1024*64 - 32 - numberOfChars);
  if (nPoint >= $D800 - numberOfChars) and
    (nPoint <= $DFFF) then
  begin
    // reintentar y saltar
    PaintFonts;
    Exit;
  end;
```

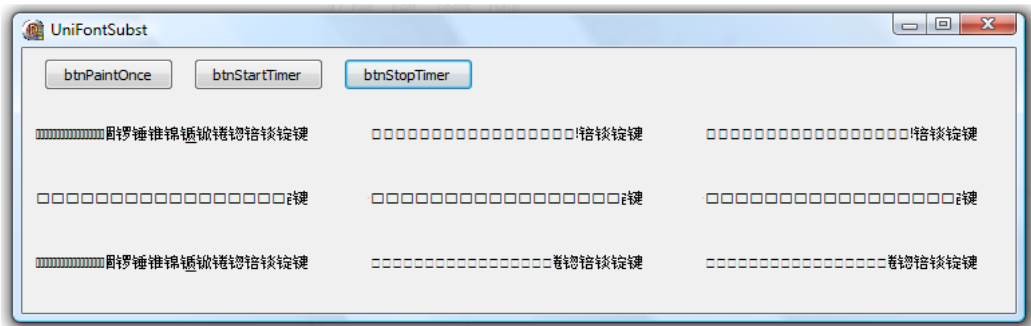
---

19 Puede encontrar gran cantidad de información detallada acerca de la sustitución de fuentes realizada por diferentes llamadas a la API de Microsoft en este artículo: http://www.microsoft.com/globaldev/getwr/steps/wrg_font.mspx.

20 Para obtener información y disponibilidad de fuentes Unicode para Windows, puede referirse a Recursos Unicode de Alan Wood(data de unos pocos años atrás, pero se mantiene hasta la fecha) en: http://www.alanwood.net/unicode/fonts.html

```
str1 := ConvertFromUtf32(UCS4Char (nPoint));
for I := 1 to numberOfChars do
  str1 := str1 + ConvertFromUtf32(
    UCS4Char (nPoint + I));
```

En este programa se puede pintar el formulario de una vez o activar un temporizador automático para consecutivas proyecciones. En la mayoría de los casos, verá la misma cadena mostradas 9 veces. Muy a menudo, sin embargo, verá que el texto no se muestra correctamente cuando se utilizan determinados tipos de letra (Tahoma, más comúnmente), sustituyéndose por bloques cuadrados. En otros casos, verá que algunas de las llamadas a la API (en `DrawText` en particular) puede mostrar los puntos de código de cualquier forma, pero reemplazando la fuente con otra distinta. A continuación se muestra una captura de pantalla en la que se puede ver ambos casos a la vez:



Unicode Antes de Delphi 2009

Si Unicode se soporta en el núcleo de nuestro lenguaje (para el tipo string) e igualmente las bibliotecas runtime y las bibliotecas de componentes visuales (VCL) es ciertamente, una nueva característica de Delphi 2009, el soporte parcial de Unicode ha sido parte de Delphi desde hace muchos años.

Desde anteriores versiones, en su mayoría para apoyar COM, Delphi tuvo un tipo de datos WideChar (caracteres 16-bits) y un tipo de datos WideString (strings de caracteres WideChar). Sin embargo, el tipo WideString no era (y sigue sin serlo) una referencia óptima y es mucho menos eficiente que las

cadenas habituales de Delphi. Ello no es más que una encapsulación del tipo[21] de dato COM BSTR.

También hay varias unidades y clases con un apoyo específico para WideStrings, incluida la unidad WideStrUtils de características extras (que también incluye una buena cantidad de funciones relacionadas con UTF-8), la clase `TWideStringList`, y WideString ampliamente soportadas en las clases `TDataSet` y `Tfield`.

A continuación

No quiero ahondar en los detalles de las características que cambian considerablemente en Delphi 2009 y que tendremos tiempo para cubrir posteriormente en este libro. Ahora es hora de empezar a buscar la aplicación efectiva de las cadenas Unicode, el nuevo tipo UnicodeString. Hay muchos cambios en el uso de cadenas en Delphi 2009, esta es la razón por la que el próximo capítulo es uno de los más largos en el libro.

21 WideString es un BSTR COM que usa UTF-16 en Windows 2000 y superiores, si bien es sobre la base de UCS-2 en Win9x y NT. Algunos detalles más están disponibles en: http://msdn.microsoft.com/en-us/library/ms221069.aspx

Capítulo 2: El Tipo Unicode String

Como usted seguramente sabe, una de las nuevas, y mejores características de Delphi 2009, es la introducción de un nuevo tipo de cadena UnicodeString, que es también el formato ligado al tipo básico string. Cada vez que escribe "string" en su código, se está refiriendo de hecho a UnicodeString, mientras que en versiones anteriores de Delphi (con excepción de Delphi 1) se referían a AnsiString.

Junto con Char que se convierte en un alias de WideChar, este es un cambio muy significativo, que afecta a todo su código base. Es por eso que no será suficiente un solo capítulo para explicar todo lo que necesita saber. Voy a tratar la totalidad del nuevo tipo string, pero son muchos e inevitables los inconvenientes que surgen al importar código Delphi existente a la nueva

versión del compilador Embarcadero, habilitada para Unicode, que trataré en el próximo capítulo.

De AnsiChar a WideChar

Durante algún tiempo, Delphi ha incluido dos tipos de datos para representar caracteres:

- **AnsiChar**, con una representación de 8 bits (con un total de 256 símbolos diferentes), interpretados en función de su código de página;
- **WideChar**, con 16 bits de representación (con un total de 64K de símbolos diferentes)[22].

En este sentido, nada ha cambiado en Delphi 2009. Lo que es diferente es que el tipo Char antes era un alias de AnsiChar y ahora es un alias de WideChar. Cada vez que el compilador detecta Char en su código, lee realmente WideChar. Tenga en cuenta que no hay forma de cambiar esta nueva compilación por defecto[23].

Este es un cambio, que afecta a una gran cantidad de código fuente y con muchas ramificaciones. Por ejemplo, el puntero `PChar` ahora un alias de `PWideChar`, en lugar de `PAnsiChar`, como solía ser. Veremos cómo esto afecta a las llamadas a las funciones de la API de Windows en una sección posterior de este capítulo.

Char como un Tipo Ordinal

El nuevo tipo Char *de gran tamaño* sigue siendo un tipo ordinal, de modo que puede utilizar `Inc` y `Dec` sobre él, para escribir bucles con un contador Char, y cosas por el estilo.

---

22 WideChar es simplemente un entero de 16-bits sin signo sin ningún tipo de codificación de caracteres específica adjunta. Cuando se utiliza un UnicodeString, sin embargo, un WideChar puede interpretarse como un subrogado, de modo que dos WideChar pueden ser enlazados para representar a un único punto de código Unicode. Amplía esta información en la sección “UnicodeString y Unicode”.

```
var
  ch: Char;
begin
  ch := 'a';
  Inc (ch, 100);
  ...
  for ch := #32 to High(Char) do
    str := str + ch;
```

Estos fragmentos de código son parte de los diferentes métodos del ejemplo TestChar. El único asunto que podría ocasionarle algún problema (limitado) es cuando declare un conjunto basado en el tipo Char:

```
var
  CharSet = set of Char;
begin
  charSet := ['a', 'b', 'c'];
  if 'a' in charSet then
    ...
```

En este caso, el compilador asume que usted importa código existente a Delphi 2009, y decide considerar la posibilidad de que este Char sea un AnsiChar (ya que como un set sólo puede tener 256 elementos como máximo[24]) y emitir un mensaje de advertencia:

```
W1050 WideChar reduced to byte char in set expressions.
Consider using 'CharInSet' function in 'SysUtils' unit.
```

El código probablemente funcione como se espera, pero no todo el código recuperado se transformará fácilmente, ya que no es posible obtener un conjunto de todos los caracteres nunca más. Si esto es lo que usted necesita, tendrá que cambiar su algoritmo (posiblemente siguiendo lo que nos sugiere la advertencia, tocaré este tema ampliándolo en el siguiente capítulo, en la sección "Cuidado con Set of Char", centrado en importar código existente a Delphi 2009).

---

23 Como con el tipo string, el tipo Char está específicamente mapeado a un tipo fijo de datos por código. Los desarrolladores solicitaron una directiva de compilación para cambiarlo, pero sería una pesadilla en terminos de control de calidad, soporte o compatibilidad de paquetes entre otras cosas. De todas formas, todavía tiene la opciónde convertir su código para que use un tipo específico como el AnsiChar.

24 Si usted intenta declarar un amplio conjunto de, digamos, una serie de números enteros (como lo hice en algunas líneas comentadas de código del ejemplo CharTest) obtendrá el error: `E2028 Sets may have at most 256 elements.`

Si lo que está buscando, en cambio, es suprimir las advertencias (la compilación de las cinco líneas de código anterior causan dos de estas) puede escribir:

```
var
  charSet: set of AnsiChar; // suprime la alerta
begin
  charSet := ['a', 'b', 'c'];
  if AnsiChar('a') in charSet then // suprime la alerta
    ...
```

Conversiones con Chr

Valore también que usted no sólo puede convertir un valor numérico a un carácter usando la conversión al tipo AnsiChar o WideChar, sino también sobre la base de la clásica técnica Pascal, el uso de la función *mágica del compilador* `Chr` (que puede ser considerada como la contraria de `Ord`). Esta función mágica se ha ampliado para pasar una palabra como parámetro, en lugar de un byte.

Fíjese, sin embargo, que, a diferencia de caracteres literales (cubiertos en la sección "Cadenas y Caracteres Literales" más adelante en este mismo capítulo), las llamadas a `Chr` estarán ahora siempre interpretadas en el ámbito Unicode. Por lo tanto, si su antiguo código declara:

```
Chr (128)
```

desde Delphi 2007 a Delphi 2009 puede tener alguna sorpresa. Si utiliza `#128`, en su lugar, usted podría obtener un resultado diferente, o no, dependiendo de su código de página.

Caracteres de 32-bit

A pesar de que el tipo por defecto Char está ahora asignada a WideChar, vale la pena comentar que Delphi define también un tipo de carácter de 4-bytes, `UCS4Char`, que se define en la unidad System como:

```
type
  UCS4Char = type LongWord;
```

Si bien este tipo de definición y el correspondiente para UCS4String (definido como un array de UCS4Char) ya se encontraban en Delphi 2007, la relevancia del tipo de datos UCS4Char en Delphi 2009 viene del hecho de que es ahora cuando se utilizan significativamente en varias rutinas de la RTL, incluidas las de la nueva unidad Character que trataremos próximamente.

La Nueva Unidad Character

Para un mejor soporte de los nuevos caracteres Unicode (y también de las cadenas Unicode, por supuesto) Delphi 2009 introduce una nueva unidad en la RTL, denominada Character[25]. Esta unidad define la clase sellada TCharacter que es básicamente una colección de funciones[26], estáticas de clase, además de una serie de rutinas globales asignadas a las funciones de clase como públicas (y algunas como privadas).

La unidad también define dos tipos enumerados interesantes. El primero se llama TUnicodeCategory y mapea distintos caracteres en amplias categorías tales como control, espacio, mayúsculas o minúsculas, número decimal, puntuación, símbolos matemáticos, y muchos más. El segundo enumerado se llama TUnicodeBreak y define la familia de los diversos espacios, guión, y retornos.

La clase sellada TCharacter tiene más de 40 métodos que o bien trabajan en un carácter independiente o bien dentro de una cadena:

- Obtener la representación numérica del carácter (GetNumericValue).
- Solicitar la categoría (GetUnicodeCategory) o clasificarla contra una de las distintas categorías (IsLetterOrDigit, IsLetter, IsDigit, IsNumber, IsControl, IsWhiteSpace, IsPunctuation, IsSymbol, y IsSeparator)

---

25 El "Character" como nombre de unidad parece un poco fuera de sincronización con la nomenclatura general de normas adoptadas por la RTL de Delphi, donde "Utils" es a menudo el final de la colección de las funciones. Depende más del hecho de que, en efecto, hay una clase dentro de la unidad, llamada TCharacter, aunque es una clase muy extraña.

26 Para la definición de métodos de clase estáticos y clases selladas ver, entre otras fuentes, mi Manual de Delphi 2007.

- Comprobar si es minúscula o mayúscula (`IsLower` y `IsUpper`) o la conversión de estas (`ToLower` y `ToUpper`)
- Verificar si es parte de un par subrogado a UTF-16 (`IsSurrogatePair`, `IsSurrogate`, `IsLowSurrogate`, e `IsHighSurrogate`)
- Convertirlo desde y hacia UTF32 (`ConvertFromUtf32` y `ConvertToUtf32`)

Las funciones globales son casi una coincidencia exacta de estos métodos estáticos de clase, algunos de los cuales corresponden a las funciones de la RTL de Delphi aunque en general con diferentes nombres. Hay sobrecarga de algunas de las funciones RTL básicas de trabajo con caracteres, con versiones ampliadas que requieren el código Unicode-habilitado. Por ejemplo, en el programa CharTest he añadido los siguientes fragmentos al tratar de convertir a una letra acentuada en mayúsculas:

```
var
  ch1: Char;
  ch2: AnsiChar;
begin
  ch1 := 'ù';
  Memo1.Lines.Add ('WideChar');
  Memo1.Lines.Add ('UpCase ù:  ' + UpCase(ch1));
  Memo1.Lines.Add ('ToUpper ù:  ' + ToUpper (ch1));

  ch2 := 'ù';
  Memo1.Lines.Add ('AnsiChar');
  Memo1.Lines.Add ('UpCase ù:  ' + UpCase(ch2));
  Memo1.Lines.Add ('ToUpper ù:  ' + ToUpper (ch2));
```

El código tradicional de Delphi (la función `UpCase` sobre la versión AnsiChar) se ocupa solamente de caracteres ASCII, por lo que no convertirá este carácter[27]. Lo mismo también es aplicable (probablemente por razones de compatibilidad hacia atrás) si usted pasa un WideChar a la misma. La función `ToUpper` es correcta (termina llamando a la función `CharUpper` de la API de Windows):

```
WideChar
UpCase ù: ù
ToUpper ù: Ù
AnsiChar
UpCase ù: ù
```

---

27 Lo mismo también es válido para la función `UpperCase`, que gestiona sólo ASCII, mientras que `AnsiUpperCase` gestiona todo en Unicode, a pesar de su nombre.

```
ToUpper ù: Ù
```

Observe que puede mantener su actual código de Delphi, con la llamada a `UpCase` sobre un Char, y que se mantendrá la norma de comportamiento de Delphi.

Para una mejor demostración de las características mencionadas en Unicode introducidas por la unidad Characters, puede ver el método `btnUTF16Click` del ejemplo CharTest que define una cadena como[28]:

```
var
  str1: string;
begin
  str1 := '1.' + #9 + ConvertFromUtf32 (128) +
    ConvertFromUtf32($1D11E);
```

El programa a continuación, hace las siguientes pruebas (todas retornando True) sobre los diversos caracteres de la cadena:

```
  TCharacter.IsNumber(str1, 1)
  TCharacter.IsPunctuation (str1, 2)
  TCharacter.IsWhiteSpace (str1, 3)
  TCharacter.IsControl(str1, 4)
  TCharacter.IsSurrogate(str1, 5)
```

Por último, compruebe cómo la función `IsLeadChar` de SysUtils se ha modificado para manejar pares subrogados Unicode, así como otras funciones relacionadas con las utilizadas para pasar al siguiente carácter de una cadena y similares. Voy a utilizar algunas de estas funciones de las que trabajan con una cadena con un par subrogado en la sección ”UnicodeString y Unicode”.

Sobre String y UnicodeString

El cambio en la definición del tipo Char es importante porque está ligado con el cambio en la definición del tipo String. A diferencia de characters, strings se asigna a un nuevo tipo de datos que antes no existía, llamado UnicodeString. Como veremos, su representación interna es también, muy diferente de la del tipo *classic AnsiString*[29].

---

28 Punto de código Unicode $1D11E es el *símbolo musical clave de Sol.*

Como ya había un tipo WideString en el lenguaje, representando cadenas basadas sobre el tipo WideChar, ¿por qué molestarse con la definición de un nuevo tipo de datos? WideString no tenía (y sigue sin tener) un recuento de referencia y es extremadamente pobre en términos de rendimiento y flexibilidad (por ejemplo, utiliza el asignador de memoria global de Windows, antes que los nativos FastMM4).

Al igual que AnsiString, UnicodeString tiene recuento de referencia, utiliza semánticas copy-on-write y se resuelve bastante bien. A diferencia de AnsiString, UnicodeString utiliza dos bytes por carácter[30] y se basa en UTF-16.

El tipo string está ahora asignado al tipo UnicodeString en un código duro tal como lo está el tipo Char por las mismas razones. No hay ninguna directiva del compilador u otro truco para cambiar esto. Si tiene código que debe seguir utilizando el antiguo tipo de cadenas, basta con sustituir su declaración por una explícita del tipo AnsiString.

La estructura interna de los Strings

Uno de los principales cambios relacionados con el nuevo tipo UnicodeString es su representación interior. Esta nueva representación, sin embargo, es compartida por todos los tipos strings con recuento de referencias UnicodeString y AnsiString, pero no por los tipos de cadenas de referencia no contada, incluidos los tipos ShortString[31] y WideString.

La representación del tipo classic *AnsiString* fue la siguiente:

-8	-4	Dirección de referencia de la cadena
Ref count	length	Primer carácter de la cadena

29 Estoy usando el término específico del tipo *classic AnsiString*, para referirme a la cadena con el tipo utilizado para trabajar a partir de Delphi 2 hasta Delphi 2007. El tipo AnsiString sigue siendo parte de Delphi 2009, pero tiene un comportamiento modificado, de modo que al referirse a su estructura anterior usaré el término *classic AnsiString*.

30 En realidad UTF-16 es una codificación de longitud variable, y en ocasiones UnicodeString utiliza dos elementos WideChar subrogados (es decir, cuatro bytes) para representar a un único punto de código Unicode

El primer elemento (contando hacia atrás desde el comienzo de la misma cadena) es la longitud de la cadena Pascal, el segundo elemento es el recuento de referencia. En Delphi 2009, la representación de cadenas de referencia-contada se convierte en:

-12	-10	-8	-4	Dirección de referencia de la cadena
Code page	Elem size	Ref count	length	Primer carácter de la cadena

Al lado de la longitud y la cuenta de referencia, los nuevos campos representan el tamaño de cada elemento y la página de código. Si bien el tamaño del elemento se utiliza para discriminar entre AnsiString y UnicodeString, el código de página tiene sentido en particular para el tipo AnsiString (ya que funciona en Delphi 2009), mientras que el tipo UnicodeString tiene el código de página fijo 1200.

El soporte correspondiente a la estructura de datos se declara en la sección implementation[32] de la unidad System como:

```
type
  PStrRec = ^StrRec;
  StrRec = packed record
    codePage: Word;
    elemSize: Word;
    refCnt: Longint;
    length: Longint;
  end;
```

Con la sobrecarga de una cadena pasando de 8 bytes a 12 bytes, uno podría preguntarse si una representación más compacta no sería más eficaz, aunque los nuevos campos son más compactos que los tradicionales (que podrían cambiarse sólo a expensas de la compatibilidad).

Este es un clásico estira-y-afloja entre la memoria y la velocidad: mediante el almacenamiento de datos en diferentes zonas de la memoria (y no utilizando partes de una única ubicación) debemos ganar un extra en

31 ShortString es el nombre de la tradicional cadena tipo Pascal, una cadena de AnsiChar limitada de 255 caracteres, ya que utiliza una longitud de byte como primer elemento. El tipo ShortString era la definición original de la cadena en Delphi 1. Desde que Delphi 2 introdujo referencias largas de cadenas contadas, el uso de ShortString ha disminuido, pero hay casos específicos en los que son agradables de usar y se ejecutan mejor.

velocidad de ejecución, aunque represente un costo extra de memoria por cada cadena que creemos.

Si bien en el pasado se tenían que utilizar punteros basados en código de bajo nivel para el acceso a la cuenta de referencia, la RTL de Delphi 2009 añade algunas funciones a mano para acceder a las distintas cadenas de metadatos:

```
function StringElementSize(const S: UnicodeString): Word;
function StringCodePage(const S: UnicodeString): Word;
function StringRefCount(const S: UnicodeString): Longint;
```

Hay también una nueva función de ayuda en la unidad SysUtils, llamada `ByteLength`, que devuelve el tamaño de un UnicodeString en bytes haciendo caso omiso de los atributos `StringElementSize` (así, curiosamente, no va a trabajar con cadenas de tipos distintos de UnicodeString).

Como ejemplo, puede crear una cadena y pedir alguna información acerca de esta, como ya hice en el ejemplo StringTest:

```
var
  str1: string;
begin
  str1 := 'foo';
  Memo1.Lines.Add ('SizeOf: ' + IntToStr (SizeOf (str1)));
  Memo1.Lines.Add ('Length: ' + IntToStr (Length (str1)));
  Memo1.Lines.Add ('StringElementSize: ' +
    IntToStr (StringElementSize (str1)));
  Memo1.Lines.Add ('StringRefCount: ' +
    IntToStr (StringRefCount (str1)));
  Memo1.Lines.Add ('StringCodePage: ' +
    IntToStr (StringCodePage (str1)));
  if StringCodePage (str1) = DefaultUnicodeCodePage then
    Memo1.Lines.Add ('Is Unicode');
  Memo1.Lines.Add ('Size in bytes: ' +
    IntToStr (Length (str1) * StringElementSize (str1)));
  Memo1.Lines.Add ('ByteLength: ' +
    IntToStr (ByteLength (str1)));
```

Este programa produce un resultado similar al siguiente:

```
SizeOf: 4
Length: 3
```

---

32 Como la sección de implementación no se puede utilizar en nuestro código, lo cual es comprensible para el mantenimiento de la estructura interna de datos, cuya implementación es susceptible a futuros cambios. Hay funciones de ayuda para acceder a la información que generalmente necesitamos usar.

```
StringElementSize: 2
StringRefCount: -1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

El código de página devuelto por un UnicodeString es 1200, este número se almacena en la variable global `DefaultUnicodeCodePage`. En el código anterior (y en su salida) se puede observar claramente que no hay una llamada directa a fin de determinar la longitud de una cadena de bytes, dónde `Length` devuelve el número de caracteres.

Por supuesto, usted puede (en general) multiplicar este por el tamaño en bytes de cada carácter, mediante la siguiente fórmula:

```
Length (str1) * StringElementSize (str1)
```

No sólo puede preguntar por la información de una cadena, pues también puede cambiar algo de la misma. Una manera de convertir una cadena de bajo-nivel es llamar al procedimiento `SetCodePage` (una operación que sólo se pueden aplicar a un tipo `RawByteString` type, como veremos), que sólo puede ajustar el código a la página real o realizar una conversión completa de la cadena. Voy a utilizar este procedimiento en la sección "Conversión de Cadenas".

UnicodeString y Unicode

Huelga decir que el nuevo tipo string (o el nuevo tipo UnicodeString, para ser más preciso) utiliza el conjunto de caracteres Unicode. Si ha leído el anterior capítulo, la pregunta sería, “¿Qué sabor de Unicode?”

No debería de sorprendernos conocer que los nuevos tipos de cadena usan, como ya he mencionado, UTF-16[33]. Esto tiene mucho sentido por muchas razones, la más importante es que este es el tipo nativo de cadena gestionado por la API de Windows en las recientes versiones de este sistema operativo.

Como hemos visto en la sección correspondiente al tipo WideChar en Delphi 2009, la nueva clase de apoyo `TCharacter` (no sólo se utiliza para

33 Más precisamente, el tipo de UnicodeString almacenado en la memoria como un UTF-16 con una cadena de representación little endian, o UTF-16 LE.

WideChar, sino también para procesar UnicodeString) tiene pleno soporte de UTF-16 y de pares subrogados. Lo que no mencioné en esa sección es que esto tiene el efecto secundario sensible de hacer el número de elementos de una cadena WideChar diferente del número de puntos de código Unicode que esta contiene, ya que un único punto de código Unicode puede ser representado por un par subrogado (es decir dos WideChar).

Una forma de crear una cadena con pares subrogados es utilizar la función `ConvertFromUtf32`[34] que devuelve una cadena con el sustituto par (dos WideChar) en las circunstancias propias, como las siguientes:

```
var
  str1: string;
begin
  str1 := 'Surr.  ' + ConvertFromUtf32($1D11E);
```

Si pregunta ahora por la longitud de la cadena, obtendrá 8, que es el número de WideChar, pero no el número lógico de puntos de código Unicode en la cadena. Si imprime la cadena consigue el efecto adecuado (bueno, al menos Windows en general, muestra un bloque cuadrado como sustituto de la pareja, en lugar de dos).

Como se ha demostrado con el método de `btnSurrogateClick` del formulario principal del ejemplo StringTest, calcular el número lógico de puntos de código Unicode podría ser más complejo de lo que usted esperaba:

```
n := CharToByteLen (str1, Length (str1) - 1 );
CountChars (str1, Length (str1), cChar, cByte);
n := cChar – 1;
```

Una cuestión relacionada es lo que ocurre cuando se reproduce un bucle para cada carácter de la cadena. Lo normal de un bucle con ciclo `for-in` es que tan sólo le permita trabajar en cada elemento WideChar de la cadena, no en cada punto de código lógico Unicode. Entonces debería usar el bucle while basado en la función `NextCharIndex` o adaptar el bucle para que compruebe los subrogados:

```
if TCharacter.IsHighSurrogate (str1 [I]) then
  Memo1.Lines.Add (str1 [I] + str1 [I+1])
```

34 En el código de `ConvertFromUtf32` (o más precisamente en el método de clase `ConvertFromUtf32` que llama a la clase `Tcharacter`) se puede ver el algoritmo actual usado para asignar puntos de código Unicode en pares subrogados. Interesante lectura si está interesado en los detalles.

La lista completa para ambos casos está disponible en el mismo ejemplo StringTest, y que no listo aquí, como en la mayoría de los casos, se puede asumir que trabaja con el BMP (Basic Multilingual Plane) que trata a cada WideChar de la cadena Unicode como un único punto de código[35].

El Tipo UCS4String

Existe también otro tipo de cadena que puede utilizar para manejar una serie de puntos de código Unicode, el tipo UCS4String. Este tipo de datos representa una matriz dinámica de caracteres de 4-bytes (del tipo UCS4Char). Como tal, no tiene ningún recuento de referencia o apoyo copy-on-write, y muy poco soporte en la RTL.

Aunque este tipo de datos (que ya estaba disponible en Delphi 2007) puede ser utilizado en situaciones específicas, no es especialmente adecuado para las circunstancias generales. Sin duda puede ser un residuo en la memoria, ya que no es sólo cadenas utilizando 4 bytes por carácter, sino que además puede terminar con múltiples copias en la memoria.

Los múltiples tipos de cadenas

Junto con la introducción del nuevo tipo UnicodeString, la actualización en la representación interna compartida por todos los tipos de cadenas (incluido el tipo AnsiString) deja espacio para algunas mejoras adicionales en la gestión del string. El equipo I+D de Delphi tomó ventaja de esta nueva representación interna (y todo el trabajo se realizó a nivel del compilador para mejorar la gestión de la cadena) para podernos proporcionar actualmente múltiples tipos de datos e incluso una nueva definición del mecanismo de tipo string.

La tipos predefinidos de cadenas con recuento de referencia[36], además de UnicodeString, son los siguientes:

---

35 El hecho de que dos puntos de código Unicode pueda ser visualizado como un único grafema (véase la sección de "puntos de código Unicode y Graphemes" en el Capítulo 1) hace que sea aún más difícil mapear el número de WideChar en una cadena Unicode con el número de caracteres de la pantalla.

- **AnsiString** un tipo cadena de único-byte-por-carácter basado en la actual página de códigos del sistema operativo, sigue muy de cerca del *AnsiString* clásico de versiones anteriores de Delphi;
- **UTF8String** es una cadena basada en la variable de caracteres de formato UTF-8;
- **RawByteString** es una matriz de caracteres sin página de códigos establecidos, sobre la que la conversión de caracteres no se logra por el sistema (por lo tanto, en parte se asemeja a la clásica *AnsiString*, cuando se utiliza como una matriz pura de caracteres) .[37]

La definición de tipo de mecanismo se revela cuando se observa en la definición de estos nuevos tipos de cadenas:

```
type
  UTF8String = type AnsiString(65001);
  RawByteString = type AnsiString($FFFF);
```

En esta sección voy a cubrir el AnsiString y los tipos personalizados de cadenas y, a continuación, el tipo UTF8String. Voy a centrarme en RawByteString en la siguiente sección que abarca conversiones de cadenas, tal como usted utilizará generalmente este tipo de cadenas para evitar las conversiones.

El Nuevo Tipo AnsiString

A diferencia del pasado, el nuevo tipo de cadena AnsiString lleva una más pieza de información, la página de códigos de los caracteres de la cadena. El `DefaultSystemCodePage` cuya variable por defecto es `CP_ACP`, actual código de la página en Windows, pero puede ser modificado por llamar al procedimiento especial `SetMultiByteConversionCodePage`. Esto se puede hacer para obligar a todo un programa a trabajar (por defecto) con caracteres de una determinada página de códigos (que la instalación del sistema operativo debe soportar, por supuesto).

---

36 Esto excluye los tipos de cadenas que no son por referencia, entre los que se incluyen los tipos ShortString, WideString, y UCS4String.

37 Con matrices de bytes disponibles (tal como se contempla en el capítulo siguiente) usted debe tratar de pasar a la construcción más específica, aunque RawByteString le permitirá migrar los datos de código con menos esfuerzo.

En general, en cambio, usted se atendría a la página de código corriente, o la cambiaría para cadenas individuales, llamando al procedimiento `SetCodePage` (presentado anteriormente, mientras he comentado sobre caracteres y páginas de códigos). Este procedimiento puede ser llamado de dos maneras diferentes. En el primer caso, sólo cambia el código de página para una cadena (cargada tal vez por un archivo separado o socket), ya que usted conoce su formato original. En el segundo caso, usted puede llamar la función para convertir una cadena dada (algo que ocurre automáticamente en la asignación de una cadena a otra con un códigos de página diferente, como se explica más adelante).

Aunque puede seguir utilizando el tipo AnsiString para tener una representación en memoria más compacta de las cadenas, en la mayoría de los casos usted realmente preferirá convertir su código para utilizar el nuevo tipo UnicodeString. Esto es, mantener sus cadenas declaradas con el string de tipo genérico. Sin embargo, hay circunstancias en que es necesario utilizar una cadena con tipo específico. Por ejemplo, cuando esté cargando o guardando archivos, transfiriendo los datos desde y hacia una base de datos, utilizando protocolos de Internet donde el código debe seguir estando basado en un formato de caracteres de 8 bits. En todos estos casos, convierta su código para utilizar AnsiString[38].

Creando una cadena de tipo personalizado

Además de utilizar el nuevo tipo AnsiString, que está vinculado por defecto a la página de códigos utilizada cuando compiló la aplicación, usted puede utilizar el mismo mecanismo para definir su propio tipo de cadena personalizada. Por ejemplo, puede definir un tipo de cadena Latin-1 (como lo he hecho en el ejemplo LatinTest) escribiendo:

```
type
  Latin1String = type AnsiString(28591);

procedure TFormLatinTest.btnNewTypeClick(
  Sender: TObject);
var
```

38 Más técnicas de importación se tratarán en el Capítulo 3.

```
  str1: Latin1String;
begin
  str1 := 'a string with an accent: Cantù';
  Log ('String: ' + str1);
```

Puede utilizar este tipo de cadenas como cualquier otro, pero estará vinculado a un determinado código de la página. Por lo tanto, si usted usa este tipo de cadenas, al convertir un `Latin1String` en un UnicodeString (por ejemplo, para mostrarla en la llamada `Log` anterior), el compilador Delphi añadirá una conversión más. La última línea del fragmento de código, ha *ocultado* un llamamiento a `_UStrFromLStr`, que termina llamando a más funciones internas de la unidad System, hasta la operación de conversión real realizada por el `MultiByteToWideChar` de la API de Windows. Esta es la secuencia de llamadas[39]:

```
procedure _UStrFromLStr(var Dest: UnicodeString;
  const Source: AnsiString);
procedure InternalUStrFromPCharLen(
  var Dest: UnicodeString; Source: PAnsiChar;
  Length: Integer; CodePage: Integer);
function WCharFromChar(WCharDest: PWideChar;
  DestChars: Integer; const CharSource: PAnsiChar;
  SrcBytes: Integer; CodePage: Integer): Integer;
function MultiByteToWideChar(CodePage, Flags: Integer;
  MBStr: PAnsiChar; MBCount: Integer;
  WCStr: PWideChar; WCCount: Integer): Integer; stdcall;
  external kernel name 'MultiByteToWideChar';
```

La API de Windows puede realizar las conversiones adecuadas, pero estas son potencialmente conversiones con pérdidas, ya que incluso algunos caracteres disponibles en las distintas páginas de código Windows no pueden estar representados en Latin1. Un ejemplo de esto sería el símbolo del Euro, otra simpática anécdota.

El método anterior `btnNewTypeClick` sigue mostrando algunos detalles más de la cadena:

```
  Log ('Last char: ' + IntToStr (
    Ord (str1[Length(str1)])));
  Log('ElemSize: ' + IntToStr (StringElementSize (str1)));
  Log('Length: ' + IntToStr (Length (str1)));
  Log ('CodePage: ' + IntToStr (StringCodePage (str1)));
```

39 Como veremos más adelante en la sección "Conversión de Cadenas" estas conversiones pueden considerablemente ralentizar las operaciones propias. Es por eso que el compilador emite advertencias similares en operaciones implícitas de conversión.

Ejecutar este código produce los siguientes resultados:

```
Last char: 249
ElemSize: 1
Length: 30
CodePage: 28591
```

Para demostrar que mi nueva cadena personalizada trata de manera diferente que el tipo estándar AnsiString (por lo menos en mi ordenador y con mi configuración local), he escrito un método de prueba en el proyecto LatinTest que agrega el mismo carácter final mayúscula (de `#128` a `#255`) a ambos, una AnsiString y un `Latin1String`, mostrándose en grupos en un Memo:

```
procedure TFormLatinTest.btnCompareCharSetClick(
  Sender: TObject);
var
  str1: Latin1String;
  str2: AnsiString;
  I: Integer;
begin
  for I := 128 to 255 do
  begin
    str1 := str1 + AnsiChar (I);
    str2 := str2 + AnsiChar (I);
  end;

  for I := 0 to 15 do
  begin
    Log (IntToStr (128 + I*8) + ' - ' +
      IntToStr (128 + I*8 + 7));
    Log ('Lati: ' + Copy (str1, 1 + i*8, 8));
    Log ('Ansi: ' + Copy (str2, 1 + i*8, 8));
  end;
end;
```

La parte inicial del resultado pone de relieve las diferencias entre los dos conjuntos (de nuevo, el resultado verá como puede variar en función de su propia ubicación):

```
128 - 135
Lati: ?,f".??
Ansi: €,ƒ„…†‡
136 - 143
Lati: ^?S<OZ
Ansi: ˆ‰Š‹ŒŽ
144 - 151
Lati: ''"".--
Ansi: ‘’“”•–—
152 - 159
```

```
Lati: ~Ts>ozY
Ansi: ˜™š›œžŸ
```

Dicho esto, al menos en mi latitud, un ejemplo mucho más interesante sería utilizar el código de página de un alfabeto no latino, como el Cirílico. Como un ejemplo, se define un segundo tipo de cadena personalizada en el proyecto LatinTest:

```
type
  CyrillicString = type Ansistring(1251);
```

Puede utilizar esta cadena en un modo muy similar al anterior fragmento de código, pero la parte interesante es utilizar el orden de caracteres altos, aquellos con un valor numérico de más de 127. He elegido unos cuantos con un bucle `for`:

```
procedure TFormLatinTest.btnCyrillicClick(
  Sender: TObject);
var
  str1: CyrillicString;
  I: Integer;
begin
  str1 := 'a string with an accent: Cantù';
  Log ('String: ' + str1);
  Log ('Last char: ' + IntToStr (
    Ord (str1[Length(str1)])));
  Log('ElemSize: ' + IntToStr (StringElementSize (str1)));
  Log('Length: ' + IntToStr (Length (str1)));
  Log ('CodePage: ' + IntToStr (StringCodePage (str1)));

  str1 := '';
  for I := 150 to 250 do
    str1 := str1 + CyrillicString(AnsiChar (I));
  Log ('High end chars: ' + str1);
end;
```

La salida de este método es la siguiente:

```
String: a string with an accent: Cantu
Last char: 117
ElemSize: 1
Length: 30
CodePage: 1251
High end chars: ––™љ›њќћџ ЎўЈ¤Ґ¦§Ё©Є«¬-
®Ї°±Ііґµ¶·ё№є»јЅѕїАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежз
ийклмнопрстуфхцчшщъ
```

Usted podrá comprobar que la letra acentuada se ha convertido en la correspondiente versión no acentuada, ya que el valor original no está

disponible[40]. La cadena constante es una cadena Unicode y la cesión a `str1` realiza una conversión implícita. De hecho, el valor numérico del último carácter es diferente.

También esta vez los caracteres de gama alta son completamente diferentes. Para obtener el efecto deseado, considere que tiene que escribir el doble moldeado:

```
CyrillicString(AnsiChar (I))
```

Si simplemente concatena los caracteres en la cadena para convertirlos después, serán tratados como caracteres Unicode.

Gestión de Cadenas UTF-8

Uno de los efectos secundarios de la nueva estructura interna de tipos de cadenas, es que ahora también podemos manejar cadenas en el formato UTF-8 de una forma más nativa. A diferencia del pasado, cuando UTF8String era simplemente un alias del tipo String, el nuevo tipo ya es plenamente reconocido: las conversiones son automáticas y todas las rutinas de manipulación existentes de cadenas UTF-8 se han importado para utilizar los nuevos tipos específicos.

Considere este código trivial (parte del ejemplo Utf8Test):

```
var
  str8: Utf8String;
  str16: string;
begin
  str8 := 'Cantù';
  Memo1.Lines.Add('UTF-8');
  Memo1.Lines.Add('Length: ' + IntToStr (Length (str8)));
  Memo1.Lines.Add('5: ' + IntToStr (Ord (str8[5])));
  Memo1.Lines.Add('6: ' + IntToStr (Ord (str8[6])));

  str16 := str8;
  Memo1.Lines.Add('UTF-16');
  Memo1.Lines.Add('Length: ' + IntToStr (Length (str16)));
  Memo1.Lines.Add('5: ' + IntToStr (Ord (str16[5])));
```

40 La `WideCharToMultiByte` tras la conversión trata de fallar graciosamente en algunas situaciones. Por ejemplo, degradar dobles comillas en comillas rectas en lugar de marcas dudosas y las letras acentuadas, cómo en el código de ejemplo, pierden su acento.

Como se podría esperar, la cadena `str8` tiene una longitud de 6 (es decir, 6 bytes), mientras que la cadena `str16` tiene una longitud de 5 (lo que, sin embargo, significa 10 bytes). Compruebe cómo invariablemente `Length` devuelve el número de elementos de la cadena, que en el caso de representaciones de longitud variable no coincide con el número de puntos de código Unicode representados por la cadena. Esta es la salida del programa:

```
UTF-8
Length: 6
5: 195
6: 185

UTF-16
Length: 5
5: 249
```

La razón es que, como vimos en el último capítulo, las cadenas UTF-8 utilizan una variable para implementar su longitud, de modo que los caracteres fuera de la espacio inicial de 7-bits ANSI que toman al menos dos caracteres. Este es el caso de las *u acentuadas* anteriormente expuestas. Asignando la misma cadena UTF-8 a una variable AnsiString, y ejecutando un código similar (de nuevo en el ejemplo Utf8Test), obtenemos lo siguiente:

```
ANSI
Length: 5
5: 249
```

Sin embargo esta vez la longitud de la cadena de 5 realmente significa 5 bytes y no sólo 5 caracteres.

El soporte del formato UTF-8 podría no ser tan completo como el de UTF-16, la implementación nativa de string en Delphi 2009, pero se ha mejorado mucho de manera muy significativa. Existen rutinas específicas para la manipulación de UTF-8 en la unidad WideStrUtils, y también todo el apoyo para el streaming de archivos de texto en este formato[41]. Lo que es básico, sin embargo, es el hecho de que, por ejemplo, puede trabajar en una cadena y mostrarla en cualquier control sin tener que realizar una conversión explícita (ni tener que recordar siempre y cuando llevarla a cabo), lo que sin duda es una gran ayuda.

---

41 Voy a cubrir la clase `TEncoding` y las conversiones de archivos de texto más adelante en este capítulo, en la sección “Streams y Codificaciones”.

Aunque algunas operaciones con cadenas UTF-8 puedan ser lentas, por las conversiones extras y desde las propias del tipo UnicodeString, teniendo un determinado tipo de datos en lugar de un alias sin tipo forzado por el compilador, es una gran diferencia para cualquier desarrollador de Delphi tener que hacer frente a esta codificación.

Usted también es muy libre de escribir su propia versión sobrecargada de las rutinas existentes (o unas nuevas), utilizando este tipo de cadenas para evitar cualquier conversión extra.

Conversión de Cadenas

Hemos visto que usted puede asignar un valor UnicodeString a un AnsiString, o a un UTF8String y la conversión se llevará siempre a cabo. Del mismo modo, cuando usted asigne un AnsiString, con un determinado código de página, a otro basado en un código de página diferente, aparece la conversión automáticamente. También puede convertir una cadena asignando una página de códigos diferente, preguntemos por la conversión a realizar:

```
type
  Latin1String = type AnsiString(28591);

procedure TFormStringConvert.btnLatin1Click(
  Sender: TObject);
var
  str1: AnsiString;
  str2: Latin1String;
  rbs: RawByteString;
begin
  str1 := 'any string with a €';
  str2 := str1;

  Memo1.Lines.Add (str1);
  Memo1.Lines.Add (IntToStr (Ord (str1[19])));

  Memo1.Lines.Add (str2);
  Memo1.Lines.Add (IntToStr (Ord (str2[19])));
  rbs := str1;
  SetCodePage(rbs, 28591, True);
  Memo1.Lines.Add (rbs);
  Memo1.Lines.Add (IntToStr (Ord (rbs[19])));
end;
```

En ambos casos anteriores, la conversión es una conversión con pérdida, ya que el símbolo Euro no puede estar representado en el código de página Latin-1. Observe la utilización de la rutina `SetCodePage`, que sólo puede aplicarse a un parámetro `RawByteString`, de ahí su asignación. Este es el resultado que obtendrá:

```
any string with a €
128
any string with a ?
63
any string with a ?
63
```

Conversiones Ralentizando Nuestro Código

Las conversiones automáticas, realizándose en segundo plano, son muy prácticas, ya que el sistema hace un montón de trabajo por nosotros, pero si no lo hace con cuidado, considerando lo que está haciendo, podría terminar con un código extremadamente lento, por las continuas conversiones y operaciones de copia de las cadenas. Considere la posibilidad del siguiente código (parte del ejemplo StringConvert):

```
  str1 := 'Marco ';
  str2 := 'Cantù ';
  for I := 1 to 10000 do
    str1 := str1 + str2;
```

Dependiendo del tipo real de cada una de las dos cadenas, el algoritmo puede ser extremadamente rápido o excesivamente lento. La demo utiliza string (que es UnicodeString) en una primera vuelta y una combinación de AnsiString y UTF8String (el peor caso posible, ya que tendrán que ser convertidas de ida y vuelta a UnicodeString para cada asignación) en un segundo. Este es el resultado de 10.000 iteraciones:

```
plain: 00.001
mixed: 01.717
```

Sí, usted está leyendo los números correctos, ¡este es el resultado para 1.000 veces o 3 ordenes de magnitud! Por si esto no fuera lo bastante malo, considere la posibilidad de lo que ocurre con 50.000 concatenaciones:

```
plain: 00:00.003
mixed: 00:42.879
```

¡Esto si es un incremento exponencial[42]! En otras palabras, una conversión implícita ocasional está muy bien, pero nunca debe dejar que ocurra dentro de ¡un bucle o rutina recursiva!

Lo que es importante saber, es que puede compilar su programa con las advertencias de conversión de cadena habilitadas (que es en realidad la opción por defecto), y ver dónde el compilador añade código de conversión. En esa sola línea de código, utilizada para concatenar strings de los diferentes tipos, obtendrá las siguientes advertencias:

```
W1057 Implicit string cast from 'UTF8String' to 'string'
W1057 Implicit string cast from 'AnsiString' to 'string'
W1058 Implicit string cast with potential data loss from
'string' to 'UTF8String'
```

El problema “potential data loss” (potencial pérdida de datos) surge porque no todas las cadenas pueden ser expresadas en todos los formatos. Por ejemplo, si usted asigna un UnicodeString a un AnsiString, existen posibilidades de que la operación no sea posible. Como las operaciones de conversión de cadenas son bastante comunes, las dos advertencias (‘Explicit string cast and Implicit string cast with potential data loss’) están activadas por defecto.

Con estas advertencias habilitadas verá muchas trampas potenciales, pero un programa medio puede tener muchas, e incluso alguna conversión explícita, que no serán eliminadas cambiando simplemente a un conjunto diferente de advertencias (‘Explicit string cast’ e ‘Implicit string cast’ con potencial pérdida de datos). ¡Desactive esta opción de advertencias cuando haya realizado la comprobación![43]

Una quinta advertencia similar surge a la hora de asignar una cadena constante a una cadena, en el caso de que algunos de los caracteres no puedan ser convertidos. La advertencia en este caso es ligeramente diferente:

```
[DCC Warning] StringConvertForm.pas(63): W2455 Narrowing
given wide string constant lost information
```

---

42 El aumento exponencial es debido al hecho de que cada vez más las grandes cadenas deben volver a ser asignadas en la memoria muchas veces. Lo qué se ve frenado por el código es parcialmente la conversión, pero sobre todo la necesidad de crear nuevas cadenas grandes temporales en lugar de seguir aumentando el tamaño de la actual.

Esta es una advertencia de la que usted debería deshacerse, ya que la operación no tendrá demasiado sentido.

Como otro ejemplo de conversión implícita (y un poco oculta) frenando la ejecución del programa, considere el siguiente fragmento de código:

```
str1 := 'Marco Cantù';
for I := 1 to MaxLoop2 do
  str1 := AnsiUpperCase (str1);
```

En los casos en los que la variable `str1` es un UnicodeString todo va bien, pero en los casos en que se trate de un AnsiString, esto provocará dos conversiones. Esto no es tan malo como en el caso anterior (porque aquí la cadena es corta y de todos modos se requiere una copia de esta), pero muestra un poco de sobrecarga (para un millón de iteraciones):

```
AnsiUpperCase (string): 00:00.289
AnsiUpperCase (AnsiString): 00:00.540
```

Las Llamadas Aseguradas

Otra serie de operaciones “ocultas” añadidas por el compilador son la familia de llamadas *Ensure String*, que añaden la comprobación del código de página a un parámetro string, y eventualmente desencadenar una conversión si no coincide. La más importante de estas llamadas es `EnsureUnicodeString`.

Como es muy difícil lograr una incorrecta situación en Delphi (excepto una conversión directa al tipo de cadena incorrecto), usted podría preguntarse por qué se han añadido. La razón es que cuando un UnicodeString es administrado por el código C++Builder, las cosas pueden salir mal.

Para asegurar las cosas, el compilador añade estos chequeos extras en diferentes lugares, sobre todo cuando se trabaja en una cadena pasada como parámetro. Mientras que este es el comportamiento predeterminado, usted puede compilar código específico (y librerías) deshabilitando esta opción, utilizando la opción del compilador `--string-checks` o la directiva

43 Estas “advertencias diagnósticas”, como el indicador y de seguridad de tipo introducidos para mantener la compatibilidad con .NET, pueden ser desactivadas por defecto. Incluso aunque yo sea un defensor de la regla “mantener todas las advertencias y tratar de compilar los programas sin insinuaciones y advertencias " convengo que las “advertencias diagnósticas” deberían ser tratadas de manera diferente.

`$STRINGCHECKS`. Teniendo en cuenta que usted tiene que utilizar por su propio riesgo, estos modificadores ya que no están documentados y no tienen soporte oficial (aunque, curiosamente, están directamente disponibles en las opciones de proyecto). Sin embargo, como los posibles problemas son muy limitados cuando se utiliza en Delphi (y no C++ Builder), probablemente pueda desactivarlo, como una técnica para ganar velocidad.

Por ejemplo, si escribe la siguiente línea:

```
function DoubleLengthUpperOn(
  const S: UnicodeString): Integer;
begin
  Result := Length(AnsiUppercase(S));
end;
```

y compila estas en una versión idéntica y con un nombre diferente, con la cadena de controles on y off, y después la llama dentro de un bucle de 10 millones de iteraciones, verá el siguiente cronómetro:

```
UpperOn: 00:02.202
UpperOff: 00:02.159
```

Esto parece muy modesto, y bastante irrelevante. Sin embargo, si usted optimiza el código con funciones ardientemente rápidas (por ejemplo, la eliminación de la llamada `AnsiUpperCase`), los resultados de 10 veces más iteraciones se convierten en:

```
On: 00:03.556
Off: 00:00.310
```

En este caso, la diferencia es significativa (diez veces más) y parece lineal con el número de iteraciones del bucle. Incluso si los casos similares son marginales, esta es probablemente una buena razón para mantener esta opción desactivada.

Para obtener más información sobre estos ajustes y sus efectos en términos de generar código ensamblador, puede consultar el siguiente blog de Jan Goyvaerts:

```
http://www.micro-isv.asia/2008/10/needless-string-
  checks-with-ensureunicodestring
```

Tenga cuidado con Literales en Concatenación

Hablando de la concatenación de cadenas, usted tiene que estar atento a las concatenaciones que engloben cadenas literales. Por ejemplo, debe considerar las aparentes siguientes líneas triviales de código:

```
Log ('String: ' + str1);
Log (str1 + ' is a string');
```

Ahora bien, si `str1` es un UnicodeString, no debería haber ningún problema en absoluto. Si `str1` es un AnsiString, en alguna de sus variantes, la concatenación con una cadena Unicode literal podría forzar diferentes conversiones dependiendo de si la cadena literal viene antes o después de la cadena variable. En la mayoría de los casos de cadenas mixtas con participación de literales, mi sugerencia sería añadir un tipo explícito emitido por la cadena, como en:

```
Log ('String: ' + UnicodeString(str1));
Log (UnicodeString(str1) + ' is a string');
```

Usando RawByteString

¿Qué pasa si usted necesita pasar un AnsiString como parámetro a una rutina? Cuándo el parámetro es asignado a una cadena con un tipo de codificación especifico, esta se convertirá al tipo adecuado, con la posibilidad de pérdida de datos. Esa es la razón por la que Delphi 2009 introduce un nuevo tipo de cadena personalizada, llamado RawByteString y que se define como:

```
type
  RawByteString = type AnsiString($ffff);
```

Esta es la definición de una cadena sin ningún tipo de codificación o, para ser más precisos, con el marcador de posición, indicando `$ffff` "*no encoding*". El RawByteString puede considerarse como una cadena de bytes, que ignora la codificación adjunta en el caso de una conversión automática a la hora de asignarse un AnsiString. En otras palabras, cuando pase un string de 1-byte por carácter como un parámetro RawByteString, no se realizará conversión alguna, a diferencia de cualquier otro tipo derivado de tipo AnsiString. Usted puede hacer una conversión llamando a la rutina

`SetCodePage`, como se ha demostrado anteriormente en la sección "Conversión de Cadenas".

Como tal, puede convertirse en un práctico reemplazo del tipo de string (o AnsiString) en código que utilice strings para el proceso de datos genérico y personalizado que usted desee mantener con 1-byte de representación por carácter[44].

Declarar variables de tipo RawByteString para almacenar una cadena real rara vez sucederá[45]. Teniendo en cuenta el código definido de la página, esto puede llevar a un comportamiento indefinido y a una potencial pérdida de datos. Por otra parte, si su objetivo es el ahorro de datos binarios utilizando una cadena como una asignación de memoria y su representación, puede utilizar el RawByteString de la misma forma que utilizó los AnsiString en las ultimas versiones de Delphi. Sustituyendo el código non-string que ha utilizado con el AnsiString por RawByteString este es un camino interesante de migración (como se verá en la sección "No Mueva Datos String" del capítulo 3).

Por ahora, vamos a centrarnos en un ejemplo típico en el que se puede utilizar como parámetro RawByteString. Si desea ver información sobre algunas cadenas de 8-bits, puede escribir cualquiera de las dos siguientes declaraciones (estos son los métodos del form principal de la demo RawTest):

```
  procedure DisplayStringData (str: AnsiString);
  procedure DisplayRawData (str: RawByteString);
```

El código de los dos métodos es idéntico (He aquí el listado de uno de los dos):

```
procedure TFormRawTest.DisplayRawData(
  str: RawByteString);
begin
  Log ('DisplayRawData(str: RawByteString)');
  Log ('String: ' + UnicodeString(str));
  Log ('CodePage: ' + IntToStr (StringCodePage (str)));
  Log ('Address: ' + IntToStr (Integer (Pointer (str))));
end;
```

---

44 No debe confundirse por este soporte ampliado para 1-byte por carácter de las cadenas compatibles ANSI: la solución preferida, es de lejos, migrar el procesamiento de código de su cadena al tipo UnicodeString. No se deje tentar demasiado por estos nuevos tipos de cadenas adicionales.

45 Para ver algunas consideraciones interesantes sobre RawByteString vaya al blog de Jan Goyvaerts http://www.micro-isv.asia/2008/08/using-rawbytestring-effectively/

Tenga en cuenta que las conversiones a UnicodeString se utilizan para mostrar el formato propio de la cadena, que son necesarias para evitar que los datos se traten como un AnsiString llano debido a la concatenación de una cadena literal con una cadena cuya página de códigos no esta definida en tiempo de compilación[46].

La razón por la que muestro la dirección de memoria de la cadena (al lado de su contenido y del código de página) es que esto nos permitirá determinar si la cadena ha sido convertida (y copiada), o si es exactamente la misma cadena pasada como parámetro.

Ahora podemos utilizar una variable AnsiString (no asignando simplemente una cadena constante sino haciendo algún proceso, o el resultado sería diferente) y pasarlo como un parámetro para los dos métodos, después de haber registrado alguna cadena de datos :

```
procedure TFormRawTest.btnRawAnsiClick(Sender: TObject);
var
  strAnsi: AnsiString;
begin
  strAnsi := 'Some text ';
  strAnsi := strAnsi + AnsiChar (210) + AnsiChar (128);

  Log ('String: ' + strAnsi);
  Log ('CodePage: ' + IntToStr (
    StringCodePage (strAnsi)));
  Log ('Address: ' + IntToStr (
   Integer (Pointer (strAnsi))));

  DisplayStringData (strAnsi);
  DisplayRawData (strAnsi);
end;
```

El resultado será como el esperado, ya que al llamar a `DisplayStringData` y al método `DisplayRawData` no se llevarán a cabo las conversiones y todas las operaciones se realizaran en la misma cadena:

```
String: Some text Ò€
CodePage: 1252
Address: 28149532

DisplayStringData(str: AnsiString)
String: Some text Ò€
CodePage: 1252
Address: 28149532
```

46 La utilización directa de `Log(str)` debería funcionar, ya que no existe concatenación implicada.

```
DisplayRawData(str: RawByteString)
String: Some text Ò€
CodePage: 1252
Address: 28149532
```

Si esto parece evidente, tal vez no sea tan claro lo que ocurre cuando pasamos una UTF8String como parámetro real a los métodos. La llamada del código es bastante similar, aunque convierto cada carácter tratándolo como un valor UTF-8 :

```
var
  strUtf8: UTF8String;
  nChar: Integer;
begin
  strUtf8 := 'Some text ';
  nChar := 210;
  strUtf8 := strUtf8 + UTF8String (AnsiChar (nChar));
  nChar := 128;
  strUtf8 := strUtf8 + UTF8String (AnsiChar (nChar));

  Log ('String: ' + strUtf8);
  Log ('CodePage: ' + IntToStr (
    StringCodePage (strUtf8)));
  Log ('Address: ' + IntToStr (
    Integer (Pointer (strUtf8))));

  DisplayStringData (strUtf8);
  DisplayRawData (strUtf8);
end;
```

En este caso, pasando la cadena como un AnsiString se realiza una conversión real (que es una conversión perdida, puesto que los caracteres no pueden ser representados por un AnsiChar), mientras la operación RawByteString procesa la cadena original directamente y produce la salida correcta:

```
UTF-8 string
String: Some text Ҁ
CodePage: 65001
Address: 28804892

DisplayStringData(str: AnsiString)
String: Some text ?
CodePage: 0
Address: 28804732

DisplayRawData(str: RawByteString)
String: Some text Ҁ
CodePage: 65001
```

```
Address: 28804892
```

En el programa usted puede ver más pruebas realizadas con tipos String definidos a medida. Al igual que con el testeo de UTF8String cada vez que pasa una cadena personalizada como un AnsiString tiene lugar una conversión, que está potencialmente pérdida, mientras que utilizando como parámetro RawByteString puede mantener la cadena en su valor original y mostrarlo correctamente. Aquí está una selección de las líneas de salida:

```
Latin string
String: Some text Ò

DisplayStringData(str: AnsiString)
String: Some text Ò?

DisplayRawData(str: RawByteString)
String: Some text Ò

Cyrillic string
String: Some text TФ

DisplayStringData(str: AnsiString)
String: Some text ??

DisplayRawData(str: RawByteString)
String: Some text TФ
```

Nuevas Funciones de Conversión UTF-8

Además de muchas transformaciones automáticas de cadena también hay varias nuevas funciones de conversión directa de cadena que puede utilizar al respecto. Por ejemplo, hay una gran número de nuevas funciones de conversión de y para UTF-8, sobrecargados para los diferentes tipos de cadenas:

```
function UTF8Encode(...): RawByteString;
function UTF8EncodeToShortString(...): ShortString;
function UTF8ToWideString(...): WideString;
function UTF8ToUnicodeString(...): UnicodeString;
function UTF8ToString(...): string;
```

Cadenas y Caracteres Literales

Hemos visto varios ejemplos en los que puede asignar un carácter literal individual o una cadena literal a cualquiera de los tipos de cadenas, con la conversión propia que tienen lugar detrás de estas escenas.

Las cadenas literales se consideran invariablemente del tipo UnicodeString. Tenga en cuenta que esto podría causar problemas con resoluciones que sobrecargan al pasar un cadena constante a una función como `Pos`, que ahora tiene varias versiones. En general, sin embargo, la gestión de las cadenas literales son bastante directas. Como vimos en el último capítulo, puede añadir cualquier carácter Unicode a una cadena constante en el editor, y todo funcionará sin problemas.

Los caracteres literales son la causa de algunas cuestiones más, particularmente por razones de compatibilidad hacia atrás. Los caracteres literales simples se convertirán en función de su contexto. Es más difícil para el compilador determinar qué hacer con hexadecimales (o decimales) de caracteres literales, como en el siguiente código tomado del ejemplo HighCharTest:

```
var
  str1: string;
begin
  str1 := #$80;
```

Por razones de compatibilidad hacia atrás, todos literales de cadena de 2 dígitos son analizados como AnsiChar por defecto, por lo que un desarrollador como yo que viva en Europa (o, más técnicamente, que tenga el mismo código de página establecido que yo) verá como el símbolo de moneda Euro se muestra en la cadena. En realidad, por la ejecución de la declaración:

```
  Log (str1 + ' - ' + IntToStr (Ord (str1[1])));
```

Obtengo la siguiente salida:

```
€ - 8364
```

En otras palabras, el literal es tratado como un AnsiChar y convertido al punto de código Unicode correcto. Si desea pasar plenamente a Unicode, puede ser que no le guste este comportamiento, ya que nunca se sabe cómo un determinado literal va a ser interpretado. Esa es la razón por la que Delphi presentó en 2009 una nueva directiva del compilador:

```
{$HIGHCHARUNICODE <ON|OFF>}
```

Esta directiva determina cómo los valores literales entre `#$80` y `#$FF` serán tratados por el compilador. Lo que se ha señalado anteriormente es el efecto de la opción por defecto (OFF). Si la activa, el mismo programa producirá este resultado:

```
 - 128
```

El número se interpreta como un punto de código real Unicode y su salida contendrá su carácter de control no imprimible. Otra opción para expresar este punto de código específico (o de cualquier punto de código Unicode por debajo de `#$FFFF`) es utilizar la notación de cuatro dígitos:

```
  str1 := #$0080;
```

Este no es interpretado como el símbolo de divisa Euro, independientemente de la configuración de la directiva `$HIGHCHARUNICODE`.

Lo que es agradable es que se puede utilizar la notación de cuatro dígitos para expresar caracteres del Lejano Oriente, al igual que los siguientes dos caracteres Japoneses:

```
  str1 := #$3042#$3044;
```

mostrado[47] como (junto con su representación numérica):

```
あい - 12354 - 12356
```

También puede utilizar literalmente los elementos `#$FFFF` que se convertirán en su propio par subrogado.

Por último, aviso que para los literales de cadena, el código de la página está tomado de las opciones del compilador, que puede usted modificar para un proyecto específico, y no a partir del sistema de código de la página del equipo en el que usted está compilando o ejecutando el programa.

---

47 あい se traduce por "reunión", según BabelFish, pero no estoy 100% seguro de dónde lo encontré inicialmente .

Streams y Codificaciones

Si en nuestra aplicación movemos todas nuestras cadenas a Unicode, cuando se trabaja con la RTL y la VCL, y mientras que la invocación de la API de Windows no sea difícil, las cosas pueden ser un poco más complicadas, tal como leer y escribir sus cadenas desde ficheros. ¿Qué sucede, por ejemplo, con las operaciones de archivos con `TStrings`?

Delphi 2009 introduce otra nueva clase de base para manejar archivos codificados, denominada `TEncoding` y como imitando algo a la clase `System.Text.Encoding` del .NET Framework. La clase `TEncoding`, que se define en la unidad SysUtils, tiene varias subclases que representan las codificaciones automáticamente soportadas por Delphi (estas son *codificaciones estándares* a las que puede añadir las suyas propias):

```
type
  TEncoding = class
    TMBCSEncoding = class(TEncoding)
      TUTF7Encoding = class(TMBCSEncoding)
        TUTF8Encoding = class(TUTF7Encoding)
    TUnicodeEncoding = class(TEncoding)
      TBigEndianUnicodeEncoding = class(TUnicodeEncoding)
```

La clase `TUnicodeEncoding` utiliza el mismo formato UTF-16 LE (Little Endian) usado por el tipo UnicodeString. Un objeto de cada una de estas clases está disponible en la clase `TEncoding`, como datos de clase, tiene su función getter correspondiente y su propiedad de clase (class property):

```
type
  TEncoding = class
    ...
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding
      read GetBigEndianUnicode;
    class property Default: TEncoding read GetDefault;
    class property Unicode: TEncoding read GetUnicode;
    class property UTF7: TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

La clase `TEncoding` tiene métodos para leer y escribir caracteres a byte streams, para llevar a cabo las conversiones, además de una función especial para manejar el BOM llamado `GetPreamble`. Por lo tanto, se puede escribir (en cualquier parte del código):

```
TEncoding.UTF8.GetPreamble
```

Streaming Listas de Cadenas

Los métodos `ReadFromFile` y `WriteToFile` de la clase `TStrings` pueden ser llamados con una codificación determinada. Si escribe una lista de cadenas de archivo de texto sin proporcionar una codificación específica, esta clase utilizará `TEncoding.Default`, que utiliza a su vez el `DefaultEncoding` interno extraído en la primera aparición de la actual página de códigos de Windows. En otras palabras, si guarda un archivo, obtendrá el mismo archivo ANSI que antes.

Por supuesto, usted también puede fácilmente forzar el fichero a un formato diferente, por ejemplo al formato UTF-16:

```
Memo1.Lines.SaveToFile('test.txt', TEncoding.Unicode);
```

Esto guarda el archivo con un BOM o preámbulo Unicode. Cuando realice la operación correspondiente `LoadFromFile` si no especifica una codificación, el método de carga va a terminar llamando a `GetBufferEncoding` método de la clase `TEncoding` que determinará la codificación en función de la presencia de un BOM (en el caso de su ausencia, se utilizará la codificación ANSI por defecto).

¿Qué pasa si usted especifica una codificación en `LoadFromFile`? La codificación que usted proporcione será utilizada para leer el fichero, independientemente del BOM que este contenga, a menudo produciendo un error. Yo preferiría esperar una excepción en caso de una discrepancia como esta, guardar un fichero con un código de página y forzar su carga con otro distinto, es sin duda un error del desarrollador. El no obtener una excepción puede ayudar en el caso que el archivo codificado se haya guardado sin un BOM, y todavía no se ha considerado como un archivo ASCII, pero si como un UTF.

Pero vamos a centrarnos en la operación de guardado del fichero. Si no cambia el actual código de Delphi, sus programas guardarán los archivos como ANSI. Si los programas importados no manejan datos Unicode, sus programas y sus ficheros serán completamente compatibles hacia atrás.

Pero y ¿si un programa no maneja datos Unicode? Supongamos que tenemos una StringList con las líneas escritas en diferentes idiomas[48], al

48 Estas líneas se han extraído de la web "¿Qué es Unicode?" del Consorcio de Unicode, que tiene su texto traducido en varios idiomas utilizando una variedad de alfabetos.

igual que en el siguiente formulario en tiempo de diseño del proyecto StreamEncoding:

Si tenemos código Delphi anterior que guarde listas de cadenas a un fichero y también lo recargue, es probable que tenga el siguiente aspecto:

```
procedure TFormStreamEncoding.btnPlainClick(
  Sender: TObject);
var
  strFileName: string;
begin
  strFileName := 'PlainText.txt';
  ListBox1.Items.SaveToFile(strFileName);
  ListBox1.Clear;
  ListBox1.Items.LoadFromFile(strFileName);
end;
```

Huelga decir que el efecto sería un desastre total, ya que sólo una fracción de los caracteres utilizados tienen una representación ANSI, por lo que terminaría con un montón de signos de interrogación en el objeto listbox.

Una simple alternativa sería cambiar este código como en el evento del segundo botón al pie del proyecto:

```
  strFileName := 'Utf8Text.txt';
  ListBox1.Items.SaveToFile(strFileName, TEncoding.UTF8);
```

De nuevo, no tenemos que especificar una codificación cuando se cargue la lista de cadenas, ya que Delphi lo tomará a partir del BOM.

Si prefiere guardar los datos como ANSI a menos que sea necesario, se puede comprobar el contenido de la lista de cadenas para determinar si guardarlos como ASCII ó UTF-8:

```
procedure TFormStreamEncoding.btnAsNeededClick(
  Sender: TObject);
var
  strFileName: string;
  encoding1: TEncoding;
begin
  strFileName := 'AsNeededText.txt';
  encoding1 := TEncoding.Default;

  if ListBox1.Items.Text <>
      UnicodeString (AnsiString(ListBox1.Items.Text)) then
    encoding1 := TEncoding.UTF8;

  ListBox1.Items.SaveToFile(strFileName, Encoding1);
```

Este código comprueba si se puede convertir una cadena a un AnsiString y devolverlo a un UnicodeString sin perder su contenido. Para una cadena muy larga, esta doble conversión y la comparación sería muy compleja, por lo que podría utilizar la alternativa de código siguiente (que no es tan precisa, ya que se basa en un código de página[49] específico, pero se acerca):

```
var
  ch: Char;
begin
  ...
  for ch in ListBox1.Items.Text do
    if Ord (ch) >= 256 then
    begin
      encoding1 := TEncoding.UTF8;
      break;
    end;
```

Usando un código similar usted puede decidir qué formato utilizar, en función de la situación. Aunque, podría ser una idea mejor, mover todos sus ficheros a codificación Unicode (UTF-8 ó UTF-16), independientemente de los datos reales. Usando UTF-16 generará archivos más grandes, pero también reducirá las conversiones al guardar y cargar.

---

49 Compruebe como `ch >= 256` no funciona si el código de la página por defecto es distinto que Windows 1252. Por ejemplo, "*Cantù*" no tiene ningún carácter >= 256, pero no puede ser representado en la página de códigos 1251.

Sin embargo, dado que no hay forma de especificar una conversión por omisión, pasar todos los archivos por la codificación Unicode significaría la necesidad de cambiar todos y cada una de las operaciones al guardar estos archivos... a menos que use un truco, como la modificación de la norma de comportamiento por defecto de la clase. Este *truco* podría concretarse en forma de un ayudante de clase helper[50]. Considere el siguiente código:

```
type
  TStringsHelper = class helper for TStrings
    procedure SaveToFile (const strFileName: string);
  end;

procedure TStringsHelper.SaveToFile(
  const strFileName: string);
begin
  inherited SaveToFile (strFileName, TEncoding.UTF8);
end;
```

Observe que la herencia aquí no significa llamar a una clase base, pero la clase será ayudada por su clase de ayuda. Ahora sólo tiene que escribir (o mantener su código cómo):

```
  ListBox1.Items.SaveToFile(strFileName);
```

para guardarlo como UTF-8 (o cualquier otra codificación de su elección). Encontrará este código en el ejemplo StreamEncoding.

Definiendo una codificación personalizada

Incluso si Delphi 2009 viene con algunas codificaciones preestablecidas, podría incluso terminar necesitando alguna más. Un ejemplo de codificación, no tan común, que puede necesitar es UTF-32 (little-endian). Definir y utilizar una codificación personalizada es, sin duda, posible, aunque hay algunas asperezas.

En primer lugar, usted tiene que definir una clase que herede de cualquier `TEncoding` o de uno de sus descendientes. Dado que no existen las clases para manejar la codificación de caracteres de 4-bytes, he comenzado por heredarlos de la clase base:

---

50 Si está interesado en aprender más acerca de los ayudantes de clase, una buena fuente es mi “Manual Delphi 2007”, pero seguramente puede encontrar otras referencias buscando en la web. El concepto de clase de ayuda es poco conocido pero es una característica muy potente de las últimas versiones Delphi.

```
type
  TUTF32Encoding = class (TEncoding)
  class var
    UTF32Encoding: TUTF32Encoding;
  strict protected
    function GetByteCount(Chars: PChar;
      CharCount: Integer): Integer; override;
    function GetBytes(Chars: PChar;
      CharCount: Integer; Bytes: PByte;
      ByteCount: Integer): Integer; override;
    function GetCharCount(Bytes: PByte;
      ByteCount: Integer): Integer; override;
    function GetChars(Bytes: PByte;
      ByteCount: Integer; Chars: PChar;
      CharCount: Integer): Integer; override;
  public
    function GetPreamble: TBytes; override;
    class function Encoding: TEncoding;
    function GetMaxByteCount(
      CharCount: Integer): Integer; override;
    function GetMaxCharCount(
      ByteCount: Integer): Integer; override;
  end;
```

Aquí hay básicamente dos métodos básicos de conversión (`GetBytes` y `GetChars`), métodos de recuento para cuatro caracteres/bytes, un método para definir el BOM o preámbulo, y una función de clase utilizada para devolver una instancia única (singleton), guardada en la variable de clase. Sólo los dos métodos de conversión son complejos, mientras que para todo lo demás todo lo que tiene que tener en cuenta es que usted toma 4 bytes, que es `SizeOf(UCS4Char)`, para cada carácter. Estos son los métodos, excepto para la conversión, que se describe más adelante con más detalle:

```
class function TUTF32Encoding.Encoding: TEncoding;
begin
  if not Assigned (UTF32Encoding) then
    UTF32Encoding := TUTF32Encoding.Create;
  Result := UTF32Encoding;
end;

function TUTF32Encoding.GetByteCount(
  Chars: PChar; CharCount: Integer): Integer;
begin
  Result := CharCount * SizeOf(UCS4Char);
end;

function TUTF32Encoding.GetCharCount(
  Bytes: PByte; ByteCount: Integer): Integer;
begin
```

```
  Result := ByteCount div SizeOf(UCS4Char);
end;

function TUTF32Encoding.GetMaxByteCount(
  CharCount: Integer): Integer;
begin
  Result := (CharCount + 1) * 4;
end;

function TUTF32Encoding.GetMaxCharCount(
  ByteCount: Integer): Integer;
begin
  Result := (ByteCount div 4) + (ByteCount and 1) + 1;
end;

function TUTF32Encoding.GetPreamble: TBytes;
begin
  // UTF-32, little-endian
  SetLength(Result, 4);
  Result[0] := $FF;
  Result[1] := $FE;
  Result[2] := $00;
  Result[3] := $00;
end;
```

El código imita un poco las clases de la RTL de Delphi, en particular, se asemeja a la clase `TUnicodeEncoding`. Los métodos de conversión son ligeramente más complicados. Dado que no quiero manejar los caracteres directamente, estoy usando las funciones la conversión de UnicodeString a UCS4String previstas por la RTL de Delphi. Extraer los bytes físicos para almacenarlos en el stream es una consecuencia de su conversión a UCS4String y mover los datos a bajo nivel:

```
function TUTF32Encoding.GetBytes(Chars: PChar;
  CharCount: Integer; Bytes: PByte;
  ByteCount: Integer): Integer;
var
  u4String: UCS4String;
begin
  Result := CharCount * SizeOf(UCS4Char);
  u4String := UnicodeStringToUCS4String (
    UnicodeString (Chars));
  Move(u4String[0], Bytes^, Result);
end;
```

Para la conversión contraria necesita mover los datos binarios al tipo UCS4String, transformarlos, y copiar el resultado en el buffer de salida:

```
function TUTF32Encoding.GetChars(Bytes: PByte;
  ByteCount: Integer; Chars: PChar;
```

```
  CharCount: Integer): Integer;
var
  u4String: UCS4String;
  uString: UnicodeString;
begin
  Result := CharCount;
  SetLength (u4String, Result);
  Move(Bytes^, u4String[0],
    CharCount * SizeOf(UCS4Char));
  uString := UCS4StringToUnicodeString (u4String);
  Move(uString[1], Chars^,
    CharCount * SizeOf(Char));
end;
```

Ahora, con este hábito de codificación disponible, simplemente puede escribir código como en el ejemplo CustomEncoding:

```
procedure TFormCustomEncoding.btnTestEncoding2Click(
  Sender: TObject);
begin
  Memo1.Lines.LoadFromFile ('Utf8Text.txt');
  Memo1.Lines.SaveToFile ('Utf32.txt',
    TUTF32Encoding.Encoding);
  Memo1.Lines.LoadFromFile ('Utf32.txt',
    TUTF32Encoding.Encoding);
end;
```

El único problema potencial es que no podemos simplemente llamar a `LoadFromFile` sin el método de codificación y esperar que la RTL de Delphi lo reconozca, ya que esto, simplemente, no sucederá[51]. En Delphi 2009 no hay manera de *instalar* nuestras costumbres de codificación en la RTL para que esta pueda reconocer su preámbulo con su código BOM-detection en el interior de la función de clase `TEncoding.GetBufferEncoding`. Esto se demuestra a través del último botón del ejemplo.

Unicode y la VCL

Disponer de soporte para cadenas Unicode en el lenguaje Delphi es emocionante, después de haber sido reasignada la API de Win32 a la versión Wide ofrece así un montón de migraciones fáciles, pero el cambio

51 Asimismo, tenga en cuenta que muy pocos editores reconocen el BOM de la UTF-32 y su codificación.

fundamental es que toda la RTL y la Visual Component Library (VCL) están ahora plenamente habilitadas para Unicode. Todos los tipos strings (y las string lists) gestionados por los componentes están declarados como string, por lo que ahora coinciden con el nuevo tipo UnicodeString.

Algunas de las zonas internas de la RTL, de bajo-nivel, se basan sin embargo en diferentes formatos. Por ejemplo, los nombres de las propiedades se basan en UTF-8, por lo que es parte del soporte de la RTTI disponible en la unidad de TypInfo. Además de algunas excepciones muy específicas, todo lo demás se ha migrado a UnicodeString y a UTF-16.

El soporte para Unicode es un elemento clave, pero no la única característica que ayuda a mejorar el apoyo para la construcción de aplicaciones internacionales. Otras características se refieren a la utilización de BiDiMode y al soporte para la Traducción.

En cuanto a los archivos de código fuente, tenga en cuenta que puede guardarlos en cualquier formato que le guste, pero es necesario utilizar un formato Unicode si está utilizando cualquier code point por encima de 255 en su código fuente (para identificar nombres, cadenas, comentarios, o simplemente cualquier otra cosa). El editor le pedirá que use cada formato cuando sea necesario, aunque puede ir a los archivos de código fuente Unicode de todos modos.

¿Un núcleo creciente en la RTL?

Con todos los extras de proceso de cadenas y de gestión de código en tiempo de ejecución, ¿son los archivos ejecutables Delphi 2009 más grandes que en el pasado? He comparado el tamaño mínimo de una aplicación VCL compilado con los paquetes en tiempo de ejecución (MiniPack) y lo básico de un programa[52] (MiniSize), compilado en Delphi 2007 y Delphi 2009, obteniendo los siguientes resultados:

	Delphi 2007	Delphi 2009
MiniPack	15,872	16,896

52 Estos dos programas son parte de “Mastering Delphi 2005” y no he copiado el código fuente, ya que son bastante simples. Puedes descargarlos al igual que todo el código fuente del libro desde www.marcocantu.com/md2005

MiniSize	19,456	20,992

El "peso extra" es alrededor de 1 KB, una cantidad que aumenta si usted realiza conversiones de cadenas, pero debería seguir siendo bastante mínima la comparación con el tamaño de cualquier aplicación del mundo real.

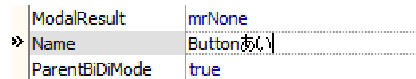
Unicode en archivos DFM

Acabo de mencionar la forma en que el IDE de Delphi puede tratar el Unicode habilitado para archivos de código fuente, pero no hemos visto lo que ocurre con los archivos DFM al añadir un carácter extendido a una de las propiedades. Un experimento muy simple será abrir un nuevo programa, colocar un botón en él, y pegar en el título del botón un carácter Unicode, como los caracteres Japoneses de la sección “Literales de Cadena”.

Visualizando el formulario como texto u observando en el fichero DFM actual verá el texto siguiente:

```
object Button1: TButton
  Left = 176
  Top = 104
  Width = 75
  Height = 25
  Caption = #12354#12356
  TabOrder = 0
end
```

Ahora añada otro formulario al programa (como lo hice en el programa DfmTest o modifique este mismo), y esta vez cambie el nombre del botón de añadir caracteres Unicode, así:



¿Cuál es el efecto sobre el DFM en este caso? Se guardará en el formato UTF-8 (junto con el archivo de PAS). Ábralo como texto y verá una diferencia dispar entre el `nombre` del componente y su `título`, que son coincidentes, pero utilizan diferentes representaciones:

```
object Buttonあい: TButton
  Left = 224
  Top = 112
```

```
    Width = 75
    Height = 25
    Caption = 'Button'#12354#12356
    TabOrder = 0
  end
```

En este caso, el archivo DFM no es compatible hacia atrás con Delphi 2007.

Localizando la VCL

Con el soporte a Unicode, el tradicional soporte de Delphi "bi-direccional mode ", o `BiDiMode`, reflejo de los controles en un formulario, y el Translation Manager, que forma parte del IDE, es cada vez más relevante.

No voy a cubrir `BiDiMode` en detalle, ni proveer orientación para el Internal y External Translation Managers, ya que estas herramientas no cambian comparadas con las versiones anteriores de Delphi. La arquitectura de traducción de la VCL son herramientas basadas en las que han estado en Delphi desde anteriores versiones, pero que han sido ciertamente mejoradas (y algunos de sus errores corregidos), ya que ahora están en el punto de mira.

A continuación

Ahora que ha visto cómo trata Delphi 2009 las cadenas Unicode, podemos centrarnos en la transición existente en el código fuente Object Pascal del mundo ASCII al mundo Unicode. Hay muchas cuestiones relacionadas, como PChar basado en un puntero matemático, una buena razón por la que se trata este tema en un capítulo separado.

La cobertura de las nuevas características relacionadas con cadenas, no termina aquí, en el capítulo 7 cubriré otras nuevas características de la RTL como la clase `TStringBuilder` y otras clases mejoradas para el tratamiento de texto.

Capítulo 3: Exportando A Unicode

Disponer de soporte nativo Unicode en Delphi es un gran paso adelante, y el hecho de poder continuar utilizando el tipo de string significa que usted puede importar código existente simplemente a costa de recompilar. Este cambio es un gran cambio. Desde las llamadas a la API de Windows empleando punteros `PChar` sólo con el soporte matemático del puntero, hay muchas áreas de Delphi para las que puede esperarse que nuestra migración no será tan fácil y sencilla. En este capítulo se profundiza en estos y otros problemas posibles.

Antes de sumergirse en este capítulo, sea consciente de que si usted necesita mantener la compilación de su código con versiones anteriores de Delphi, podrá emplear favorablemente la directiva del compilador UNICODE, que

está definida por el compilador de Delphi 2009. Podrá entonces escribir fragmentos de código que no compilen en versiones anteriores de Delphi escribiendo:

```
{$IFDEF UNICODE}
  // código espefícico para Delphi 2009
{$ENDIF}
```

Operaciones Char Erróneas

Como acabo de mencionar, la mayor parte de operaciones basadas en cadenas de caracteres se recompilan y migran sin problemas. Sin embargo, hay algunas que no y requerirán una solución en el código.

Una gran cantidad de código Pascal, Turbo Pascal, Object Pascal, y Objet Pascal de Delphi, asume que el tamaño de un carácter es un byte. Todo este código puede potencialmente fallar al pasar a Delphi 2009. Como veremos en la sección acerca de `FillChar` a continuación, para obtener el tamaño actual en bytes de una cadena se deberá siempre multiplicar la longitud de la cadena por el valor `StringElementSize`, ya que a menudo un carácter requiere de dos bytes (pero no siempre).

Cuidado con Set of Char

Ya he mencionado que no se puede declarar un `set of Char` nunca más, al menos no con el sentido tenía en versiones anteriores de Delphi, que tiene ahora un conjunto que incluye todos los caracteres posibles. Como se ha cubierto en el anterior capítulo, el compilador asumirá que está portando código existente para Delphi 2009 y decidirá considerar su declaración `set of Char` como si fuese `set of AnsiChar`, emitiendo una simple advertencia. Es posible que, si tiene código antiguo que utilice esta construcción, falle.

Puede ver un ejemplo de esta advertencia (y la solución explícita usada para removerla) en el ejemplo CharTest del capítulo 2. La verdadera cuestión en este caso es que no hay manera de definir un conjunto de todos los caracteres nunca más, o de expresar la inclusión de un carácter en un conjunto con este método de codificación. Realmente, ¡tiene que cambiar el

código completamente! Considere, por ejemplo, el sencillo código de la demo CharTest que acabo de mencionar:

```
var
  charSet: set of Char;
begin
  charSet := ['a', 'b', 'c'];
  if 'a' in charSet then ...
```

El enfoque alternativo es evitar totalmente usar un conjunto de caracteres y usar un algoritmo diferente, como:

```
var
  charSet: string;
begin
  charSet := 'abc';
  if Pos ('a', charSet) > 0 then ...
```

Esto tiene la ventaja de que también funciona cuando los caracteres no son ASCII, mientras que el uso de sets fija los límites a 256 valores en la comparación. De forma similar, con los ensayos para la inclusión de un carácter en un rango, como:

```
if ch1 in ['0'..'9'] then ...
```

que compila y sólo funciona gracias a la reducción de la variable ch1 a un *byte char* (como se anuncia en la advertencia), usted debe escribir su código cómo lo debería hacer en la mayoría de lenguajes de programación, como:

```
if (ch1 >= '0') and (ch1 <= '9') then ...
```

Todas estas técnicas tienen la ventaja de ser compatibles hacia atrás. Si necesita reemplazar su *is character in set* pruebe específicamente Delphi 2009, yo le recomendaría, sin duda, que utilice la nueva función específica `CharInSet` de la unidad SysUtils:

```
  if CharInSet ('a', charSet) then ...
  if CharInSet ('a', ['a', 'b', 'c']) then ...
```

Este código es casi idéntico a la prueba inicial, y es muy fácil de reemplazar las antiguas pruebas con las nuevas funciones cuando portamos el código a Delphi 2009. La función `CharInSet` se define para ambos valores AnsiChar y WideChar utiliza como segundo parámetro el tipo de conjunto definido a continuación:

```
type
  TSysCharSet = set of AnsiChar;

function CharInSet(C: AnsiChar;
  const CharSet: TSysCharSet): Boolean; overload; inline;
```

```
function CharInSet(C: WideChar;
  const CharSet: TSysCharSet): Boolean; overload; inline;
```

Otro enfoque alternativo que es bastante eficiente y trabaja a través de distintas versiones, es sustituir estas pruebas con una declaración como esta:

```
case Ch of
  'a'..'c': ...
end;
```

Este tiene también la ventaja de ser mucho más rápido que la función `CharInSet`. Un último caso específico es comprobar su inclusión en el conjunto `LeadBytes`:

```
if str[i] in LeadBytes then ...
```

En este caso debe usar la nueva función `IsLeadChar` para sustituir esta sentencia. En algunos casos, las comprobaciones Unicode relacionadas están proporcionadas por la unidad Character que puede serle de ayuda igualmente.

Evite FillChar para Caracteres

Aunque el procedimiento `FillChar` originalmente se destinaba a ser utilizado para el rellenado de una cadena con el mismo carácter muchas veces, también se (aún más comúnmente) utilizaba para llenar un buffer genérico con datos. El uso más frecuente es probablemente la reducción a cero de una estructura de datos, por lo que es imposible cambiar la definición actual del procedimiento (a pesar de su nombre):

```
var
  rc: TRect;
begin
  FillChar (rc, SizeOf (rc), 0);
```

Con strings cambiando a un tamaño de carácter de dos bytes surgen los problemas. El primero es porque la cadena es ahora el doble de grande, mientras que el segundo parámetro de `FillChar` se expresa en bytes, y no en el número de caracteres lógico. Por ello la primera de las dos operaciones `FillChar` en este fragmento de código (a partir del ejemplo CharTroubles) falla, mostrando con la posterior operación en pantalla una *t*:

```
var
```

```
  str1, str2: string;
begin
  str1 := 'here comes a string';
  str2 := 'here comes a string';

  FillChar (str1[1], Length (str1), 0); // nope!
  Memo1.Lines.Add ('15 char is: ' + str1[15]);

  FillChar (str2[1],
    Length (str2) * StringElementSize (str2), 0); // yes!
  Memo1.Lines.Add ('15 char is: ' + str2[15]);
```

¿Qué es peor?, piense lo que el rellenado con ceros del string hará, pero el rellenado con un carácter específico causará un completo desorden. De hecho, si usted rellena la cadena con la letra A, por ejemplo:

```
  FillChar (str2[1],
    Length (str2) * StringElementSize (str2), 'A');
  Memo1.Lines.Add ('15 char is: ' + str2[15]);
```

lo que termina haciendo es no llenar la cadena con el carácter `$41` pero si con el `$4141`, por lo que la cadena se convertirá en una secuencia de caracteres Chinos:

```
15 char is: 案
```

En otras palabras, puede absolutamente dejar de usar `FillChar` para rellenar una cadena con copias de un mismo carácter y seguir usando este procedimiento sólo para las estructuras de datos. Para rellenar una cadena puede utilizar en su lugar `StringOfChar`:

```
  str2 := StringOfChar ('A', 15);
  Memo1.Lines.Add (str2);
```

Si usted necesita un resultado AnsiString, sepa que hay una versión sobrecargada de `StringOfChar`, tomando un carácter AnsiChar. Considere, sin embargo, que si escribe:

```
var
  S: AnsiString;
begin
  S := StringOfChar('A', 15);
```

el compilador utilizará la versión de WideChar de `StringOfChar` y convertirá la UnicodeString resultante a un AnsiString. Para evitar esta conversión puede escribir:

```
S := StringOfChar(AnsiChar('A'), 15)
```

La simple razón es que el compilador no puede llamar a una función sobrecargada basándose en el tipo de resultado, lo hará sólo sobre la base de los parámetros de entrada.

Operaciones de Cadena Que Fallan o Ralentizan

Cuando se trabaja con cadenas, y en particular cuando se reutiliza código existente de procesamiento de cadena en Delphi 2009, hay dos cuestiones diferentes por las que debe estar atento y comprobar la salida. En primer lugar, usted tiene que comprobar si todas las operaciones producen el mismo resultado, ya que a veces no es el caso. En segundo lugar, usted tiene que estar seguro de que algunas operaciones no se conviertan en un proceso terriblemente lento.

La terrible ralentización sucede generalmente a causa de las conversiones implícitas de cadenas, y, en particular, cuando se intenta mantener las variables en torno a AnsiString. Si la declaración AnsiString original en su código fue para diferenciarla del tipo ShortString, y a continuación importamos el código, posiblemente su mejor opción sea utilizar el tipo string genérico.

Atendiendo a Todas las Alertas de Conversión de Cadenas

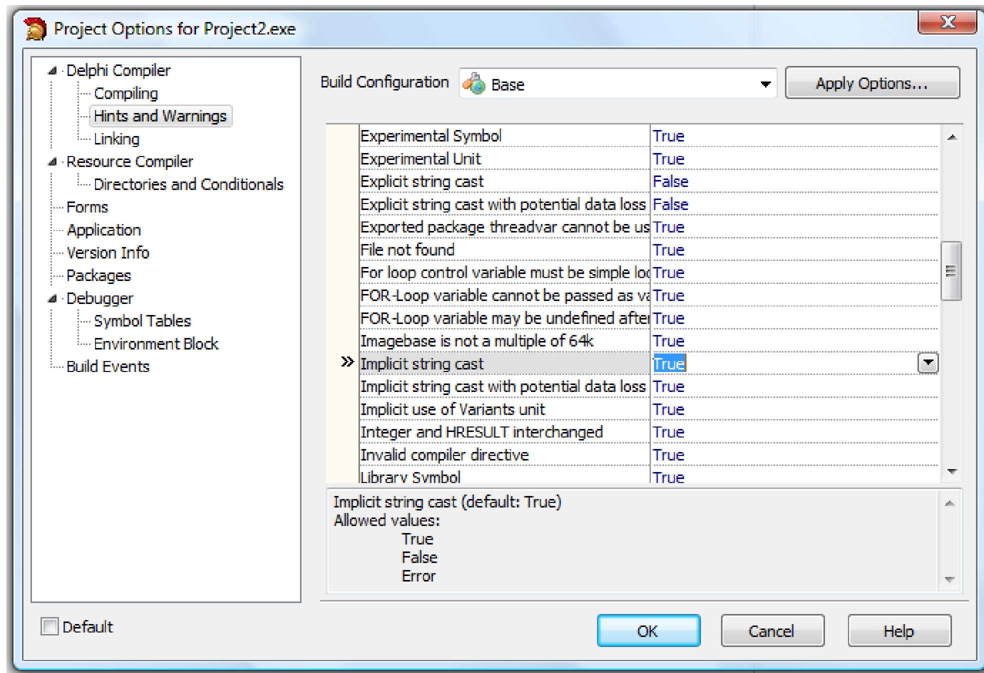
Si usted necesita mantener distintos tipos de cadenas o está importando código anterior en general, debería, por lo menos durante algún tiempo, atender las alertas de conversión implícitas de cadena, para tener una mejor comprensión de la forma en que el compilador interpreta el código y de las operaciones extras (posiblemente innecesarias) que este inyectará en su código. Como vimos en la sección “Asignando y Convirtiendo Cadenas” del capítulo 2, esto puede ralentizar el código del orden de unas mil veces.

Recuerde que en la actualidad hay varias advertencias de conversión de cadenas, algunas de los cuales no están habilitadas por defecto. Esta es una lista completa de las alertas relativas a string/character que vale la pena atender, al menos en la fase de conversión:

- Conversión explicita de cadena
- Conversión explicita de cadena con potencial pérdida de datos
- Conversión implícita de cadena
- Conversión implícita de cadena con potencial pérdida de datos
- Al reducir una constante de cadena wide/Unicode se puede perder información
- Al reducir una constante WideChar a AnsiChar se puede perder información
- WideChar reducido a byte char en una expresión de asignación
- Al ampliar una constante AnsiChar dada a un WideChar se puede perder información
- Al ampliar una constante AnsiString dada se puede perder información

Recuerde que también puede convertir alertas de conversiones de cadenas *especificas* en errores, al menos temporalmente, por lo que será más fácil capturar los mismos en caso de que su código esté causando demasiadas alertas y consejos[53]:

53 Puedes ampliar la información sobre el nuevo cuadro de diálogo opciones de proyecto y ajustes en el Capítulo 4, en la sección "Rediseñado el Diálogo de Opciones de Proyecto".



No Mueva Datos String

Acceder a datos string a bajo nivel, por ejemplo, con una llamada `Move` no era una muy buena idea anteriormente, ya que se podría perder la cuenta por referencia y causar problemas de sobrecarga de memoria y otros más. Llamar a la función `Move` con caracteres es aún peor ahora, ya que tenemos varias representaciones de cadena que son incompatibles en el nivel binario.

Como ejemplo, considere el siguiente código (tomado de la demo MoveStrings) que mueve los datos de una cadena a un buffer y los vuelve a cargar en otra cadena:

```
procedure TFormMoveStrings.btnMoveFailureClick(
  Sender: TObject);
var
  str1, str2: string;
  buffer: TBytes;
begin
  str1 := 'Hello world';

  SetLength (buffer, Length (str1));
  Move (str1[1], buffer[1], Length (str1));
```

```
  SetLength (str2, Length (buffer));
  Move (buffer[1], str2[1], Length (buffer));

  Log (str1 + ' becomes ' + str2);
end;
```

Como `Length` devuelve el número de caracteres en dos-bytes y `Move` trabaja en bytes, sólo la primera mitad de la cadena se copia en el buffer y y luego la segunda cadena con la parte principal de la cadena final rellena de cualquier tipo de dato que estuviera en la memoria no inicializada:

```
Hello world becomes Hello 圠湩潤we
```

Si tiene código existente que utiliza cadenas como búfers (como en el ejemplo anterior) y no quiere cambiar su código, una solución podría ser cambiar la definición del tipo de cadena a RawByteString (o, en menor medida, AnsiString). La misma demo MoveStrings tiene el mismo algoritmo con las cadenas declaradas como:

```
var
  str1, str2: RawByteString;
```

Esta versión del código producirá la salida correcta, ya que será mediante el uso de AnsiString. Dependiendo de las circunstancias (el significado real de los datos que traslade, string o búfer de datos), usted puede que preferir uno de estos tipos sobre el otro.

Aún mejor, siempre que sea posible, cambie su código utilizando una matriz dinámica de bytes como `TBytes`, la estructura de datos que he usado en el fragmento de código de esta sección para mantener un búfer genérico. El tipo `TBytes` se define en la unidad SysUtils como una matriz dinámica de bytes:

```
type
  TBytes = array of Byte;
```

Leyendo y Escribiendo Búfers

Cuando utilice operaciones puras con strings, su código anterior se importará a Delphi 2009 sin mayores obstáculos. Cuando usted guarde su

cadena de datos a archivos o a búfers de memoria, es mucho más fácil ver que las cosas van mal[54].

El siguiente es un ejemplo de un mal uso de streams en memoria, en el que los datos se extraen e insertan utilizando diferentes codificaciones, que no fuerzan una conversión, pero hace que el sistema considere los caracteres de un tipo diferente de lo que realmente son y se mezclen:

```
var
  memStr: TMemoryStream;
begin
  memStr := TMemoryStream.Create;
  try
    Memo1.Lines.
      LoadFromFile ('StreamTroubles_MainForm.pas');
    Memo1.Lines.SaveToStream(memStr, TEncoding.UTF8);
    memStr.Position := 0; // reset
    Memo2.Lines.LoadFromStream(memStr, TEncoding.Unicode);
  finally
    memStr.Free;
  end;
```

La ejecución de este código se traduce en un contenido totalmente ilegible para Memo2. Ahora bien, si en este fragmento de código revela este error bastante evidentemente, en la mayoría de situaciones del mundo real, puede caer en el mismo efecto pero de formas mucho más sutiles.

La recomendación número uno, siempre que tenga que guardar un archivo, es guardar asimismo el BOM, para que quede claro en qué formato está[55]. Esto no es difícil de lograr trabajando con la memoria, porque aunque usted no recuerde el formato actual, el código streaming de Delphi agrega el BOM correcto incluso a un memory stream.

Así, puede fijar el programa anterior llamando al método `LoadFromStream` sin codificación ninguna, dejando que el sistema compruebe por si mismo el formato declarado en el propio stream:

---

54 Espero que un desajuste del BOM en el archivo, con el preámbulo de codificación utilizado para su carga, plantee una excepción, ya que esto le daría un poco más de sentido. Por supuesto, siempre puede leer los bytes correspondientes, en lugar de utilizar strings, en caso de operaciones de bajo nivel que desee controlar plenamente

55 En el capítulo anterior, en la sección "Streams y Codificaciones" hemos visto cómo definir una clase de ayuda para la clase `TStrings` para cambiar la codificación predeterminada para el streaming. Vea este ejemplo como una manera de personalizar el comportamiento de su código existente sin tener que actualizar en muchos lugares.

```
var
  memStr: TMemoryStream;
begin
  memStr := TMemoryStream.Create;
  try
    Memo1.Lines.
      LoadFromFile ('StreamTroubles_MainForm.pas');
    Memo1.Lines.SaveToStream(memStr, TEncoding.UTF8);
    memStr.Position := 0; // reset
    Memo2.Lines.LoadFromStream(memStr);
  finally
    memStr.Free;
  end;
```

Añadiendo y Concatenando Cadenas

Otro tipo de codificación que debe sustituir siempre que sea posible, es la cadena de concatenación de código. No es demasiado difícil encontrar las apariciones de `AppendStr` en su código (también porque utilizándolo provocará una advertencia por obsoleto: deprecated warning), pero es más complicado el modo de sustituir la concatenación de cadenas directa realizada con el signo +.

También, la sugerencia es sustituir la concatenación de cadenas directa con el uso de la clase `TStringBuilder` en el caso que esté realizando una concatenación[56] compleja. Para mostrar un mensaje compuesto de dos cadenas, mantener el signo + es totalmente correcto.

Una vez más, lo que realmente debería velar es por la concatenación que cause conversiones implícitas de cadena, ya que esto significa que se copien los datos muchas, muchas veces. En otras palabras, añadir y concatenar cadenas funciona bien cuando todas las variables de cadena (left-value y right-values) son del mismo tipo (toda UnicodeString, toda AnsiString, toda UTF8String, etc.). En ese caso, el compilador utiliza el código de la página correcto para los strings literales.

---

56 A diferencia de lo que se ha escrito, utilizando la clase `TStringBuilder` no mejora específicamente el rendimiento, pero hace que el código sea más claro (en caso de conversiones complejas y concatenaciones) y si lo necesita, compatibilidad con .NET. La clase `TStringBuilder` se trata en detalle en el capítulo 7.

Cuando se mezclen cadenas con diferentes páginas de código, se harán conversiones explícitas para evitar que una conversión oculta ralentice el código (o cause pérdida de datos).

Strings son... Strings (no Bookmarks)

Aunque usted tiene el nuevo tipo RawByteString disponible para representar conjuntos de caracteres genéricos o bytes, mi recomendación es utilizar tipos de strings sólo para la tramitación de las cadenas de caracteres. Usted podría pensar esto es obvio, pero no lo es tanto.

Como ejemplo, la VCL ha utilizado largamente un tipo string para representar el tipo bookmarks de un dataset. La propiedad `Bookmark` de la clase `TDataSet` en Delphi 2007 es del tipo `TBookmarkStr`, que se define como:

```
type
  TBookmarkStr = string;
```

La cadena no es realmente lo que parece, es simplemente un alias a un puntero, pero con una extraña definición que hizo posible, como referencia de conteo, un mecanismo para marcadores de este tipo sin mayores costes.

Esta definición ha cambiado en Delphi 2009, causando potencialmente algunas incompatibilidades, siendo la propiedad `Bookmark` de la clase `TDataSet` ahora de un nuevo tipo, que se define como:

```
type
  TBookmark = TBytes;
```

La antigua propiedad del tipo `TBookmarkSt` sigue estando definida como un alias, pero ha quedado obsoleta (aunque no técnicamente marcada con la directiva `deprecated`), así que tendrá que cambiar su código para utilizar el tipo `TBookmark` en lugar del tipo `TBookmarkStr`. Cubriré los problemas relacionados con la importación de código que utiliza bookmarks a Delphi 2009 en el capítulo 12. Mi opinión principal aquí es que, si nunca hizo nada como esto, reescriba su código: no hay una forma mejor.

Molestas “Importaciones” Actuales

Mientras recompilaba cientos de aplicaciones de mis anteriores libros, choqué con unos pocos casos concretos en los que portar a Unicode

requeriría algunas actualizaciones reales y no una simple recompilación. Algunos de estos casos son realmente sólo errores triviales, y fáciles de solucionar in situ observando el código fuente, pero creo que es importante referenciarlos aquí, ya que su código podría utilizar técnicas similares. Yo podría haber construido interminables ejemplos “a propósito”, pero decidí utilizar casos reales, aunque posiblemente sean menos importantes.

InliningTest que usaba AnsiString

El siguiente fragmento de código proviene de Mastering Delphi 2005 y se actualizó posteriormente en Delphi 2007 Handbook[57]. Este demuestra la velocidad extra adquirida por el alineado asignado a la función, frente a una similar que no esté alineada, escrita de la siguiente forma:

```
const
  LoopCount = 100000000;

var
  ssample : string;

function LengthStdcall (const s: AnsiString): Longint;
begin
  Result := Integer(S);
  if Result <> 0 then
    Result := PInteger(Result-4)^;
end;

procedure TForm3.bntLenghtClick(Sender: TObject);
var
  ttt: TTimeTest;
  I, J: Integer;
begin
  ssample:= 'sample string';
  J := 0;
  ttt := TTimeTest.Create;
  try
    for I := 0 to LoopCount do
      Inc (J, LengthStdcall (ssample));
    memo1.Lines.Add ('Length ' +
      ttt.Elapsed + '[' + IntToStr (J) + ']');
  finally
    FreeAndNil (ttt);
```

57 El código fuente completo está disponible en la carpeta del proyecto dh2007_InliningTest

```
  end;
end;
```

El código todavía funciona, pero el tiempo registrado es ciertamente sospechoso:

```
Length 15,188[1300000013]
Inline 203[1300000013]
```

Los 15 segundos extra son necesarios para convertir el corto UnicodeString en un AnsiString cien millones de veces. Todo lo que tiene que hacer para solucionar este código y regresar al resultado esperado es cambiar la declaración de la función `LengthStdcall` a:

```
function LengthStdcall (const s: string): Longint;
```

Así vuelve a su tiempo esperado (aproximadamente):

```
Length 408[1300000013]
Inline 204[1300000013]
```

Utilizando Funciones Ansi-predeterminadas

En StrUtils y SysUtils solía haber una gran variedad de funciones y procedimientos con ANSI en su nombre. ¿Qué sucede con ellos en el Delphi Unicode? Debemos de eliminarlos, mantenerlos, ¿o qué?

No hay una respuesta única, pero en general pueden seguir tal como están implementadas aplicándose a las cadenas de tipo genérico, pero sería aún mejor renombrarlas para utilizar el mismo nombre, sin el prefijo Ansi.

En algunos casos, la actualización es automática. La mayor parte de las funciones de cadenas Ansi de la RTL (con o sin un prefijo Ansi) se han trasladado a la nueva unidad AnsiStrings y la mayoría de ellas tienen una versión sobrecargada sobre la base del tipo UnicodeString. Si usted no incluye esta unidad, acabará vinculando automáticamente[58] a la versión UnicodeString.

---

58 Si lo que desea es seguir usando el tipo AnsiString, y no desea añadir la unidad AnsiStrings en demasiados lugares, considere la posibilidad de utilizar una directiva de alias a la unidad, por ejemplo, la redefinición de SysUtils como SysUtils más AnsiStrings. En una línea de comandos sería:
`-AsysUtils=SysUtils;AnsiString`

De hecho, hay dos situaciones diferentes:

- Algunas de las funciones Ansi ahora trabajan internamente sólo con el tipo UnicodeString. Al llamarlas con un parámetro AnsiString, estas lo convertirán automáticamente.
- Algunas de las funciones Ansi tienen sus versiones AnsiString y UnicodeString sobrecargadas. Dependiendo de la cadena que pase como parámetro, está llamando a una u otra versión.

Una de las funciones en el primer grupo es `AnsiResemblesText`. Puede sustituir una llamada a `AnsiResemblesText` con una a `ResemblesText`, ya que ambas ruedan ahora sobre el tipo UnicodeString:

```
function ResemblesText(const AText, AOther: string):
  Boolean; overload;
function AnsiResemblesText(const AText, AOther: string):
  Boolean; overload;
```

Por supuesto, si su código necesita usar un AnsiString actual, incluso llamando a la versión Ansi de esta función no se ahorrará la conversión implícita de estas cadenas. La directiva de sobrecarga overload, en este caso, no se utiliza de manera efectiva, pero le permite añadir una versión de las funciones AnsiString en sus propias unidades.

Una de las funciones en el segundo grupo es `ReverseString`. En este caso, la versión Ansi de la función toma un parámetro AnsiString:

```
function ReverseString(const AText: string):
  string; overload;
function AnsiReverseString(const AText: AnsiString):
  AnsiString;
```

En esta situación, usted podría tener el problema opuesto, es decir si mantiene la llamada Ansi y utiliza el tipo UnicodeString, el compilador inyectará conversiones extras inútiles y, posiblemente pérdidas. En esta situación, usted tendría que actualizar su código para utilizar la versión no-Ansi de la función. En la situación opuesta (es decir, si utiliza `ReverseString` pasando AnsiString), puede añadir, además, una declaración para utilizar la unidad de AnsiStrings y así habilitar la tercera versión disponible de esta función:

```
function ReverseString(const AText: AnsiString):
  AnsiString; overload;
```

Para hacer las cosas aún más complicadas, hay funciones que tienen una versión Wide (de nuevo para mantener la compatibilidad con la unidad WideStrUtils) y algunas funciones que sólo tienen una versión Ansi.

En un buen número de casos, sin embargo, las unidades primordiales (SysUtils y StrUtils) sólo tienen versiones UnicodeString con las versiones AnsiString movidas ahora a la Unidad de AnsiStrings. Un buen ejemplo podría ser el de `UpperCase`[59] y sus funciones relacionadas:

```
// in SysUtils
function UpperCase(const S: string): string; overload;
function UpperCase(const S: string;
  LocaleOptions: TLocaleOptions): string;
  overload inline;
function AnsiUpperCase(const S: string):
  string; overload;
function WideUpperCase(const S: WideString): WideString;

// in AnsiStrings
function UpperCase(const S: AnsiString):
  AnsiString; overload;
function UpperCase(const S: AnsiString;
  LocaleOptions: TLocaleOptions): AnsiString;
  overload; inline;
function AnsiUpperCase(const S: AnsiString):
  AnsiString; overload;

// in WideStrUtils
function UTF8UpperCase(const S: UTF8string): UTF8string;
```

En general, por lo que veo, se aplican tres normas:

- Embarcadero trata de minimizar sus cambios en el código, de modo que usted pueda mantener, al menos, alguna llamada ANSI, ya que son sin perjuicio.
- Usted debe tratar de deshacerse de todas las llamadas ANSI (y llamadas Wide) en sus programas, aún a coste de emplear algún tiempo extra en la migración.
- En el caso de que quiera seguir usando el tipo AnsiString (lo que no está recomendado), utilice la nueva unidad AnsiStrings.

---

59 Aunque las funciones como `UpperCase` ahora toman un parámetro UnicodeString, todavía operan con caracteres ASCII solamente.

Cadenas Unicode y Win32

Como ya se ha mencionado en el capítulo anterior, la API de Win32 tiene en la mayoría de los casos llamadas separadas para cadenas ANSI (marcada con A) o cadenas Unicode (marcada con una W de Wide). Para ser más preciso, estas APIs utilizan cualquiera de los dos tipos `PAnsiChar` ó `PWideChar`. Ya he declarado que el hecho de que la API Win32 esté tan fuertemente basada en UTF-16 hace de este formato la elección más obvia para el desarrollo nativo en Windows, con herramientas como Delphi.

En muchos casos, las versiones ANSI de la API de Windows llaman a la versión Wide realizando una conversión extra. En otros casos, las APIs Wide son, de hecho, más lentas. El cambiar al mismo tiempo el formato string, el alias de tipo `PChar` y la versión de la API de Windows mapeada significa que puede convertir la mayor parte de su código de una forma muy sencilla: ¡Usted no tiene que hacer nada absolutamente! Considere el siguiente ejemplo:

```
TextOut (Canvas.Handle, 104, 224,
  PChar(str1), Length (str1));
```

Esto funciona por igual en Delphi 2007 y Delphi 2009, aunque la llamada al API de Windows termine siendo diferente. Esto es cierto para la mayoría de las llamadas a la API de Win32 con parámetros de cadena, que se han reestructurado de la versión A (ASCII) a la W (Wide), junto con un cambio en el compilador a nivel de `PChar` desde `PAnsiChar` a `PWideChar`.

Hay algunas API específicas, sin embargo, que no tienen dos versiones separadas y siempre exigen un puntero `PAnsiChar`. Un ejemplo típico nos lo da la función `GetProcAddress`, una DLL exportando símbolos limitados a Ansi. En estos casos, debe convertir ambos, la cadena y el puntero a otro tipo de caracteres distinto. Por lo tanto, una línea como:

```
GetProcAddress (hmodule, PChar (strFnName))
```

Se convertirá en:

```
GetProcAddress (hmodule, PAnsiChar (
  AnsiString(strFnName)));
```

Otro caso particular es el de la función `CreateProcessW`. Esta existe, pero puede modificar el contenido de la cadena con el nombre del archivo ejecutable y fallaría con una excepción, si usted pasa una cadena como valor constante.

Aplicaciones de consola Win32

Si está interesado en el desarrollo de aplicaciones de consola, tenga en cuenta que todos los Input/Output de Consola en Delphi aún se basan y seguirán basándose en la base ANSI. Esto es cierto en el caso de las rutinas como `Read`, `Write`, `ReadLn`, y `WriteLn`, entre otras. La ventana de consola muestra los caracteres utilizando código de página ANSI (o incluso un código de página OEM), y las mismas operaciones, cuando se redirijan a los archivos, podrían causar el mismo problema.

Verdaderamente, la consola de Windows se puede abrir en modo Unicode (si ejecuta con el comando */u* flag: `cmd /u`), pero esto se hace raras veces, ya que sólo funciona si su salida se envía a un archivo y no a la pantalla. Delphi 2009 no es compatible con el modo Unicode de la consola.

Si usted desea hacer experimentos, puede comenzar con la demo UnicodeConsoleTest en el que utilizo un objecto[60] `TTextWriter` con codificación Unicode, conectado con un stream asociado a su salida estándar. Mostrarlo en la pantalla de la consola es incorrecto, como esperaba. Al redirigir el resultado a un archivo se obtiene un archivo UTF-16, pero sin la marca BOM. Este es el código:

```
var
  aString: string;
  textWriter1: TTextWriter;
  fileStream1: TFileStream;

begin
  aString := 'My ten Euros (10€)';
  try
    fileStream1 := TFileStream.Create(
      GetStdHandle(STD_OUTPUT_HANDLE));
    textWriter1 := TStreamWriter.Create (
      fileStream1, TEncoding.Unicode);
    try
      textWriter1.Write (aString);
    finally
      textWriter1.Free;
      fileStream1.Free;
    end;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
```

60 La clase `TTextWriter` se trata en el capítulo 7.

```
  end;
end.
```

PChar y el Puntero Matemático

El tipo `PChar` en Delphi ha sido tradicionalmente utilizado en dos escenarios totalmente diferentes. El primero es la gestión de cadenas de caracteres de un modo compatible con el lenguaje C y la API de Windows. El segundo es para sustituir un puntero de tipo genérico, porque `PChar` era el único tipo de puntero con soporte a punteros matemáticos. Por ejemplo, puede moverse al siguiente carácter de una cadena escribiendo uno de los siguientes métodos:

```
var
  pCh1: PChar;
begin
  ...
  pCh1 := PChar1 + 1;
  Inc (PChar1);
```

No sólo puede aumentar un `PChar`, sino que también puede disminuirlo, compararlo con otro puntero, y hacer muchas otras operaciones.

El Problema Con PChar

Este uso de `PChar` fue tan útil que este tipo se utiliza a menudo en lugar de otros punteros, como `PInteger` en el siguiente fragmento de código extraído del ejemplo PointerMath, el cual lee una matriz utilizando un puntero (un puntero `PChar`) y moviendo el puntero desde un Integer hasta el siguiente sumándole 4 a este mismo (ya que un Integer es de cuatro bytes). Aquí tiene el código completo del método:

```
procedure TFormPointerMath.Button1Click(Sender: TObject);
var
  TenIntegers: array [1..10] of Integer;
  pOneInteger: PChar;
  I: Integer;
begin
  // escribir
  for I := 1 to 10 do
    TenIntegers [I] := I;
```

```
  // ahora leerlo usando un puntero
  pOneInteger := @TenIntegers;
  for I := 1 to 10 do
  begin
    Memo1.Lines.Add(
      'Address: ' + IntToHex (Integer(pOneInteger), 8) +
      ' - Value: ' + IntToStr (PInteger(pOneInteger)^));
    pOneInteger := pOneInteger + 4;
  end;
end;
```

Si compila este código en cualquier versión de Delphi desde Delphi 2 a Delphi 2007, obtendrá el resultado siguiente:

```
Address: 0012F4A8 - Value: 1
Address: 0012F4AC - Value: 2
Address: 0012F4B0 - Value: 3
Address: 0012F4B4 - Value: 4
Address: 0012F4B8 - Value: 5
Address: 0012F4BC - Value: 6
Address: 0012F4C0 - Value: 7
Address: 0012F4C4 - Value: 8
Address: 0012F4C8 - Value: 9
Address: 0012F4CC - Value: 10
```

Puede ver como la dirección se aumenta de 4 en 4 cada vez, y el valor correcto que devuelve. Tengo que subrayar, que si ahora recompila el mismo código exacto en Delphi 2009, usted obtendrá unos resultados totalmente diferentes:

```
Address: 0012F4AC - Value: 1
Address: 0012F4B4 - Value: 3
Address: 0012F4BC - Value: 5
Address: 0012F4C4 - Value: 7
Address: 0012F4CC - Value: 9
Address: 0012F4D4 - Value: 29043072
Address: 0012F4DC - Value: 4476177
Address: 0012F4E4 - Value: 4400843
Address: 0012F4EC - Value: 4403501
Address: 0012F4F4 - Value: 4474789
```

Esto no es lo que el código pretende, por supuesto, pero es lo que realmente manda este código. Incrementar 4 caracteres el puntero en Delphi 2009 significa que se desplazan 8 bytes hacia adelante, ya que cada carácter es ahora de dos bytes. No sólo la salida esta mal, sino pero también estamos haciendo un acceso ilegal a la memoria, que podría ser muy peligroso en los casos en que hubiéramos escrito en este área de memoria.

De PChar a PByte

Si este problema es potencialmente problemático, al menos para una secuencia de código de bajo nivel, la solución está al alcance de la mano. En el código anterior, usted puede simplemente sustituir la versión-específica del tipo `PChar` por la versión-agnóstica tipo `PByte`[61]. Un *puntero* a *byte*, de hecho, sigue siendo lo mismo y se comporta igual, independientemente del tamaño de los caracteres. Todo lo que tienes que hacer en un método similar al anterior es cambiar este puntero a una declaración de variables:

```
var
  pOneInteger: PByte;
```

Sin cambiar nada en el código, recompile el programa y este debería funcionar. Lo bueno es que (en Delphi 2009) `PByte` soporta el mismo puntero matemático que soporta `PChar`. En versiones anteriores de Delphi, `PByte` no soportaba el puntero matemático, pero usted todavía puede usar un algoritmo como el discutido, cambiando a *más uno* el ciclo de incremento:

```
Inc (pOneInteger, 4);
```

El hecho de que `Inc` y `Dec` trabajen con más tipos de punteros es poco conocido entre los usuarios de Delphi. Aunque, disponer del puntero matemático al completo significa que puede también comparar punteros, y realizar otras operaciones.

PInteger y la Directiva POINTERMATH

Sin embargo, como estamos tratando con números enteros, ¿no sería mejor escribir código como este (cambiando al incremento a uno y pasando del modelado del código original)?

```
procedure TFormPointerMath.btnPIntegerClick (
  Sender: TObject);
var
  TenIntegers: array [1..10] of Integer;
  pOneInteger: PInteger;
```

61 Una solución alternativa, que es más compatible con versiones anteriores de Delphi, es usar `PAnsiChar` en lugar de `PChar`. Sin embargo, el uso `PByte` se recomienda generalmente para hacer que su intención sea más clara y legible que usando `PansiChar`.

```
  I: Integer;
begin
  // escribir
  for I := 1 to 10 do
    TenIntegers [I] := I;

  // lee mediante un puntero
  pOneInteger := @TenIntegers;
  for I := 1 to 10 do
  begin
    Memo1.Lines.Add(
      'Address: ' + IntToHex (Integer(pOneInteger), 8) +
      ' - Value: ' + IntToStr (pOneInteger^));
    pOneInteger := POneInteger + 1;
  end;
end;
```

Una vez más, esto ha sido posible usando una llamada `Inc` incluso en Delphi 2007 (y que el ejemplo PointerMathD2007 pueda abrirse con esta versión del IDE, lo demuestra), pero en Delphi 2009 usted puede compilar el código anterior añadiendo al código fuente la directiva:

```
{$POINTERMATH ON}
```

No utilice PChar para Pointer Math

Para resumir esta sección, deje de utilizar `PChar` para cualquier cosa que no sea un carácter o relacionado con las cadenas. Si necesita que su código sea capaz de mantener la compilación con versiones anteriores de Delphi, puede utilizar `Inc` y `Dec`, y hacer algunos cambios en el código anterior. Si todo lo que necesita es el soporte de Delphi de 2009, convierta el código a `PByte` (generalmente es el camino más fácil) o use un tipo de puntero específico y la nueva directiva `POINTERMATH`.

En cualquier caso, ¡hacer una búsqueda de `PChar` en toda su base de código es generalmente una buena idea!

Parámetros Variants y Open Arrays

Cuando esté trabajando con variants, la mayoría del código de conversión de variants a cadenas funcionará tal como cabe esperar, ya que hay un nuevo tipo de variant:

```
varUString = $0102;
  { Unicode string 258 } {not OLE compatible}
```

Todas las conversiones basadas en variants deberían trabajar correctamente, sin muchas diferencias con su código variant relacionado, a menos que tenga que interactuar con COM o automatización OLE, en cuyo caso todavía tiene que utilizar el tipo WideString (al igual que antes, esto no cambia).

Cuando trabaje con parámetros variant open array, y otras estructuras de datos sin tipificar, las opciones AnsiString y UnicodeString deben ser manipuladas específicamente. Por ejemplo, la estructura `TVarRec` tiene ahora tres entradas distintas relacionadas con la cadena (solían ser dos), entre muchos otros tipos que he omitido:

```
type
  TVarRec = record
    case Byte of
      vtString:        (VString: PShortString);
      vtAnsiString:    (VAnsiString: Pointer);
      vtUnicodeString: (VUnicodeString: Pointer);
      ...
```

Si procesa un parámetro open array con una sentencia `case` que tiene ramas para cadenas específicas, tiene que considerar esta nueva alternativa.

A continuación

Acabamos de finalizar la cobertura al soporte de Unicode en Delphi 2009. A continuación, en la Parte II voy a tratar en profundidad los cambios en el IDE, el compilador y la RTL. En la sección RTL volveremos a estudiar las funciones de gestión de cadenas. Y más adelante, en la parte del libro dedicada a bases de datos, cubriré la forma en que los cambios introducidos

por el soporte Unicode afectan a la clase `TDataSet` y sus clases relacionadas.