EMBARCADERO
TECHNOLOGIES®

Миграция проектов Delphi для поддержки Unicode в новых версиях: практические рекомендации

Кэри Дженсен (Cary Jensen), Jensen Data Systems, Inc.

Декабрь 2009 г.

Embarcadero Technologies Россия, СНГ
129343 Россия, Москва, проезд Серебрякова, 6; тел.: +7 (495) 708-43-93
russia.info@embarcadero.com

Аннотация

Выпустив Embarcadero® RAD Studio 2010 (впрочем, начало было положено уже в RAD Studio 2009), компания Embarcadero Technologies предоставила разработчикам, работающим в среде Delphi® и C++Builder®, возможность создавать современные приложения с поддержкой Unicode. Это открывает новые рынки для вашего программного обеспечения, однако в некоторых случаях создает трудности с миграцией существующих проектов и применением методов разработки, особенно при наличии в них кода, построенного на определенных допущениях в отношении длины строк.

Настоящий документ вместил в себя идеи и опыт многочисленных разработчиков на Delphi, которым уже удалось осуществить миграцию проектов для поддержки Unicode. В начале представлено общее описание проблемы и дан краткий обзор основных понятий Unicode. Затем происходит систематическая проработка фрагментов приложений, требующих наибольшего внимания, с рекомендациями и примерами из реальной жизни. В конце документа вы найдете список ссылок на ресурсы, которые будут полезны в ходе миграции в новые версии Delphi с поддержкой Unicode.

Введение

Компания Embarcadero впервые реализовала полную поддержку Unicode в составе RAD Studio в августе 2008 года. Благодаря этому Delphi и C++Builder будут активно применяться для разработки приложений Windows на протяжении многих последующих лет.

Тем не менее, в отличие от других серьезных усовершенствований, появившихся в Delphi за последнее время, таких тип данных variant и интерфейсы (Delphi 3), фреймы (Delphi 5), inline-функции и вложенные классы (Delphi 2005), родовые типы generics (RAD Studio 2009), поддержка Unicode означает не просто добавление новых функций в состав Delphi. Она подразумевает радикальное изменение нескольких фундаментальных типов данных, присутствующих практически в любом приложении Delphi. В частности, изменилось определение типов String, Char и PChar.

К этим изменениям разработчики отнеслись со всей серьезностью. Поддержка Unicode была внедрена только после тщательного анализа возможных последствий для существующих и будущих проектов по разработке ПО. Кроме того, компания Embarcadero учла пожелания и рекомендации многих технологических партнеров, которые поддерживают и продвигают Delphi.

Реализация поддержки Unicode была неизбежно сопряжена с определенными неудобствами. Один из соавторов этого документа (он просил называть его просто Стив) сказал: «Я считаю, что смысловая нагрузка типов PChar и String не должна была измениться. <...> Иными словами, какой бы выбор ни сделали разработчики Delphi, это не уберегло бы их от критики. Ситуация была безвыходная».

В конце концов было решено, что изменение типов String, Char и PChar является наименее разрушительным путем, имеющим, впрочем, ряд последствий. С одной стороны, пользователи RAD Studio сразу получили возможность создавать первоклассные приложения, которые глобально интерпретируют и графические интерфейсы, и обрабатываемые данные. Это устранило ряд значительных преград для разработки и развертывания приложений в современных рыночных условиях.

Но нельзя забыть и о недостатках. Изменение типов String, Char и PChar привело к возникновению потенциальных проблем, связанных с миграцией приложений, библиотек, общих модулей и проверенных временем методов из более ранних версий Delphi/C++Builder.

Давайте будем реалистами. Практически при любом обновлении существующего проекта возможны проблемы с миграцией, которые требуют изменения кода либо внедрения сторонних компонентов и библиотек более новых версий. Это утверждение справедливо и в отношении перехода на RAD Studio 2009 или более поздней версии. В одних случаях обновление пройдет легче, в других труднее.

Вот мы и подошли к сути настоящего документа. Нужно честно сказать, что из-за изменения нескольких фундаментальных типов данных, которые использовались начиная с Delphi 1 (Char и PChar) или Delphi 2 (String), миграция существующего приложения в RAD Studio 2009 или более поздней версии требует больше усилий, чем любая предыдущая миграция.

Соавтор Роджер Конелл (Roger Connell) из компании Innova Solutions Pty Ltd выразился следующим образом: «[Специалисты компании Delphi,] по моему мнению, отлично справились с задачей [добавления поддержки Unicode, но] это была самая трудная (в принципе, единственная трудная) миграция Delphi». К счастью, для каждой проблемы, с которой вы столкнетесь, есть решение, и этот документ поможет вам его найти.

В самом начале данного проекта я обратился к сообществу RAD Studio. В частности, я попросил разработчиков, которым удалось успешно переместить свои приложения в RAD Studio 2009 или более поздней версии, поделиться идеями, рекомендациями и реальными историями миграции. Полученные ответы превзошли все мои ожидания.

На призыв отозвались разработчики практически всех возможных категорий. «Вольные стрелки» и коллективные труженики. Создатели отраслевых решений и внутрифирменных приложений, создатели популярных наборов компонентов и средств, которыми пользуются разработчики приложений. А также авторитетные специалисты по Delphi, которые выступают на конференциях и пишут книги, читаемые многими из нас.

Их истории, рекомендации и методики, естественно, разнились. Если одни рассказывали о довольно простых случаях миграции, то другим довелось стать участниками сложных проектов, как правило, касавшихся тех, которые были выпущены давно и включали в себя разнообразные технологии и решения.

Вне зависимости от сложности миграции можно выделить комплекс стандартных подходов, практических решений и потенциальных проблем, которыми я и хочу с вами поделиться.

Публикация настоящего документа не ставит точку в этом деле. Я буду и дальше собирать успешные истории миграции в формат Unicode, по необходимости обновляя содержимое документа. Таким образом, если документ вам понравился, и вы хотите дополнить его или развить одну из рассмотренных тем, то присоединяйтесь к коллективу соавторов. В конце я вернусь к этому вопросу более подробно.

В следующем разделе представлен краткий обзор описаний и определений Unicode. Если вы уже знакомы с Unicode, имеете базовое представление о форматах UTF-8 и UTF-16, а также понимаете разницу между кодовыми страницами и кодовыми точками, то пропустите этот раздел или быстро просмотрите его на предмет незнакомых терминов.

Прежде чем продолжить, хочу подчеркнуть следующее. Поддержка Unicode в RAD Studio имеет два дополняющих, но независимых друг от друга следствия для разрабатываемых вами приложений. Первое связано с различными способами обработки строк в программном коде, написанном в Delphi 2009 (или более поздних версий) и в ранних версиях Delphi. Второе относится к локализации, т. е. процессу адаптации программного обеспечения к языку и культуре определенной страны.

В настоящем документе рассмотрен только первый круг вопросов. Реализация поддержки нескольких языков и наборов символов выходит за рамки нашей дискуссии, и мы больше не будем возвращаться к этой теме.

Что такое Unicode?

Unicode — стандартная спецификация для кодирования всех знаков и символов всех существующих в мире письменных языков с целью хранения, извлечения и отображения компьютерами. Подобно набору символов стандарта ANSI (American National Standards Institute), который содержит как управляющие символы (табуляция, перевод строки, перевод страницы и т. д.), так и 26 печатаемых символов латинского алфавита, стандарт Unicode предусматривает как минимум один уникальный номер для каждого символа.

Кроме того, в Unicode, как и в ANSI, представлены символы многих других типов: например, обозначения валют, научные и математические, а также некоторые экзотические символы. Для реализации такого большого количества символов (в настоящее время их больше миллиона) в Unicode может использоваться до 4 байт (32 бита) данных. Для сравнения: стандарт ANSI основан на 8-битовом кодировании и потому не может содержать более 255 отдельных символов.

Каждому символу, управляющему символу или знаку в Unicode соответствует цифровое значение, которое называют *кодовой точкой*. Кодовая точка, назначенная символу Техническим комитетом Unicode, изменению не подлежит.

Например, символ «A» имеет кодовую точку 65 ($0041 в шестнадцатеричной системе счисления или U+0041 в Unicode). Каждому символу также назначается уникальное неизменяемое имя (в данном случае — «LATIN CAPITAL LETTER A»). Номер и имя никогда не меняются, что гарантирует неограниченный срок использования сегодняшней кодировки.

Каждая кодовая точка может быть представлена одним, двумя или четырьмя байтами; большая часть основных кодовых точек (64 КБ) имеет размер два байта или меньше. В Unicode эти первые 64 килобайта символов носят название *базовая многоязыковая плоскость* (basic multilingual plane, или BMP — запомните эту аббревиатуру, она еще не раз встретится в настоящем документе).

Несколько усложняет ситуацию то, что Unicode допускает использование некоторыми символами двух и более последовательных кодовых точек. Такие символы называют комбинированными, или разложимыми.

Например, символу «ö» соответствует код $00F6. Этот символ называется составным. Его также можно представить в виде символа «o» ($006F), за которым следует символ «трема» (¨) ($0308). Правила обработки Unicode объединяют эти два кода, создавая единый символ.

Это наглядно демонстрирует следующий фрагмент программного кода.

```
var
  s: string;
begin
  ListBox1.Items.Clear;
  s := #$00F6;
  ListBox1.Items.Add('ö');
  ListBox1.Items.Add(s);
  ListBox1.Items.Add((IntToStr(Ord('ö'))));
  s := #$006F + #$0308;
  ListBox1.Items.Add(s);
```

Комбинированные символы позволяют более точно анализировать содержимое файла Unicode. Например, чтобы посчитать частоту использования диакритического знака «трема» (¨), независимо от того, над каким символом он стоит, достаточно просто разложить все символы с таким знаком.

Все кодовые точки, которые назначены на сегодняшний день (и будут назначены в обозримом будущем), можно достоверно представить с помощью четырех байт данных, однако далеко не во всех случаях имеет смысл использовать для каждого символа столько памяти. Так, большинство людей, говорящих на английском языке, пользуется довольно небольшим набором символов (около 100 или даже меньше).

По этой причине Unicode дополнительно определяет несколько разных стандартов представления кодовых точек, каждый из которых является компромиссным с точки зрения единообразия, обработки и хранения. В Delphi вы чаще всего будете встречаться со стандартами UTF-8, UTF-16 и UTF-32. (UTF расшифровывается как Unicode Transformation Format или UCS Transformation Format, в зависимости от того, кого спросить.) Также иногда используется UCS-2 и UCS-4 (UCS означает Universal Character Set).

Формат UTF-8 хранит кодовые точки в одном, двух, трех или четырех байтах памяти в зависимости от размера целого числа, представляющего кодовую точку. Это предпочтительный формат для стандартов, где размер имеет большое значение, таких как HTML и XML. В частности, символы (например, латинский алфавит), которые могут быть представлены одним байтом и на которые приходится значительная часть HTML-кода (по крайней мере, на большинстве веб-страниц), используют только один байт. Лишь те кодовые точки, которые нельзя выразить с помощью 7 бит, потребляют дополнительные байты (как только значение кодовой точки превышает 127, формат UTF-8 использует для его кодирования не менее двух байт). Это увеличивает потребность в обработке, но сокращается объем памяти для представления текста и, соответственно, пропускная способность, необходимая для передачи его по сети.

Формат UTF-16 является своего рода промежуточным решением. Для среды, где физическая память и пропускная способность менее важны, чем затраты на обработку, все символы BMP будут представлены двумя байтами (16 бит), которые в этом случае называют *кодовой единицей*. Другими словами, кодовым точкам на плоскости BMP соответствует одна кодовая единица.

Ранее в этом разделе было рассказано, как формат UTF-8 использует 1, 2, 3 или 4 байта для кодирования одной кодовой точки Unicode. В формате UTF-16 ситуация похожая; различия возникают, когда приложению нужно представить символ, находящийся за пределами плоскости BMP. Таким кодовым точкам необходимо две кодовых единицы (4 байта), которые называются *суррогатной парой*. Формат UTF-16 позволяет представлять кодовые точки, требующие более 16 бит, путем использования суррогатных пар, и в то же время пара кодовых единиц однозначно идентифицирует одну кодовую точку.
Как и следовало ожидать, формат UTF-32 представляет все кодовые точки с помощью четырех байт. Он наиболее расточителен с точки зрения физической памяти, но требует меньше всего обработки.

Кроме того, существует два вида UTF-16 и UTF-32 (а также UCS-2 и UCS-4): с обратным порядком байтов (big-endian, BE) и с прямым порядком байтов (little-endian, LE). Кодировка с обратным порядком байтов начинается со старшего байта, а кодировка с прямым порядком байтов — с младшего байта. Выбранный метод определяется по метке порядка следования байтов (byte order mark, BOM) в начале закодированного файла. Метка BOM также позволяет идентифицировать формат: UTF-8, UTF-16 или UTF-32.

В отличие от формата UTF-16, который представляет символ либо 2, либо 4 байтами, формат UCS-2 всегда использует 2 байта. По этой причине в нем содержатся только символы из плоскости BMP. Другими словами, форматы UCS-2 и UTF-16 идентичны в том, что касается плоскости BMP, однако UCS-2 не поддерживает суррогатные пары и не может представлять символы за пределами плоскости BMP.
Формат UCS-4 использует 4 байта и позволяет представить тот же набор символов Unicode, что и UTF-32. Тем не менее формат UTF-32 определяет дополнительные функции Unicode и фактически заменил UCS-4.

Думаю, пора заканчивать техническую подготовку. В следующем разделе мы увидим, какое значение все это имеет для разработчиков на Delphi.

Переход на Unicode и приложения Delphi

Поддержка Unicode появилась не в Delphi 2009, просто в этой версии Delphi она стала всеобъемлющей. Например, в Delphi 2007 многие драйверы dbExpress, которые взаимодействовали с серверами, совместимыми с Unicode, также поддерживали Unicode. Кроме того, начиная с Delphi 2005 можно было сохранять и компилировать исходные файлы в формате UTF-8. Нужно также упомянуть двухбайтовый строковый тип WideString, появившийся в Delphi 3.

Один из соавторов этого документа по имени Стив пишет: «Самая большая проблема, с которой я столкнулся [при миграции на Delphi 2009], состояла в том, что приложение уже было совместимо с Unicode благодаря применению типа WideString и элементов управления TNT. Думаю, если бы приложение использовало типы String и PChar, то мне пришлось бы не так трудно».

В Delphi 2009 появились кардинальные изменения. Например, в названиях компонентов, методов, переменных и постоянных, строковых литералах и пр. можно использовать строки Unicode. Но с точки зрения большинства разработчиков, самые серьезные изменения произошли в строковых и символьных типах данных. В начале настоящего раздела мы в общих чертах рассмотрим изменение строковых и символьных типов, а затем перейдем к конкретным областям разработки приложений, где эти изменения ощутимы.

String, Char и PChar

Тип String теперь определен с помощью типа UnicodeString, который является строкой UTF-16. Аналогично: тип Char превратился в двухбайтовый символьный тип WideChar, а PChar — в PWideChar (указатель на двухбайтовый тип Char).

Важным моментом является то, что в результате изменения этих базовых типов данных каждый символ представлен как минимум одной кодовой единицей (два байта), а иногда и больше. Соответственно, размер строки в байтах уже не равен количеству содержащихся в ней символов. (Если только вы и раньше не использовали набор многобайтовых символов: например, для китайского языка. В таком случае благодаря новой реализации Unicode в Delphi задача разработчика упрощается.) Значение типа Char отныне также имеет длину не один, а два байта.

Старый, всем известный и многими любимый строковый тип AnsiString все еще поддерживается. Как и прежде, строки AnsiString содержат одно 8-битовое значение ANSI на каждый символ, используют счетчик ссылок и семантику копирования при записи. Также можно использовать 8-битовый символьный тип AnsiChar и 8-битовый символьный указатель PAnsiChar.

Остались даже традиционные строки Pascal. Они не имеют счетчика ссылок и могут содержать до 255 байт данных. Эти строки определены типом ShortString, хранят символы в ячейках с 1 по 255, а значение длины строки — в нулевом байте.

Вы можете и дальше использовать переменные AnsiString. Существует специальный модуль AnsiStrings.pas, содержащий AnsiString-версии многих функций для обработки традиционных строк (например, UpperCase и Trim). Кроме того, некоторые классические функции для работы со строками перегружены и представлены в двух версиях: AnsiString и UnicodeString. Многие соавторы настоящего документа придерживаются мнения, что преобразование существующих объявлений String в объявления AnsiString является эффективным способом миграции старого программного кода.

Например, в следующем фрагменте кода переменная s объявлена как AnsiString:

```
var
  s: AnsiString;
…
```

Чем среда Delphi 2009 отличается от более ранних версий, так это следующим объявлением:

```
var
  s: String;
…
```

Здесь переменная s имеет тип UnicodeString. Типы UnicodeString и AnsiString имеют много общих черт, но также и ряд серьезных различий. Основное сходство состоит в использовании счетчика ссылок и семантики копирования при записи.

Подсчет ссылок означает, что Delphi ведет собственный учет программного кода, который ссылается на строку. Если код перестает ссылаться на строку, то используемая строкой память автоматически освобождается.

Еще одной эффективной особенностью является копирование при записи. Для типов данных, поддерживающих копирование при записи (в Delphi к ним также относятся динамические массивы), действует следующее правило: если имеется несколько переменных, ссылающихся на определенное значение, то они все ссылаются на одну и ту же ячейку памяти до тех пор, пока вы не попытаетесь изменить значение, на которое ссылается одна из переменных. Когда вы меняете значение, на которое ссылалась одна из переменных, создается копия и изменения применяются к этой копии.

Тип WideString (в отличие от String) не изменился с момента своего появления в составе Delphi. Это двухбайтовая символьная ссылка, не содержащая счетчика ссылок и не поддерживающая копирование при записи. Кроме того, он менее производителен, поскольку не использует диспетчер памяти Delphi FastMM. Некоторые разработчики применяют тип WideString для реализации поддержки Unicode в более ранних версиях Delphi (до 2009), однако он в первую очередь предназначен для разработки COM-приложений и соответствует типу данных BSTR в COM.

Класс AnsiString, который ведет себя подобно типам String в версиях до Delphi 2009, отличается от своих предшественников внутренней структурой. Память, выделявшаяся для традиционного типа AnsiString, содержала по одному байту для

каждого символа в строке плюс восемь дополнительных байтов. В четырех дополнительных байтах хранилась длина AnsiString, а остальные четыре использовались для подсчета ссылок.

Для сравнения: теперь и тип AnsiString, и тип UnicodeString используют двенадцать дополнительных байтов (в дополнение к памяти, содержащей символьные данные), т. е. на четыре байта больше, чем классический тип AnsiString. По аналогии с AnsiString: последние восемь байт хранят длину строки (в символах для AnsiString и кодовых единицах для UnicodeString) и применяются для подсчета ссылок. Два из четырех оставшихся байтов в AnsiString и UnicodeString указывают размер элемента символов, а два — кодовую страницу строки.

Размер элемента для AnsiString равен 1, а для UnicodeString сейчас он составляет 2 (поскольку в будущем размер может измениться, то во внутренней структуре предусмотрено свободнее место). Кодовая страница — более сложная тема, которая будет рассмотрена позже в контексте преобразования строк.

Начало работы

Начнем с хороших новостей. Некоторые старые проекты требуют лишь незначительного изменения для Delphi 2009 или более поздней версии либо могут использоваться в исходном виде. Если вы преимущественно работаете с компонентами VCL (в большинстве случаев они обеспечивают надлежащую поддержку Unicode) или компонентами сторонних поставщиков, которые не поленились разобраться с последствиями реализации поддержки Unicode, то вам будет легче.

Один из соавторов, программист и архитектор ПО Редж Клотье (Rej Cloutier), рассказал, что пока не занимался полным преобразованием своего приложения, но осуществил пробную миграцию в Delphi 2010: «Миграция прошла очень легко. Во-первых, все [наши] строковые функции инкапсулированы в одном модуле... [таким образом,] нам пришлось тщательно проанализировать лишь один модуль (3-4 незначительных изменения). Была успешно выполнена компиляция около 8 проектов для СУБД, от 100 до 185 тысяч программных строк в каждом».

А вот другой пример. У меня есть сотни проектов Delphi, которые я использую в своих обучающих материалах. Одни были написаны еще во времена Delphi 1, а другие иллюстрируют применение новейших функций Delphi. На протяжении многих лет я модифицировал эти проекты по мере обновления обучающих материалов. Большинство из них последний раз было скомпилировано в BDS 2006 или RAD Studio 2007.

После выпуска Delphi 2009 я преобразовал более 100 проектов в Delphi 2009 или Delphi 2010. Лишь в пять проектов или около того пришлось вносить изменения, которые в основном были связаны с передачей данных подпрограммам в библиотеках DLL, а также с записью и считыванием текстовых файлов. Можно ли считать это объективным доказательством простоты преобразования старых приложений Delphi? Нет. Как правильно говорит мой соавтор Стив, «нельзя сравнивать образцы программного кода со сложными реальными приложениями».

Тем не менее, из этого примера можно сделать полезное заключение. Мои учебные проекты иллюстрировали применение функций Delphi, таких как пакеты, библиотеки DLL, компоненты, OpenTools API, COM, DataSnap, строение пользовательского интерфейса, потоки и их синхронизация. Другими словами, большинство этих приложений демонстрировало возможности библиотеки времени выполнения (RTL), библиотеки визуальных компонентов (VCL), компилятора, отладчика, интегрированной среды разработки и редактора Delphi. Функционирование данных компонентов, как правило, не нарушается. Поэтому миграция среды Delphi в формат Unicode прошла согласованно и эффективно.

Настоящие трудности начинаются, когда выходишь за пределы непосредственной области действия Delphi, и именно здесь убеждаешься в правильности утверждения Стива по поводу демонстрационных проектов. Реальные приложения обычно многофункциональны и используют возможности не только самой операционной системы, но также внешних библиотек, пакетов, потоков, файлов и программного кода. Тут-то вы и сталкиваетесь с проблемами.

Еще одно различие между образцами программного кода и реальными приложениями состоит в том, что большинство старых приложений, которые заслуживают миграции, создавались довольно давно. По этой причине в них часто используются методы, первоначально необходимые для повышения производительности или поддержки определенных функций, но безнадежно устаревшие по причине наличия более эффективных альтернатив. Кроме того, они могли быть написаны разными разработчиками, каждый из которых применял свой индивидуальный подход. Возможно также, что использовавшиеся ранее средства и библиотеки сторонних поставщиков сегодня уже не поддерживаются. Список можно продолжать.

Если вам требуется довольно объективный показатель сложности миграции для поддержки Unicode, то последуйте совету Стефана Фризмоза (Steffen Friismose) и воспользуйтесь средством Unicode Statistics Tool, которое можно загрузить с сайта Embarcadero Code Central. Это средство проанализирует ваш исходный код и оценит относительную сложность миграции в формат Unicode. Средство и его описание можно найти на веб-странице http://cc.embarcadero.com/item/27398.

Опираясь на отзывы, полученные от многих соавторов настоящего документа, я позволю себе предположить, что при миграции существующего приложения в Delphi 2009 или более поздней версии вам нужно учесть следующее:

- Размер строк и символов.
- Возврат к AnsiString.
- Преобразование строк.
- Sets of Char.
- Буферы и операции с указателями.
- Считывание и запись внешних данных.
- Внешние библиотеки и сторонние компоненты.

Вопросы, связанные с базами данных.

Каждая из этих тем будет рассмотрена отдельно.

Размер строк и символов

До выпуска Delphi 2009 размер строки в байтах был предсказуем. Сегодня это уже не так просто. Поскольку тип UnicodeString имеет формат UTF-16, то вы, возможно, подумаете, что размер строки в байтах равен удвоенному количеству содержащихся в ней символов (значение Char имеет длину 2 байта). Иначе говоря:

```
var
  SizeOfString: Integer;
  MyString: String;
begin
  MyString := 'Hello World';
  SizeOfString := Length(MyString) * 2;
```

И это действительно почти всегда так. Следующий фрагмент кода еще лучше:

```
var
  SizeOfString: Integer;
  MyString: String;
begin
  MyString := 'Hello World';
  SizeOfString := Length(MyString) * StringElementSize(MyString);
```

Преимущество второго примера (барабанная дробь!) состоит в том, что он содержит меньше допущений в отношении размера строки. Функция StringElementSize определяет размер Char в байтах, вместо того чтобы просто принимать его равным 2.

Но узнать количество символов в конкретной строке не так-то просто. Если вы думаете, что функция Length возвращает количество символов в строке, то вы ошибаетесь. Функция Length возвращает количество кодовых единиц в UnicodeString.

Лучше всего по этому поводу выразился соавтор Джаспер Потьер (Jasper Potjer) из компании Unit 4 Agresso: «Представьте себе 5-символьную строку UTF-16, которая содержит [одну] суррогатную пару. Вернет функция Length количество символов [кодовых точек] (5) или 16-битовых слов [кодовых единиц] (6)?» Он также задал несколько дополнительных вопросов по теме, и я позволю себе здесь их перефразировать.

1. Функция Length возвращает количество кодовых точек или количество кодовых единиц?

2. Если первый символ в UnicodeString представлен суррогатной парой, то MyString[1] содержит кодовую точку (символ) или кодовую единицу (половину суррогатной пары)?

3. Может ли тип Char содержать суррогатную пару? Другими словами, тип Char содержит кодовую точку или кодовую единицу?

4. Если функция Length возвращает кодовые единицы, а не кодовые точки, то как определить количество символов в строке UnicodeString?

Странно, но при работе над настоящим документом мне практически не встречались случаи обсуждения этих вопросов. Единственным исключением является блог Джейкоба Турмана (Jacob Thurman), который можно прочитать по адресу http://www.jacobthurman.com/?p=30. Кроме того, я консультировался с Сеппи Блумом (Seppy Bloom) и Томом Гердесом (Thom Gerdes), которые являются членами группы разработчиков Delphi.

Ниже я своими словами изложил ответы на представленные выше вопросы.

1. Каждый элемент UnicodeString является кодовой единицей. Таким образом, размер строки в байтах равен ее длине, умноженной на размер элементов (StringElementSize или 2, выберите сами). Длина строки UnicodeString в символах часто равна ее длине в кодовых единицах, но если строка содержит суррогатные пары, то это не так.

2. MyString[1] содержит кодовую единицу, которая не обязательно является кодовой точкой.

3. Нет, одно значение Char не может содержать суррогатную пару. Char может содержать одну кодовую единицу.

4. Чтобы точно определить количество символов в строке UnicodeString, используйте одну из справочных функций из модуля SysUtils. Например, если строка UnicodeString включает в себя и символы BMP, и суррогатные пары, то вам поможет функция ElementToCharLen. (Похожий подход применялся с наборами многобайтовых символов до появления Delphi 2009.)

Эти ответы иллюстрирует следующий фрагмент программного кода:

```
var
 s: String;
begin
  s := 'Look '#$D840#$DC01'!';
  ListBox1.Items.Add(s);
  ListBox1.Items.Add(IntToStr(Length(s)));
  ListBox1.Items.Add(IntToHex(Ord(s[6]),0));
  ListBox1.Items.Add(IntToHex(Ord(s[7]),0));
  ListBox1.Items.Add(IntToStr(Length(s) * StringElementSize(s)));
  ListBox1.Items.Add(IntToStr(ElementToCharLen(s, Length(s))));
```

В результате содержимое ListBox1 выглядит следующим образом:

Look 𠀀!
8
D840
DC01
16
7

Несмотря на то, что при печати выводится 7 символов, строка UnicodeString содержит 8 кодовых единиц (см. значение функции Length). Анализ шестого и

седьмого элементов UnicodeString выявляет старший и младший заменитель, каждый из которых является кодовой единицей. Размер строки UnicodeString равен 16 байт, и она содержит 7 кодовых точек, как сообщает функция ElementToCharLen.

Эти ответы действительны в случае с суррогатными парами; при наличии составных символов многое меняется. Если строка UnicodeString содержит хотя бы один составной символ, то он может занимать две кодовых единицы и больше, но при выводе строки будет отображаться как один символ. Более того, функция ElementToCharLen предназначена для работы с парами заменителей, а не с составными символами.

При использовании составных символов возникает потребность в нормализации строк, которая в данный момент не может быть удовлетворена средствами библиотеки времени выполнения Delphi. Когда я спросил об этом Сеппи Блума, он ответил, что корпорация Майкрософт недавно добавила интерфейсы API для нормализации в последние версии ОС Windows®, включая Windows® Vista, Windows® Server 2008 и Windows® 7.

Кроме того, г-н Блум предложил образец программного кода для вычисления количества символов в строке UnicodeString, содержащей комбинированные символы. Я включил этот образец в настоящий документ, но должен сделать несколько предупреждений. Во-первых, образец не подвергался тщательному тестированию и не сертифицирован. Вы можете использовать его на свой собственный риск. Во-вторых, этот код не будет работать в среде с ОС версии до Windows XP; чтобы использовать его с Windows XP, установите интерфейсы Microsoft Internationalized Domain Names (IDN) Mitigation API 1.1.

Вот этот фрагмент кода:

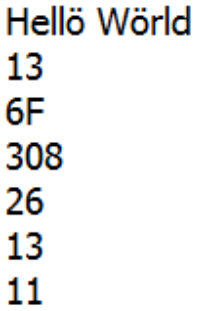
```
const
  NormalizationOther   = 0;
  NormalizationC       = 1;
  NormalizationD       = 2;
  NormalizationKC      = 5;
  NormalizationKD      = 6;

function IsNormalizedString(NormForm: Integer; lpString: LPCWSTR;
  cwLength: Integer): BOOL; stdcall; external 'Normaliz.dll';

function NormalizeString(NormForm: Integer; lpSrcString: LPCWSTR;
  cwSrcLength: Integer; lpDstString: LPWSTR;
  cwDstLength: Integer): Integer; stdcall; external 'Normaliz.dll';

function NormalizedStringLength(const S: string): Integer;
var
  Buf: string;
begin
  if not IsNormalizedString(NormalizationC, PChar(S), -1) then
  begin
    SetLength(Buf, NormalizeString(NormalizationC,
              PChar(S), Length(S), nil, 0));
    Result := NormalizeString(NormalizationC, PChar(S),
                              Length(S), PChar(Buf), Length(Buf));
  end
  else
```

```
    Result := Length(S);
end;
```

Следующий фрагмент кода для строки UnicodeString с двумя комбинированными символами демонстрирует использование функции NormalizedStringLength:

```
var
 s: String;
begin
  ListBox1.Items.Clear;
  s := 'Hell'#$006F + #$0308' W'#$006F + #$0308'rld';
  ListBox1.Items.Add(s);
  ListBox1.Items.Add(IntToStr(Length(s)));
  ListBox1.Items.Add(IntToHex(Ord(s[5]),0));
  ListBox1.Items.Add(IntToHex(Ord(s[6]),0));
  ListBox1.Items.Add(IntToStr(Length(s) * StringElementSize(s)));
  ListBox1.Items.Add(IntToStr(ElementToCharLen(s, Length(s))));
  ListBox1.Items.Add(IntToStr(NormalizedStringLength(s)));
```

В результате содержимое ListBox1 выглядит следующим образом:



Как видите, отображаемая строка содержит 11 символов, а функция Length возвращает 13 кодовых единиц (и это правильно). Более того, пятый и шестой элементы строки UnicodeString являются компонентами первого комбинированного символа. Функция ElementToCharLen насчитала 13 символов, а функция NormalizedStringLength — 11.
Как это понимать? Функция ElementToCharLen ошибается? На самом деле нет. Строка UnicodeString действительно содержит 13 кодовых точек, просто правила Unicode дважды объединяют две кодовых точки в одну комбинированную кодовую точку, которая и отображается на экране. (Происходит нормализация строки.)

Давайте сравним с предыдущим примером с суррогатными парами. Каждая суррогатная пара занимала две кодовых единицы, однако эти кодовые единицы представляли одну кодовую точку. Функция ElementToCharLen считает кодовые точки.
Помните пример, который я привел ранее в этом документе при первом упоминании о составны символах? Требовалось посчитать частоту использования знака «трема» независимо от символа, над которым он стоял. В ходе подсчета знак «трема» является отдельным символом. Нужно также отметить, что составные символы редко встречаются в обычных приложениях, а их применение ограничено специальными случаями, наподобие этого примера.
В завершение раздела хочу сказать несколько слов о модуле Character.pas, который впервые появился в составе Delphi 2009. В этом модуле есть класс TCharacter, содержащий множество функций для получения информации об

отдельных символах в строке UnicodeString. Каждая функция класса имеет соответствующую автономную функцию, которая вызывает ее напрямую.

Например, существуют функции, определяющие регистр конкретного символа и то, является ли он графическим знаком, знаком препинания или управляющим символом. Предусмотрены также функции для преобразования отдельных символов в формат UTF-32 и обратно.

По некоторым причинам модуль Character.pas упоминается в этом документе только раз и то вскользь. Обязательно обратите на него внимание — он содержит много полезного.

Возврат к AnsiString

Анализируя полученные отзывы, я вижу, что имеется два основных подхода к миграции существующих приложений в Delphi 2009 или более поздней версии. Первый состоит в том, чтобы оставить объявления String, Char и PChar как есть и сосредоточиться на фрагментах, где новые типы Unicode недействительны (например, вызов передает значение PChar внешней процедуре, которая требует массива байтов).

Второй подход предполагает преобразование объявлений String в AnsiString, Char в WideChar, а PChar в PWideChar. Это минимизирует структурное рассогласование между символами ANSI и UTF-16.

Есть убедительный аргумент в пользу применения Unicode при миграции существующих приложений. Марко Канту (Marco Cantù), автор книги *«Delphi 2009 Handbook»*, пишет: «В большинстве случаев целесообразно перейти на новый тип UnicodeString» — и далее называет ряд получаемых при этом преимуществ, например ускорение многих вызовов интерфейса Windows API (он сегодня преимущественно имеет формат Unicode).

Однако, как правило, речь идет не о полном переходе на Unicode или о полном возврате к AnsiString, а об определенной комбинации этих подходов. Как утверждает г-н Канту, «преобразование кода к AnsiString уместно для загрузки или сохранения файлов, перемещения информации в базу данных и извлечения ее оттуда, а также использования протоколов Интернета, где символы должны оставаться 8-битовыми».

Эта проблема имеет и чисто практический аспект. Если вам нужно быстро выполнить преобразование без дальнейшей оптимизации кода, то, возможно, следует просто оставить однобайтовые переменные типа Char. С другой стороны, если ваше приложение подвергается всесторонней модернизации и будет активно использоваться еще длительное время, а вы располагаете временем для внесения изменений, то целесообразно полностью перейти на Unicode.

Давайте вначале рассмотрим использование AnsiString. Роджер Конелл пишет: «Предоставленный [компанией Embarcadero] перечень действий [по миграции устаревшего кода в формат Unicode] практически невыполним, учитывая количество строк кода[, требующих внимания]. Я решил поддерживать строки в

формате AnsiString и разработал для этой цели специальный преобразователь. Планирую постепенно переходить… на Unicode [по мере появления свободного времени]».

Этот подход имеет полное право на жизнь. Г-н Конелл, член Австралийской группы пользователей Delphi (ADUG), даже бесплатно распространяет свою программу преобразования через Интернет. Найти программу вместе с предостережениями по части использования можно на веб-странице:

```
http://www.innovasolutions.com.au/delphistuf/ADUGStringToAnsiStringConv.htm
```

Предлагая быстрый переход на Delphi 2009, Роджер пишет: «[Этот подход] гарантирует, что ваш код можно будет компилировать в Delphi 6, Delphi 7 и Delphi 2009. Возможны некоторые проблемы с производительностью в пользовательском интерфейсе, но я считаю, что… их причина заключается в самой среде Delphi 7».

Тем не менее, простой возврат к AnsiString не всегда помогает. Соавтор Мариано Винсент де Уркиса (Mariano Vincent de Urquiza) из компании MVU Technologies LLC пишет: «Тотальная замена String на AnsiString и Char на AnsiChar не дала желаемого результата. Каждую процедуру пришлось переделать и протестировать. Было затронуто несколько модулей. Я был в отчаянии, казалось, это никогда не кончится».
Ларс Дибдал (Lars Dybdahl), руководитель группы разработчиков в компании Daintel ApS, предлагает выбирать способ обработки строк в зависимости от типа программного кода. Так, он пишет: «У нас был очень старый программный код с указателями и внешними компонентами. Преобразовать его было крайне трудно. Однако мы обратили внимание, что объем входящих и исходящих строковых данных невелик; проще всего оказалось переименовать String в RawByteString, а PChar в PAnsiChar и использовать интерфейсы Windows API версии ANSI. Разумеется, эта часть программы не поддерживала Unicode, что, впрочем, вполне допустимо в отдельных случаях, например в модулях шифрования, обрабатывающих двоичные данные в виде строковых переменных».

В дополнение Ларс дает следующий совет: «Обычно после переименования [типов данных] и отладки модуля целесообразно настроить интерфейс на использование String (UnicodeString), чтобы другие модули, которые работают со строками UnicodeString, не создавали предупреждений при обращении к этому модулю. Фактически преобразование в UnicodeString и обратно инкапсулируется в разделе реализации».
По этому поводу один из анонимных соавторов рассказывает следующее: «Я написал [функцию] Delphi 2007 CharInSet и использовал ее по необходимости. Кроме того, кое-где я поменял тип Char на AnsiChar (вызовы Windows API, сторонние интерфейсы DLL, определения файловых форматов и пр.). [Также] я избавился от некоторых файлов текстового типа. Когда появился Delphi 2009, я попробовал демонстрационную версию, и через день-два у меня уже все работало. Возможно, помогло то, что я всегда с осторожностью относился к типу PChar».

Роджер Конелл отмечает, что благодаря изменению объявлений String на AnsiString программный код становится совместимым с более ранними версиями Delphi,

однако фактически такое преобразование не является обязательным. Некоторым разработчикам удавалось использовать объявления String в существующем виде (в зависимости от версии Delphi они компилируются как AnsiString или как UnicodeString).
Нард Мозели (Nard Moseley) из компании Digital Metaphors, которая поставляет популярное средство отчетности ReportBuilder для Delphi, так описывает процесс миграции: «ReportBuilder — это сложная программа, содержащая свыше 700 000 строк кода. <...> Перемещая ReportBuilder на Unicode, мы придерживались стратегии, предложенной главным научным консультантом Delphi Алленом Бауэром (Allen Bauer). Представьте, что приложение — это ящик. Внутри ящика все строки в формате Unicode. Внешние грани ящика соответствуют внешним границам приложения, где оно взаимодействует с другими системам, файлами и прочими объектами, которые могут использовать другую кодировку символов».

Далее г-н Мозели утверждает: «Одной из дополнительных трудностей, с которыми мы столкнулись, была необходимость поддерживать в одном базовом коде и старую (ANSI), и новую (Unicode) библиотеку VCL. Мы всегда стремимся уменьшать количество условных компиляций, чтобы исходный код оставался максимально компактным и понятным. Мы построили фасады для таких классов VCL (Unicode), как TCharacter и TEncoding. Другими словами, вместо того чтобы вызывать эти классы напрямую, мы вызываем набор внутренних классов, которые способны в определенных условиях вызывать классы TCharacter и TEncoding».

Еще одним сторонником подхода «пусть String остается String» является Дэвид Бернеда (David Berneda) из компании Steema Software, разработавшей программу для построения диаграмм TeeChart, которая поставляется вместе с Delphi и отдельно в версии Professional. Дэвид пишет: «Мы просто проследили, чтобы все данные имели тип String (т. е. без проблем компилировались во всех версиях Delphi)... при вызове интерфейсов API в формате, отличном от Unicode, при использовании ShortString и при выполнении... преобразований PAnsiChar(текст)».

Преобразование строк

Преобразование строки происходит, когда вы присваиваете строку одного типа строке другого типа либо приводите строку к типу данных, отличному от первоначального. Если при разработке новых приложений в Delphi 2009 или более поздней версии необходимость в преобразовании строк возникает лишь изредка, то в ходе миграции приложений оно встречается сплошь и рядом.

Но перед тем как продолжить, хочу предложить вашему вниманию одно замечание из блога Яна Гойвертца (Jan Goyvaerts). Он пишет, что если включена директива компилятора $STRINGCHECKS (по умолчанию), то Delphi добавляет дополнительный код для проверки типов строк. Поскольку в языке Delphi производится строгий контроль типов, то эту директиву можно, ничем не рискуя, выключить и тем самым повысить производительность кода. В C++Builder эта директива компилятора должна быть включена.

В отношении преобразования строк у меня есть для вас и хорошие и плохие новости. Хорошая новость состоит в том, что строковые типы совместимы по

присваиванию с другими строковыми типами, а символьные типы совместимы по присваиванию с другими символьными типами. Во время присваивания может возникнуть необходимость в преобразовании, но в случае присваивания одной строки другой строке это преобразование производится автоматически.

В зависимости от типа преобразования возможна потеря данных. Это плохая новость. Понимание причин потери данных является одним из наиболее сложных шагов на пути к совершенному владению Unicode. Давайте более внимательно рассмотрим тему, которая уже была вскользь упомянута в настоящем документе: кодовые страницы.

Кодовые страницы

Термин «кодовая страница» означает механизм, использованный для расширения исходного 7-битового набора символов ASCII (от #$00 до #$7F). В первой версии MS-DOS эти значения в диапазоне от #$80 до #$FF использовались преимущественно для символов псевдографики. (Применявшиеся в MS-DOS кодовые страницы назывались кодовыми страницами изготовителей комплектного оборудования (OEM).)

В ранних версиях Windows появился ряд новых кодовых страниц *(кодовые страницы Windows)*, которые обеспечивали отображение информации на разных языках. В этих кодовых страницах диапазон от #$80 до #$FF преимущественно содержит символы и знаки, относящиеся к определенному языку.

Каждая кодовая страница имеет свой идентификатор. Например, на большинстве компьютеров в США используется кодовая страница 1252, идентификатор 437 соответствует кодовой странице OEM в первых компьютерах IBM PC, а идентификатор 950 обозначает кодовую страницу ANSI и OEM для китайского языка (традиционное письмо).

Чтобы иметь возможность отображать информацию на языках, которым необходимо значительно больше, чем 128 дополнительных символов, например на японском или китайском, ОС Windows поддерживает и однобайтовые и многобайтовые кодовые страницы. В многобайтовых кодовых страницах каждый символ представлен одним или несколькими байтами. Например, в двухбайтовом наборе символов *старший байт* (его значение всегда больше #$7F) в комбинации с *младшим байтом* обозначает один символ. Кодовые страницы 932 (японский) и 950 (китайский, традиционное письмо) являются двухбайтовыми наборами символов.

В каждом экземпляре ОС Windows имеется стандартная кодовая страница, которая задает набор символов, по умолчанию используемый на данном компьютере для символов в формате, отличном от Unicode. Кроме того, она определяет набор символов для типа AnsiString по умолчанию (все версии Delphi).

Ранее вы уже узнали, что переменная типа String в Delphi 2009 или более поздней версии содержит 2 байта для хранения данных о кодовой странице. Типам UnicodeString соответствует кодовая страница 1200; по ней ОС Windows определяет формат строк: UTF-16LE. Для AnsiString применяется кодовая страница

Windows по умолчанию либо кодовая страница, заданная для пользовательского типа AnsiString.

Обратите внимание, что неявно все переменные AnsiString в составе приложения имеют одну и ту же кодовую страницу, а именно кодовую страницу вашего экземпляра Windows по умолчанию. Чтобы создать в приложении две переменные AnsiString с разными кодовыми страницами, нужно явно сообщить Delphi конкретную кодовую страницу (некоторое представление об этом вам дадут фрагменты кода в следующем разделе).

Несколько соавторов предоставили свои замечания к одной из более ранних версий документа. В частности, Ларс Дибдал попросил меня удалить упоминание о том, что тип UnicodeString имеет кодовую страницу (вернитесь на два абзаца назад). Я решил не делать этого, но привожу аргументы Ларса, поскольку они, по моему мнению, могут быть полезны вам в процессе миграции.

Ларс писал: «Я никогда не использую номер кодовой страницы для формата UTF-16 при программировании в Delphi 2009 или в ходе миграции. <...> Оказалось довольно трудно понять, что тип UnicodeString всегда имеет одну и ту же кодировку UTF-16LE. После этого стало намного проще, ведь достаточно придерживаться типа UnicodeString везде, где возможно, и все работает как надо. Если бы в документах, которые я читал [перед миграцией приложений в Unicode, вместо] кодовой страницы для типа UnicodeString было сказано, что строки UnicodeString очень удобны и эффективны по сравнению с AnsiString, то мне удалось бы сэкономить массу времени».

Преобразование строк и потеря данных

Как я уже упоминал ранее, существует вероятность потери данных при преобразовании одного типа строки (кодовая страница) в другой тип. Если исходная строка содержит символы, отсутствующие в кодовой странице целевой строки, то происходит потеря данных.

Это можно продемонстрировать с помощью одного примера. Рассмотрим следующий фрагмент программного кода:

```
type
  IBMAnsi = type AnsiString(437); //IBM OEM
var
  s: IBMAnsi;
  a: AnsiString;
begin
  // Используйте эту строку, если у вас по умолчанию задана кодовая страница,
отличная от 1252.
  DefaultSystemCodePage := 1252;
  s := #$B4;
  a := s;
  ListBox1.Items.Add(s);
  ListBox1.Items.Add(a);
  s := a;
  ListBox1.Items.Add(s);
```

После выполнения этого кода в ListBox1 содержатся следующие значения:

Символ #$B4 кодовой страницы OEM 437 соответствует символу Unicode U+2524, который называется «BOX DRAWINGS LIGHT VERTICAL AND LEFT». (Стоит, наверное, сказать, что #$B4 — это литерал AnsiChar, который отличается от литерала WideChar. #$2524 — литерал WideChar.) Этого символа нет в кодовой странице по умолчанию (1252) для экземпляра Windows, в котором был выполнен программный код. В результате произошла потеря данных, и был распечатан неправильный символ.

Две последние строки в этом фрагменте кода показывают, что дело не только в двух кодовых страницах, в которых код #$B4 соответствует разным символам. Значение AnsiString передается назад в переменную IBMAnsi, а затем выводится на экран. Как видите, полученный символ отличается от исходного.

Если объявить переменную a как String (UnicodeString) вместо AnsiString, то данные не будут потеряны. Так, символ U+2524 будет отображаться в строках обоих типов. Когда переменная a объявлена как String, получим следующий результат:

RawByteString

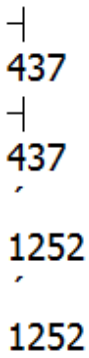
Существует специальный тип AnsiString под названием RawByteString. Тип RawByteString не имеет собственной кодовой страницы, и, следовательно, присвоение строк типу RawByteString не вызывает неявного преобразования. Таким образом, кодовая страница значения, назначенного типу RawByteString, является кодовой страницей всего, что ему назначается. Это делает RawByteString идеальным типом данных для передачи параметров AnsiString. В случае передачи параметров, имеющих разную кодовую страницу, с помощью любого другого типа данных происходит неявное преобразование типов, при котором возможна потеря данных.

Следующий пример программного кода демонстрирует, как RawByteString «принимает» кодовую страницу значения AnsiString. Обратите внимание, что когда значение IBMAnsi присваивается переменной RawByteString, то RawByteString принимает кодовую страницу 437. Когда этой же переменной RawByteString присваивается значение AnsiString (кодовая страница по умолчанию), переменная принимает кодовую страницу 1252 (кодовая страница по умолчанию для экземпляра Windows на компьютере, где был выполнен программный код).

```
type
  IBMAnsi = type AnsiString(437); //IBM OEM
var
  i: IBMAnsi;
  a: AnsiString;
```

```
  r: RawByteString;
begin
  // Используйте эту строку, если у вас по умолчанию задана кодовая страница,
отличная от 1252.
  DefaultSystemCodePage := 1252;
  i := #$B4;
  r := i;
  ListBox1.Items.Add(i);
  ListBox1.Items.Add(IntToStr(StringCodePage(i)));
  ListBox1.Items.Add(r);
  ListBox1.Items.Add(IntToStr(StringCodePage(r)));

  a := #$B4;
  r := a;
  ListBox1.Items.Add(a);
  ListBox1.Items.Add(IntToStr(StringCodePage(a)));
  ListBox1.Items.Add(r);
  ListBox1.Items.Add(IntToStr(StringCodePage(r)));
```

После выполнения этого кода ListBox1 имеет следующий вид:



Обычно при миграции в Unicode не нужно заботиться о неявном преобразовании. С другой стороны, если трудности с неявным преобразованием кодовых страниц у вас возникают, то, возможно, вам пришлось столкнуться с очень сложным случаем миграции.

«Все действительно сложно, — пишет Ларс Дибдал. — Программисту очень легко запутаться, а если программист не понимает принцип действия, он перестает верить в саму возможность миграции. Перед тем как приступить к миграции, я экспериментировал в Delphi. Особенно раздражает, что переменная AnsiString(1250) может содержать строку с другой кодовой страницей, а вам крайне нежелательно зависеть от соответствия значений байтов внутри кодовой странице».

Ларс предоставил другой фрагмент кода, который иллюстрирует его замечания, особенно в отношении литералов AnsiString и UnicodeString, переменных RawBytesString и неявного преобразования. Фрагмент довольно интересный, хоть и длиннее других образцов. Я включил его в этот документ, чтобы вы могли поэкспериментировать самостоятельно. (Я лишь немного подкорректировал комментарии Ларса, оставив все остальное в первоначальном виде.)

```
procedure TForm1.Button2Click(Sender: TObject);
type
  String1250 = type AnsiString(1250);
```

```
  String1252 = type AnsiString(1252);
var
as1:   String1250;
as1b:  String1250;
as1c:  String1250;
as2:   String1252;
s1,s2: String;
begin
DefaultSystemCodePage := 1252;
// Выражения справа выглядят похоже, однако это не так.
as1 := #$C0;                    // Литерал AnsiChar, не имеющий кодовой
страницы.
as1b := #$C0#$C0;               // Литерал UnicodeString.
as1c := RawByteString(#$C0#$C0);// Литерал UnicodeString с преобразованием
                                // в момент выполнения к локальной кодовой
странице —
                                // это не переменная RawByteString со
значениями $C0 (!).
as2 := #$C0;                    // Литерал AnsiChar, не имеющий кодовой
страницы.

// Оба литерала AnsiChar сохранили байтовое значение. Переменная UnicodeString
— нет.
Assert (ord(as1[1])=$C0);
Assert (ord(as1b[1])<>$C0);
Assert (ord(as1c[1])=$C0);  // as1c теперь имеет кодовую страницу 1252.
Assert (ord(as2[1])=$C0);

// Продемонстрируем, насколько все запутанно.
// И as1 и as1c имеют тип String1250, но переменной as1c теперь назначена
кодовая страница 1252,
// поскольку она была создана с помощью RawByteString(). Это означает, что обе
переменные
// содержат значения $C0, но они не содержат одинаковых
// символов.
Assert (length(as1)=1);
Assert (as1[1]+as1c[2] =  as1c[1]+as1c[2]);
Assert (as1   +as1c[2] <> as1c[1]+as1c[2]);

// Поскольку кодовые страницы разные, то все символы тоже разные.
s1:=as1;
s2:=as2;
Assert (s1<>s2);
end;
```

Явные преобразования

Если преобразование строк происходит автоматически, то есть ли необходимость в явном преобразовании? Ответ — да. Один из наиболее распространенных случаев: например, вам нужно использовать значение AnsiString для передачи данных в вызов Windows API, но вы получаете данные из источника, имеющего тип, отличный от AnsiString.

Ниже приведен удачный пример от хорошо известного специалиста по Delphi Боба Сворта (Bob Swart), или, как его еще называют, доктора Боба. Он пишет: «Это один из самых простых и удачных примеров, которые мне доводилось видеть. В нем используются старые функции экспорта Win32 DLL, возвращающие значения PChar. Да, старого типа PChar, который сейчас известен как PAnsiChar».

Внешние библиотеки мы обсудим немного позже, а теперь давайте сосредоточимся на явном преобразовании. Здесь мы имеет статическую инструкцию импорта для простой подпрограммы в библиотеке AnsiCharDLL.dll:

```
function EchoAnsiString(const S: PAnsiChar): PAnsiChar; stdcall
  external 'AnsiCharDLL.dll';
```

Как видите, подпрограмма принимает значение PAnsiChar и возвращает значение PAnsiChar (фактически она возвращает принятое значение PAnsiChar).
«Мы можем импортировать функцию и указать, что она использует тип PAnsiChar, — пишет Боб. — В то же время при вызове [такой] функции, которой требуется значение PAnsiChar, с использованием TEdit (со значением UnicodeString) в качестве [входного] значения потребуется не одно, а ДВА явных преобразования строки».

Это наглядно демонстрирует следующий фрагмент программного кода:

```
ShowMessage(
  EchoAnsiString(
    PAnsiChar(AnsiString(Edit1.Text)))); // Двойное преобразование!!!
```

Если значение, передаваемое EchoAnsiString, уже имеет тип AnsiString, то второе приведение типа (к AnsiString) уже не потребуется. Это демонстрирует следующий пример:

```
var
  Msg: AnsiString;
begin
  Msg := 'Hello World';
ShowMessage(EchoAnsiString(PAnsiChar(Msg)));
```

Похожий пример приводит и другой хорошо известный специалист по Delphi — Марко Канту. В этом примере (см. стр. 98) преобразование AnsiString используется для вызова GetProcAddress (вызов Windows API для динамического получения точки входа подпрограммы во внешней библиотеке DLL). Название импортируемой подпрограммы хранится в переменной strFnName, имеющей тип UnicodeString:

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

Марко также советует включить в Delphi все предупреждения о преобразовании строк (некоторые из них по умолчанию выключены). Следующий список скопирован из его книги (стр. 88):

> Explicit string cast [Явное преобразование строки]
> Explicit string cast with potential data loss [Явное преобразование строки с возможной потерей данных]
> Implicit string cast [Неявное преобразование строки]
> Implicit string cast with potential data loss [Неявное преобразование строки с возможной потерей данных]
> Narrowing given wide/Unicode string constant lost information [Ограничение данной длинной строковой константы/строковой константы Unicode приведет к потере информации]
> Narrowing given WideChar constant to AnsiChart lost information [Ограничение

данной константы WideChar до AnsiChar приведет к потере информации]
WideChar reduced to byte char in set expression [WideChar сокращен до байтового Char в таком выражении]
Widening given AnsiChart constant to WideChar lost information [Расширение данной константы AnsiChar до WideChar приведет к потере информации]
Widening given AnsiString constant lost information [Расширение данной константы AnsiChar приведет к потере информации]

В этом же духе высказывается и Ларс Дибдал: «Предупреждения о преобразовании строк нужно устранять, а не игнорировать. С большинством из них разобраться довольно легко». Кроме того, он советует: «Не создавайте преобразований UnicodeString к AnsiString, которые выполняются очень часто. Приведу в качестве примера использование типа TStringList в модуле AnsiString, так что всякое присвоение переменной TStringList приводит к преобразованию строк. Это сильно замедлит ваше приложение».

Я завершаю этот раздел еще одним интересным замечанием Ларса: «Проблема сравнения строк очень сложна. К примеру, рассмотрим следующий программный код:

```
var
  line: String;
const
  myconstant: String='<любой текст с необычными символами Unicode>';
...
ReadLn (file,line);
if line=myconstant then...
```

Будет ли этот код работать в Delphi? Не знаю. Я вижу, что строка "if line=myconstant then" компилируется на машинном языке в вызов UStrEqual, но совершенно непонятно, что это: двоичное сравнение или надлежащее сравнение строк Unicode, учитывающее возможность использования двумя идентичными строками разных байтовых значений (составные и разложенные символы)».

«Судя по всему, Windows API не используется, так что, скорее всего, это двоичное сравнение и приведенный выше фрагмент кода будет работать не всегда. <…> Попробовал найти ответ через Google, но безуспешно. Возможно, Delphi UnicodeString обычно хранит строку в нормализованном виде, но что, если значение UnicodeString считывается с помощью TStream.Read? Строка не будет нормализована, и ее будет нельзя сравнить побайтово с нормализованной строкой Unicode».

Должно быть, вы помните, что мы уже встречались с нормализацией в предыдущем разделе при обсуждении вопроса длины String и Char, и как выяснилось, библиотека RTL пока напрямую не поддерживает нормализацию. Поэтому я думаю, что выражение «line=myconstant» может быть равно False, даже если две нормализованные строки содержат идентичные значения.

С другой стороны, в модуле SysUtils имеется ряд функций для сравнения строк, вызывающих функцию CompareString Windows API (в действительности это две функции: версия ANSI и версия Unicode), которая выполняет сравнение

нормализованных строк. Следующий образец кода демонстрирует проблему и ее решение:

```
var
 s1, s2: String;
begin
  ListBox1.Items.Clear;
  s1 := 'Hell'#$006F + #$0308' W'#$006F + #$0308'rld';
  s2 := 'Hellö Wörld';
  ListBox1.Items.Add(s1);
  ListBox1.Items.Add(s2);
  ListBox1.Items.Add(BoolToStr(s1 = s2, True));
  ListBox1.Items.Add(BoolToStr(AnsiCompareStr(s1, s2) = 0, True));
```

Содержимое ListBox1 выглядит следующим образом:

Hellö Wörld
Hellö Wörld
False
True

SETS IN CHAR

Возможно, вы помните упомянутого ранее анонимного соавтора, который легко справился с преобразованием и сказал: «Я написал [функцию] Delphi 2007 CharInSet и использовал ее по необходимости». Он намекал на то, что наборы Char теперь потеряли смысл.

Причина проста. Sets в Delphi могут содержать не более 256 элементов, а тип Char имеет размер уже не 1 байт, а 2.

Если вы попробуете объявить набор Char, появится следующее предупреждение:

```
W1050 WideChar reduced to byte char in set expressions. Consider using
'CharInSet' function in 'SysUtils' unit. (WideChar сокращен до байтового
символа в таких выражениях. Рассмотрите возможность использования функции
CharInSet в модуле SysUtils.)
```

У вас есть два варианта: либо объявить набор AnsiChar, либо использовать CharInSet, как советует компилятор. В действительности функция CharInSet, как и многие другие строковые функции Delphi, перегружена и имеет как однобайтовую версию, так и версию WideChar. В модуле SysUtils она объявлена следующим образом:

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

Как видите, функция CharInSet возвращает булево значение, если символ, который вы ей передали, присутствует в наборе, который вы ей передали. Второй параметр TSysCharSet объявлен следующим образом:

```
TSysCharSet = set of AnsiChar;
```

Буферы и операции с указателями

Пожалуй, наиболее важным и сложным аспектом миграции приложений в формат Unicode является программный код, в котором присутствуют символы в операциях с указателями и массивы символов в качестве буфера. Этот вывод подтверждается комментариями соавторов документа. (Только представьте себе последствия применения неявного преобразования к строке, используемой как массив байтов.)

Например, Олаф Моньен (Olaf Monien) из компании Delphi Experts (www.DelphiExperts.net) пишет: «[Перенос] кода, в котором интенсивно используются PChar, буферы и операции с указателями... обычно требует очень много усилий, поскольку нужно проверить каждую его строку в отдельности». Косвенно подтверждает эти слова анонимный соавтор, которому удалось осуществить миграцию довольно легко: «Возможно, помогло то, что я всегда с осторожностью относился к типу PChar».

Ларс Дибдал пишет о сложности кода следующее: «У нас был очень старый программный код с указателями и внешними компонентами. Преобразовать его было крайне трудно». В таком коде нужно проверять каждую строку.

Если вы (как и я) стараетесь не увлекаться использованием указателей и буферов, то, возможно, вас удивляет, почему другие разработчики отдают предпочтение этим конструкциям. Ответ кроется в скорости и функциональности.

Как правило, такие конструкции применяются при решении сложных задач, где важна скорость, при выполнении операций, для которых нет (или не было) альтернативы, либо в составе технических приемов, созданных в ранние годы Delphi (или Turbo Pascal). (Даже при появлении новых приемов и классов, заменяющих устаревший программный код, некоторые разработчики продолжают использовать его по привычке, в ущерб удобству сопровождения. Это мое личное мнение, которое, конечно же, нельзя считать объективным.)

В принципе, указатели и буферы сами по себе не влияют на сложность такого кода. Все дело в переменных Char, которые используются в указателях. Теперь, после изменения размера String и Char в байтах, утратило свою силу одно из фундаментальных допущений, на котором строится большая часть подобного программного кода: индивидуальные Char имеют длину 1 байт.

Поскольку код этого типа так трудно поддается преобразованию в формат Unicode (и обслуживанию в общем смысле), а также требует внимательной проверки, то целесообразно по возможности его оптимизировать. Другими словами, в подобных операциях отказывайтесь от применения Char в пользу более подходящих типов данных. Например, Олаф Моньен пишет: «Я бы не советовал использовать байтовые операции с данными типа Char (или String). Если вам требуется байтовый буфер, используйте тип данных Byte: buffer: array[0..255] of Byte;».

Например, вы могли написать следующий код:

```
var
  Buffer: array[0..255] of AnsiChar;
begin
  FillChar(Buffer, Length(Buffer), 0);
```

Преобразовать его в формат Unicode можно следующим образом:

```
var
  Buffer: array[0..255] of Char;
begin
  FillChar(Buffer, Length(buffer) * StringElementSize(Buffer), 0);
```

С другой стороны, как советует Олаф, целесообразно отказаться от использования массива Char в пользу массива Byte в качестве буфера. Например, так (похоже на первый сегмент, но отличается от второго размером буфера):

```
var
  Buffer: array[0..255] of Byte;
begin
  FillChar(Buffer, Length(buffer) * StringElementSize(Buffer), 0);
```

Или так:

```
var
  Buffer: array[0..255] of Byte;
begin
  FillChar(Buffer, Length(buffer) * SizeOf(Buffer), 0);
```

Преимущество двух последних примеров состоит в том, что вы получаете необходимый буфер, способный вмещать байтовые значения. (И Delphi не будет пытаться применить неявное преобразование строк, поскольку оно функционирует с байтами, но не кодовыми единицами.) Если вы хотите работать с указателями, то используйте PByte. PByte — это указатель на Byte.

Одним из случаев, где такие изменения невозможны, является взаимодействие с внешней библиотекой, которая принимает указатель на символ или массив символов. При этом действительно требуется буфер символов, которые обычно имеют тип AnsiChar. Кроме массивов Byte, можно использовать тип TBytes, который представляет собой динамический массив Byte. Тип TBytes объявлен в модуле SysUtils и выглядит следующим образом:

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

Перед тем как перейти к следующей теме, думаю, следует привести слова Алена Бауэра, главного научного консультанта Embarcadero Technologies. В своем блоге (http://blogs.embarcadero.com/abauer/2008/01/24/38852) он пишет: «Поскольку мы можем выполнять операции с указателями, используя тип PChar... многие разработчики (включая наших) начинают вытворять сумасшедшие вещи, например, приводить указатель к типу PChar и затем производить арифметические операции над указателями. <…> В результате появляется программный код, изобилующий командами приведения указателей к типу PChar либо прямым использованием указателей PChar, даже если обрабатываемые данные не являются символами байтового размера. Другими словами, с

помощью указателей PChar выполняются операции над байтовыми буферами, содержащими произвольные данные».

«В процессе разработки RAD Studio 2009 мы обнаружили, что подобные вещи встречаются даже в нашем собственном программном коде. (Я говорил, все это делают!) Трюк с PChar был единственно возможным выходом, который упрощал программный код и повышал его удобочитаемость. <…> При взгляде на код становится понятно, что разработчик стремился получить доступ к буферу данных как к массиву байтов и использовал PChar лишь как удобное средство для доступа к массиву или выполнения простых арифметических операций».

«Если вы объявляете типизированный указатель, когда [включена директива компилятора $POINTERMATH], то любая переменная этого типа будет допускать все арифметические операции над указателями с разбиением чисел на целую и дробную части, а также операции индексирования массива. <…> Указатель PByte объявляется с включением этой директивы. Другими словами, для доступа к байтовым указателям и байтовым массивам вместо PChar теперь можно использовать PByte, причем существующие операторы и логика не требуют изменения. Нужно лишь выполнить простой поиск с заменой в исходном коде. Можно, конечно, было изменить ссылки PChar на PAnsiChar, но это только увековечило бы неясность в отношении фактического предназначения программного кода».

Считывание и запись внешних данных

Внешние файлы и потоки — это еще одна тема, требующая внимания в ходе миграции в формат Unicode. Рей Конопка (Ray Konopka) из компании Raize Software, которая выпускает популярные средства и компоненты для разработчиков на Delphi, поднял вопрос в связи с использованием методов SaveToFile и LoadFromFile в элементах управления списками. Однако его замечания применимы ко многим ситуациям, предполагающим запись в файлы и потоки либо считывание из них.

Файловый ввод/вывод и кодировка текста

«В большинстве элементов управления списками имеются методы SaveToFile и LoadFromFile, — пишет Рей. — Обычно эти методы компилируются в RAD Studio 2009 без каких-либо проблем. Даже в момент выполнения они, кажется, работают правильно. Но лишь пока в один из элементов не добавлен символ Unicode».

«Даже при вызове SaveToFile все будто бы работает правильно. Только по умолчанию файлы, создаваемые с помощью метода SaveToFile, имеют кодировку ANSI, т. е. все символы Unicode будут сохраняться в файле неправильно. При вызове метода LoadFromFile список будет заполнен неправильными данными из ранее сохраненного файла, поскольку настоящий символ уже потерян».

Рей продолжает: «Нужно указывать кодировку для следующего текстового файла. Для этого автор компонента должен предоставить перегруженные версии методов SaveToFile и LoadFromFile (а также SaveToStream и LoadFromStream), принимающие параметр кодировки».

«Решение правильное, вот только разработчик, который использует компонент, должен выбрать подходящую кодировку. Можно выбрать для всех файлов кодировку на базе Unicode (например, UTF-8) и закрыть вопрос. Однако это означает, что даже списки, которые содержат только символы ANSI, будут храниться в файле UTF-8, хоть такой необходимости и нет. В идеале файл должен сохраняться в кодировке UTF-8 [или любой другой кодировке], только когда это нужно».

В данном случае Рей ссылается на то, что сейчас почти все вызовы SaveToFile, LoadFromFile, SaveToStream и LoadFromStream (а также другие подобные вызовы) принимают название кодировки в качестве второго дополнительного параметра. Если вы явно не задаете кодировку, то используется кодировка по умолчанию.

Задать кодировку можно с помощью свойств или функций класса TEncoding, который входит в модуль SysUtils. Разработчику доступны классы TEncoding для ASCII, UTF-8 и Unicode.

О необходимости контролировать кодировку файла или потока также пишет один из анонимных соавторов. «Мы сохраняем все данные о конфигурации в текстовых потоках, и при компиляции в Delphi 2009 размер файла увеличился с 90 до 130 КБ из-за использования Unicode. Обратив внимание на то, что текст TChart (класс TeeChart) был записан с помощью однобайтовых символов, мы перекодировали многобайтовые символы в однобайтовые и сэкономили за счет этого дисковое пространство».
Рей Конопка продолжает мысль: «Мы не хотели хранить все текстовые файлы в формате UTF-8. Лучше всего было бы обрабатывать файлы так, как это делает среда Delphi IDE. Другими словами, если элемент содержит символ Unicode, файл сохраняется в формате UTF-8. Если же в нем присутствуют только символы ANSI, то используется кодировка по умолчанию».

Чтобы проиллюстрировать свой подход, Рей предоставил следующий образец программного кода:

```
{$IFDEF UNICODE}
    UseANSI := lstPreview.Items.Text =
                 UnicodeString( AnsiString(lstPreview.Items.Text ) );

    if UseANSI then
      lstPreview.SaveToFile( dlgSave.FileName, TEncoding.Default )
    else
      lstPreview.SaveToFile( dlgSave.FileName, TEncoding.UTF8 );

{$ELSE}
    lstPreview.SaveToFile( dlgSave.FileName );
{$ENDIF}
```

Он поясняет: «Если список содержит символы Unicode, то после преобразования Items.Text в AnsiString и назад в UnicodeString будет возвращена строка, отличная от исходной строки в формате Unicode. Значит, мы должны сохранять файл в кодировке UTF-8. Если преобразование строки не приводит к потере данных, то исходная и возвращенная строки будут одинаковы и файл можно сохранять в формате ANSI».

Компания Embarcadero опубликовала список подпрограмм ввода-вывода, которые могут принимать параметр TEncoding, позволяющий указать кодировку. Его можно загрузить со следующей веб-страницы (адрес длинный, и мне пришлось вставить разрыв строки):

```
http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN
/html/devwin32/usingtencodingforunicode_xml.html
```

Д.-Д. Малин (J. D. Mullin), руководитель группы НИОКР в составе проекта Advantage Database Server (издатель: Sybase iAnywhere), поделился своим методом восстановления сохраненной ранее информации. Как и анонимный соавтор, предлагавший сохранять файл в формате ANSI, г-н Малин проверяет, что сохраненные данные ANSI были восстановлены правильно. Он пишет: «Считать строковые данные ANSI из потока в строковый буфер нельзя. Нужно вначале явно считать [их] во временный буфер ANSI».

Этот прием иллюстрирует следующий пример:

```
function SReadString(S: TStream): String;
var
  sLen: LongInt;
  temp: AnsiString;
begin
  sLen := SReadLongint(S);
  SetLength( temp, sLen );
  s.ReadBuffer( temp[1], sLen);
  result := temp;
end;
```

Как объясняет г-н Малин, «потребность во временном буфере ANSI возникает из-за того, что метод ReadBuffer автоматически определяет тип целевого буфера и действует соответствующим образом. Если вы храните данные ANSI в файле или потоке, то их нужно считывать в буфер ANSI, а не в буфер Unicode. Если передать буфер Unicode методу ReadBuffer, то он посчитает, что вы хранили данные в формате Unicode и считает их как данные Unicode».

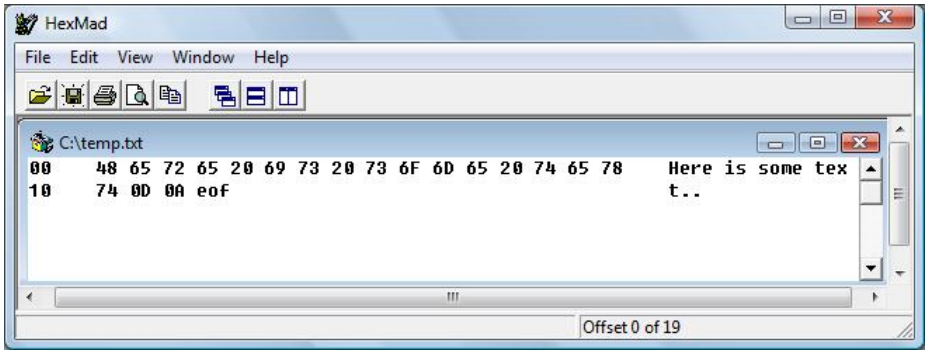
Ларс Дибдал также поделился своим мнением по поводу чтения и записи файлов: «Многие существующие подпрограммы ввода-вывода создавались для работы с кодировкой UTF-8 в Delphi 2007, а это означает, что логика и функции хранения были рассчитаны на обработку данных формата UTF-8 в строках AnsiString».
«Мы решили изъять из алгоритма операции преобразования UTF-8 и выполнять преобразование в точке ввода-вывода, чтобы все функции обработки текста использовали тип UnicodeString. Например, тип TStringList прекрасно работал с форматом UTF-8 в Delphi 2007, но в Delphi 2009 он использует UnicodeString. Нужно преобразовать данные UTF-8 в формат UnicodeString как можно раньше и уж точно перед передачей их в TStringList».
Перед тем как закончить рассмотрение вопроса чтения и записи данных, нам нужно рассмотреть еще два технических приема. Но вначале мы обратимся к теме, которая уже была кратко упомянута в настоящем документе. Я говорю о метке порядка следования байтов, или BOM (byte order mark).

МЕТКА ПОРЯДКА СЛЕДОВАНИЯ БАЙТОВ (BYTE ORDER MARK)

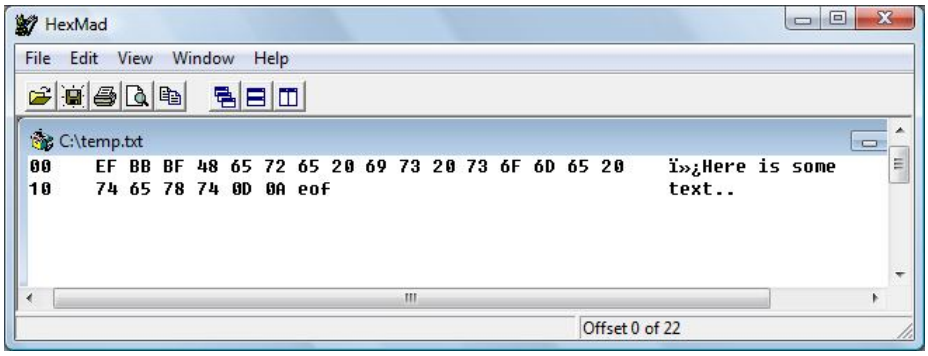
Метка порядка следования байтов — это заголовочная часть (преамбула), которая может присутствовать в текстовом файле и указывает используемую кодировку. Если вы используете методы Delphi SaveToFile (или похожие методы, позволяющие задать кодировку), то Delphi при необходимости запишет метку BOM в первых двух байтах файла. Приведем для иллюстрации следующий фрагмент кода:

```
procedure TForm1.SaveWithEncodingClick(Sender: TObject);
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  try
    sl.Text := TextEdit.Text;
    ListBox1.Items.Clear;
    ListBox1.Items.AddStrings(sl);
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] = 'ASCII' then
      sl.SaveToFile('c:\temp.txt', TEncoding.ASCII)
    else
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] = 'UTF-8' then
      sl.SaveToFile('c:\temp.txt', TEncoding.UTF8)
    else
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] =
      'UTF-16 LE (Little-endian)' then
      sl.SaveToFile('c:\temp.txt', TEncoding.Unicode)
    else
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] =
      'UTF-16 BE (Big-endian)' then
      sl.SaveToFile('c:\temp.txt', TEncoding.BigEndianUnicode);
  finally
    sl.Free;
  end;
end;
```

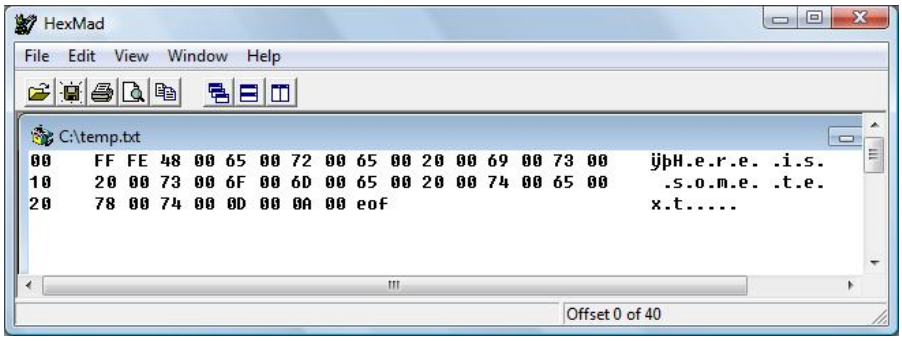
Открыв файл в окне низкоуровневого средства для просмотра в шестнадцатеричном формате (в данном случае HexMad) с помощью кодировки ASCII, получим следующий результат:



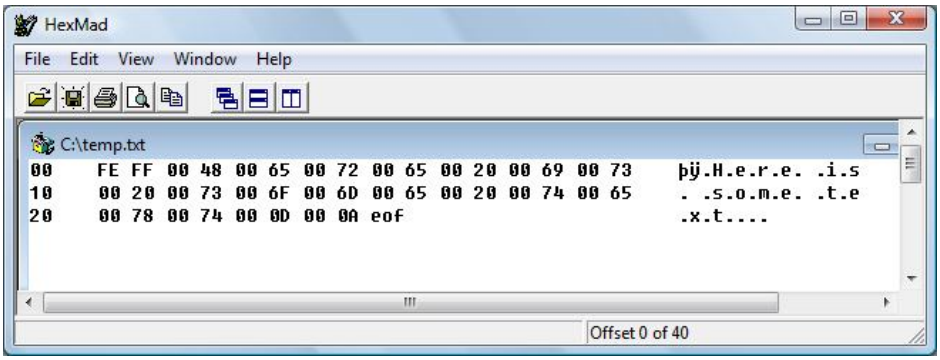
С кодировкой UTF-8 файл будет выглядеть таким образом:



«EF BB BF» — это метка порядка следования байтов, которая в данном случае указывает, что файл имеет кодировку UTF-8 (Unicode). Для сравнения: если выбрать кодировку UTF-16 (с прямым порядком байтов), файл примет следующий вид (считать метку BOM можно с помощью метода TEncoding.GetPreamble):



В кодировке UTF-16BE (с обратным порядком байтов) файл будет выглядеть так:



Обратите внимание, что заголовок в формате UTF-16 имеет длину всего 2 байта. С прямым порядком байтов (по умолчанию) она представлена символами FF FE, а с обратным порядком байтов — FE FF (наоборот).

Метка BOM не является обязательной

В книге *«Delphi 2009 Handbook»* (стр. 91) Марко Канту пишет: «Рекомендация № 1: при каждом сохранении в файл записывайте метку BOM, чтобы был известен формат данных. Работая в памяти, сделать это теперь нетрудно, поскольку если вы не помните формат, то потоковый код Delphi все равно добавит надлежащую метку BOM даже в поток в памяти».

Как мы уже увидели раньше в этом разделе, операция определения кодировки является достаточным условием для того, чтобы была записана правильная метка BOM. Далее Марко объясняет, что при чтении файла или потока нет необходимости указывать кодировку, поскольку Delphi распознает ее по метке BOM, созданной при записи данных. Более того, не следует указывать кодировку при считывании файла, записанного с помощью конкретной кодировки: Delphi позволит вам записать данные в одной кодировке, но не создаст исключение, если вы попытаетесь считать эти данные, используя другую кодировку (хотя данные, вероятно, будут неправильными).

Ситуация усложняется тем, что метка BOM не является обязательным атрибутом файла данных в формате Unicode. Ларс Дибдал поясняет: «Многие средства и приложения не способны читать файлы, содержащие метку BOM. Например, если ваше приложение создает файл конфигурации для серверного приложения Linux, то в работе последнего произойдет сбой при попытке считать метку BOM. Есть также средства чтения XML, которые могут считывать файлы XML в формате UTF-8 только при условии отсутствия в них метки BOM. Аналогично: если вы записываете текст в двоичную структуру (например, в поле большого двоичного объекта в базе данных), то средство чтения может не понять эту метку».
Таким образом, вам может повстречаться созданный другим источником файл в формате Unicode, но без метки BOM. Эту потенциальную проблему также подметили соавторы Д.-Д. Малин и Луис Кесслер (Louis Kessler). На сайте stackoverflow.com Луис прямо задал следующий вопрос: «Как узнать кодировку, если метка порядка следования байтов отсутствует?» Полученные ответы привели его к разработанному Николаем Яковлевым (Nikolaj Yakowlew) модулю CharSet

Detector с открытым исходным кодом, который анализирует комбинацию символов в файле и предсказывает кодировку. Я лично не тестировал CharSet Detector, но при желании вы можете самостоятельно заняться этим, загрузив модуль с веб-сайта http://chsdet.sourceforge.net/.

В заключение раздела, посвященного чтению и записи данных, следует хотя бы упомянуть пару новых классов для чтения и записи потоков в Delphi 2009 и более поздних версиях. Классы TStreamWriter и TStreamReader функционально идентичны своим эквивалентам в .NET. Оба класса позволяют указать желаемую кодировку при записи в поток или чтении из него.

Использование внешних библиотек и сторонних компонентов

Рассмотренные выше темы однозначно свидетельствуют о том, что существует ряд областей, требующих внимания при миграции исходного кода. К счастью, вы, скорее всего, располагаете некоторыми преимуществами, которые помогут вам при выполнении этой задачи. Вы хорошо ориентируетесь в своем программном коде, ведь он написан в вашем стиле либо в стиле, установленном корпоративной политикой, а также имеете доступ ко всему исходному коду.

Эти преимущества испаряются, когда речь заходит о внешних библиотеках и сторонних компонентах. Возможно, у вас даже не будет исходного кода, что делает вас беззащитным перед силами, которые вам неподконтрольны.

В настоящем разделе мы будем заниматься программным кодом, находящимся вне зоны вашего влияния. Начнем, пожалуй, с интерфейса Windows API. Затем рассмотрим сторонние компоненты. В завершение мы поговорим о внешних библиотеках, исходным кодом которых вы располагаете, а также библиотеках, которые были написаны вами или вашей группой либо получены из ресурса с открытым исходным кодом.

Windows API

Очевидно, что ваш программный код работает не в вакууме. Ему всегда приходится взаимодействовать с внешним кодом, как минимум это операционная система и модули, которые ее поддерживают. В ОС Windows такие библиотеки носят название Windows API, или интерфейс прикладного программирования.

Что касается поддержки Unicode, то это хорошая новость. Операционные системы Windows, по крайней мере начиная с Windows 2000, полностью поддерживают Unicode, причем даже более ранние версии способны работать с многобайтовыми символами. Кроме того, большая часть интерфейсов Windows API, которые связаны со строками, имеют как минимум две версии каждой подпрограммы: одна для строк, использующих кодовые страницы Windows (ANSI), а другая для длинных строк. Это сразу видно при анализе модуля импорта, например windows.pas, который включает в себя подпрограммы для работы со строками в версиях A и W.

Вот типичный пример набора внешних объявлений в модуле Windows:

```
function GetModuleFileName; external kernel32 name 'GetModuleFileNameW';
function GetModuleFileNameA; external kernel32 name 'GetModuleFileNameA';
function GetModuleFileNameW; external kernel32 name 'GetModuleFileNameW';
```

Этот пример говорит нам, что GetModuleFileName — функция, возвращающая полное имя исполняемого файла (консольное приложение, приложение Windows или библиотека DLL), имеет три версии. Версия GetModuleFileNameA (ANSI) принимает буфер символов ANSI (в нем хранится путь), версия GetModuleFileNameW принимает буфер символов UTF-16, а GetModuleFileName — это псевдоним версии W. Как видно из приведенного фрагмента кода, функция GetModuleFileName фактически вызывает версию W.

Интересно, что в выпусках Delphi, предшествовавших Delphi 2009, модуль windows.pas содержал следующие объявления:

```
function GetModuleFileName; external kernel32 name 'GetModuleFileNameA';
function GetModuleFileNameA; external kernel32 name 'GetModuleFileNameA';
function GetModuleFileNameW; external kernel32 name 'GetModuleFileNameW';
```

Другими словами, у нас уже давно есть доступ к большинству функций для работы со строками ANSI и многобайтовыми строками, разница лишь в том, что теперь по умолчанию применяется многобайтовая версия функции.

В общем, это означает, что если есть две версии интерфейса Windows API (ANSI и многобайтовая), а вы используете встроенные типы String, Char и PChar (с поддержкой Unicode), то миграция существующего кода пройдет безболезненно. Рассмотрим следующий фрагмент кода:

```
var
  Path: array[0..255] of Char;
begin
  GetModuleFileName(HInstance, Path, Sizeof(Path));
  Label1.Caption := Path;
```

Этот программный код функционирует и компилируется как в ранних версиях Delphi, так и в Delphi 2009 и более поздних версий. Только в Delphi 2009 и более поздних версиях будет вызвана функция для многобайтовых строк, а в ранних версиях — функция для строк ANSI.

С другой стороны, если вы приводите данные к типу AnsiString (заменяете String на AnsiString, Char на AnsiChar и т. д.), то просмотрите вызовы интерфейсов Windows API и измените их так, чтобы вызывалась версия ANSI. Например:

```
var
  Path: array[0..255] of AnsiChar;
begin
  GetModuleFileNameA(HInstance, Path, Sizeof(Path));
  Label1.Caption := Path;
```

Если версия ANSI для интерфейса Windows API не предусмотрена, то вам придется искать другое решение. Например, преобразуйте или приведите несовместимые типы данных к типам, которые поддерживаются существующими

подпрограммами. Впрочем, мне неизвестны вызовы Windows API, требующие принятия подобных мер.

Сторонние средства

В отличие от интерфейсов Windows API, являющихся неотъемлемой частью наших приложений, сторонние компоненты — это лишь средства, позволяющие нам сокращать время разработки и улучшать функциональность приложений. Их можно назвать палкой о двух концах.

Используя сторонний компонент, чтобы обеспечить значительную часть функциональных возможностей своего приложения, мы вступаем в отношения со сторонним поставщиком. Когда мы решаем мигрировать приложение в более новую версию Delphi, то это должно происходить рука об руку с обновлением стороннего компонента.

К счастью, среди поставщиков сторонних компонентов для Delphi есть те, которые отличаются особой одаренностью и надежностью. Например, четыре соавтора настоящего документа — Дэвид Бернеда (Steema Software), Рей Конопка (Raize Software), Д.-Д. Малин (Sybase iAnywhere) и Нард Мозели (Digital Metaphors) — смогли обновить свои продукты, добавив поддержку Delphi 2009 и сохранив совместимость с более ранними версиями Delphi. Благодаря этому их компании продолжают успешно развиваться и заслуженно пользуются доверием заказчиков.

С другой стороны, никаких гарантий нет. Через несколько лет поставщик может просто исчезнуть. И если в вашем приложении использовались компоненты такого поставщика, то последствия в лучшем случае будут неприятными, а в худшем — катастрофическими.

Я знаю много разработчиков, которые ни за что не будут полагаться на сторонние компоненты, если только у них не будет возможности купить исходный код, причем с таким условием, чтобы иметь право использовать этот исходный код после того, как поставщик прекратит поддержку продукта.

Наличие исходного кода не всегда решает проблему, но это, по крайней мере, хорошее начало. С другой стороны, если исходного кода нет, то у вас остается только два варианта: либо удалить сторонние компоненты из своего приложения, либо смириться с мыслью о том, что придется поддерживать приложение в самой последней версии Delphi, которую поддерживают компоненты. Удаление компонентов из приложения — процесс, мягко говоря, болезненный. Навсегда остаться со старой версией компилятора — это ужасно; пойти на такое можно только в том случае, если жизненный цикл самого приложения близится к концу.

Некоммерческие внешние библиотеки

В завершение я хочу рассмотреть библиотеки, которые не поддерживаются коммерчески, однако вы располагаете их исходным кодом. Это, например, библиотеки, разработанные лично вами или вашей группой, а также библиотеки с открытым исходным кодом.

Миграция пользовательских (т. е. разработанных собственными силами) библиотек имеет одну особенность по сравнению с миграцией обычных приложений: вам нужно не только обеспечить поддержку Unicode внутри библиотеки, но и управлять API. Например, возможно, вам придется реализовать две версии экспортированных функций (ANSI и для многобайтовых символов), которые передают в параметрах данные String и Char.

Библиотеки с открытым исходным кодом имеют аналогичную проблематику и еще одно отличие. В большинстве случаев эти библиотеки поставляются на условиях открытой лицензии GNU или иного подобного соглашения. Перед тем как начинать миграцию некоммерческой библиотеки, на которую у вас нет прав, внимательно прочитайте лицензионное соглашение и в дальнейшем придерживайтесь его положений. (Соавтор Стефан Фризмоз утверждает, что легально обойти условия открытой лицензии GNU или другого подобного соглашения можно путем предоставления проектной группе результатов своей работы.)

Вопросы, связанные с базами данных

Приложения баз данных, которым не нужно хранить или отображать строки Unicode, обычно легко поддаются миграции в версии Delphi с поддержкой Unicode. Если приложение обрабатывает строки Unicode, то ситуация несколько усложняется, о чем и поговорим далее в этом разделе.

А начнем, давайте, с изменения, которое не связано напрямую с Unicode, но может привести к возникновению проблем при миграции старого приложения базы данных в Delphi 2009 или более поздней версии. Это изменение касается закладок.

Закладка является ссылкой на позицию записи в наборе TDataSet и служит для перемещения (идентификация записи, к которой нужно будет вернуться позже). В Delphi 2009 появилось два изменения, касающиеся закладок. Строковая закладка TBookmarkStr переведена в разряд устаревших, и больше не должна использоваться. Во-вторых, изменился тип TBookmark.
Г-н Малин объясняет это такими словами: «Компания [Embarcadero] решила изменить тип TBookmark с указателя на TBytes. Это не повлияет на большинство приложений, которые просто используют методы GetBookmark[, GotoBookmark и] FreeBookmark из класса TDataSet. Если же вы обращаетесь с указателем, который получаете от метода GetBookmark, несколько легкомысленно, то берегитесь. Нам пришлось модифицировать многие автоматические тесты, чтобы обеспечить их единообразную обработку».

Не знаю, что значит легкомысленно обращаться с TBookmark, но г-н Малин, безусловно, прав. Закладка отмечает позицию записи в наборе TDataSet, чтобы вы могли быстро вернуться в нее с помощью метода GotoBookmark. Если вы используете закладки в иных целях, то тщательно протестируйте свой код.

Что же касается миграции в формат Unicode, то, к сожалению, с приложениями баз данных не все так просто. Вы, возможно, помните приведенные ранее слова одного из соавторов: «Самая большая проблема, с которой я столкнулся [при миграции на Delphi 2009], состояла в том, что приложение уже было совместимо с

Unicode благодаря применению типа WideString». И это, естественно, не единственный случай.

В книге *«Delphi 2009 Handbook»* Марко Канту отмечает, что до Delphi 2009 поддержка Unicode в классах TDataSet и TField обеспечивалась путем использования типа TWideString. Начиная с Delphi 2009 многобайтовые строковые типы объявляются как String (или явно как UnicodeString). Благодаря этому чтение и запись данных Unicode в классах TField осуществляется по правилам, установленным для типа UnicodeString, и устраняется ряд потенциальных проблем с преобразованием данных, однако некоторые названия классов и членов в TDataSet все еще могут сбить вас с толку. Например, тип TUnicodeStringField не предусмотрен и класс TStringField сохраняет свое значение как AnsiString. Для получения значения TField в формате Unicode используется TWideStringField (сохраняется как UnicodeString в Delphi 2009 и более поздних версиях, см. предыдущий абзац). Как правило, это все не затрагивает обычные приложения баз данных. Так, если до выпуска Delphi 2009 ваша база данных не поддерживала Unicode, то вам не нужны были классы WideString (например, TWideStringField) и, следовательно, не придется преобразовывать их в UnicodeString.

С другой стороны, если вы реализовали в базе данных поддержку Unicode еще до выпуска Delphi 2009, то проанализируйте использование типов WideString и убедитесь, что оно соответствует определениям типа UnicodeString в Delphi 2009 и более поздних версиях.

Например, Марко пишет, что метод TDataSet.GetFieldNames в Delphi 2006 и Delphi 2007 возвращает значение TWideStrings *(«Delphi 2009 Handbook»,* стр. 333). Если вы вызывали этот метод и присваивали значение переменной TWideStrings, то теперь при компиляции приложения в Dephi 2009 или более поздней версии в момент присвоения тип WideString будет преобразован в UnicodeString. Марко рекомендует «…переписать программный код путем выявления всех экземпляров классов TWideStrings и TWideStringList и преобразования их в желаемый тип TStrings или TStringList соответственно».

Интересно, что мои соавторы очень редко затрагивали тему миграции баз данных. Надеюсь, это означает, что в большинстве случаев миграция приложения базы данных в формат Unicode не приводит к возникновению проблем в самой базе данных. Последние два комментария по поводу баз данных и миграции в формат Unicode поступили от Ларса Дибдала.

Первый из них является рекомендацией. Ларс советует: «Перед миграцией базы данных в формат Unicode убедитесь, что средства базы данных поддерживают Unicode». Другими словами, если ваши средства не поддерживают Unicode, то не следует вводить в базу данные в формате Unicode.

Чтобы предотвратить ввод символов Unicode в базу данных, которая не поддерживает этот формат (и избежать потери данных), требуется определенное превентивное планирование. Предположим, например, что пользователи могут редактировать данные с помощью элементов управления, распознающих данные. Большинство таких элементов управления поддерживают данные WideString.

Я бы посоветовал следующее: запустите приложение в среде Delphi 2009 или более поздней версии и посмотрите, что произойдет, если вы через пользовательский интерфейс намеренно добавите в базу данных символ Unicode. Попробуйте выполнить какое-то действие над этими данными, например, отобразить их в отчете.

Нужно признать, что мы не в состоянии проконтролировать всю вводимую пользователями информацию, и трудно сказать, что хуже: ввод данных в формате Unicode или ввод неправильных данных. Бессмысленные входные данные — бессмысленный результат.

С другой стороны, если ввод в приложение данных в формате Unicode приводит к недопустимым последствиям (нарушение доступа и т. д.), то, возможно, следует изменить способы сбора информации и проверять ее пригодность перед фактическим добавлением в базу данных. Например, можно применять компонент ClientDataSet как промежуточный уровень между пользовательским интерфейсом и базой данных. ClientDataSet поддерживает класс TWideStringFields, который может кэшировать пользовательские данные до выполнения проверки.

Когда сбор информации завершен, то перед тем как записывать ее из ClientDataSet в базу данных, вы можете проверить пригодность данных для строковых полей. Например, вспомните прием, предложенный ранее в настоящем документе Реем Конопкой: преобразуйте данные TWideStringField в AnsiString, а затем назад и проверьте, происходит ли при этом потеря данных. Если нет, значит, информацию можно записывать в базу данных.

В завершение Ларс рассказывает об одной проблеме с доступом к данным, которая возникла из-за поддержки Unicode в Delphi. Он пишет: «Чтобы обрабатывать с помощью IBX данные Firebird в формате Unicode, нужно было установить исправление для IBX. Но дольше всего пришлось повозиться с большими двоичными объектами (blob). Иногда они содержат двоичные данные, а иногда текст, в Delphi 2007 это вообще не имеет значения, и в обоих случаях можно использовать [метод] AsString».

«Теперь же такие объекты нужно обрабатывать раздельно: текстовые поля в формате UTF-16 UnicodeString, а двоичные данные — в формате RawByteString. Мы продублировали многие процедуры для работы с большими двоичными объектами, создав их версии для RawByteString и для String, а затем выбирали процедуру с учетом типа поля. Кроме того, были добавлены функции для правильного извлечения больших двоичных объектов из IBX».

Рассказ Ларса относится к конкретной базе данных Firebird и драйверу IBX, однако позволяет сделать интересный вывод, касающийся миграции в формат Unicode. Как мы узнали в предыдущем разделе, при использовании чужого программного кода можно столкнуться с проблемами, не являющимися результатом наших ошибок. Тем не менее, чтобы миграция прошла успешно, эти недостатки все равно нужно устранять.

Заключение

Нард Мозели пишет: «Как все неизвестное, переход на Unicode вначале пугает. Приходится мучительно осваивать новые концепции. Но в действительности большая часть исходного кода требует лишь незначительного изменения либо может использоваться в исходном виде. Даже доступ к базам данных и вызов многих библиотек Windows API, как правило, функционирует. Если следовать рекомендациям разработчиков Delphi и пользоваться доступными средствами, такими как предупреждения компилятора Delphi и функция поиска, то переход на Unicode можно выполнить просто и безболезненно».

Вспоминая замечания многочисленных соавторов этого документа, хочу предостеречь вас, что не всегда миграция в формат Unicode происходит так легко, как пишет Нард. Все зависит от сложности и предназначения приложения, компонентов, с которыми оно взаимодействует, а также степени внедрения поддержки Unicode в прошлом (с помощью устаревшего типа WideString).

Тем не менее, несмотря на все трудности, которые может доставить миграция, ее осуществление позволит вам значительно продлить срок жизни своего приложения. Вы сможете обновить интерфейс, например, добавить поддержку функций Windows 7, а также подготовить приложение к запланированному усовершенствованию Delphi в будущем (межплатформенная компиляция, внутренняя поддержка 64-разрядных вычислений и пр.).

Выражение благодарности

Этот документ появился благодаря вдохновенному труду и поддержке многих людей. Я глубоко благодарен всем, кто оказывал мне посильную помощь. Я говорю спасибо Майку Розлогу (Mike Rozlog), генеральному директору Delphi Solutions, за предложение написать документ, посвященный миграции в формат Unicode, и Тиму Дель Кьяро (Tim Del Chiaro), Embarcadero Technologies, который приложил немало усилий для создания документа. Также я благодарю Сеппи Блума и Тома Гердеса, разработчиков Delphi из компании Embarcadero Technologies, за предоставленные в ходе работы замечания и техническую поддержку.

Я признателен моим многочисленным соавторам, которые делились опытом перехода на Unicode, давали советы и оказывали необходимую помощь. Вот их имена: Дэвид Бернеда, Марко Канту, Редж Клотье, Роджер Конелл, Мариано Винсент де Уркиса, Ларс Дибдал, Стефан Фризмоз, Луис Кесслер, Рей Конопка, Олаф Моньен, Нард Мозели, Д.-Д. Малин, Джаспер Потьер, Стив, Боб Сворт, Джейкоб Турман, а также несколько человек, которые пожелали остаться неизвестными.

Кроме того, я благодарю Лоя Андерсона (Loy Anderson) из компании Jensen Data Systems, который помогал вычитывать и редактировать документ. Отдельная благодарность Ларсу Дибдалу, Стефану Фризмозу и Такеши Арисава (Takeshi Arisawa) из компании Embarcadero Technologies за техническое рецензирование черновых версий.

Дополнительные сведения

Ниже перечислены ресурсы, которые могут быть вам полезны. Я сожалею, что не сделал полный список прочитанных блогов, документов и форумов по Unicode. Тем не менее наиболее важные из них представлены в этом разделе.

Блоги и документы, посвященные проблемам с Unicode, Delphi и программным обеспечением

http://blogs.embarcadero.com/abauer/2008/01/28/38853
http://blogs.embarcadero.com/abauer/2008/01/10/38847
http://blogs.embarcaderor.com/abauer/2008/01/09/38845
http://www.micro-isv.asia/2009/03/make-sure-your-web-site-is-always-displayed-with-the-right-characters/
http://www.micro-isv.asia/2009/03/why-not-use-utf-8-for-everything/
http://www.micro-isv.asia/2008/12/choose-the-right-file-format-for-your-delphi-source-code/
http://www.micro-isv.asia/2008/10/delphi-2009-string-performance-in-a-nutshell/
http://www.micro-isv.asia/2008/10/needless-string-checks-with-ensureunicodestring/
http://www.micro-isv.asia/2008/09/speed-benefits-of-using-the-native-win32-string-type/
http://jdmullin.blogspot.com/2008/09/tips-when-porting-delphi-application-to_16.html
http://www.bobswart.nl/Weblog/Blogs.aspx?RootId=2:2947
Delphi в мире Unicode, часть I: что такое Unicode, зачем он нужен и как с ним работать в Delphi? Автор: Ник Ходж (Nick Hodge)
http://edn.embarcadero.com/article/38437
Delphi в мире Unicode, часть II: новые функции и классы RTL для поддержки Unicode. Автор: Ник Ходж
http://edn.embarcadero.com/article/38498
Delphi в мире Unicode, часть III: приведение кода в соответствие со стандартом Unicode. Автор: Ник Ходж
http://edn.embarcadero.com/article/38693
Delphi и Unicode. Автор: Марко Канту
http://www.embarcadero.com/images/dm/technical-papers/delphi-and-unicode-marco-cantu.pdf
http://thedorictemple.blogspot.com
http://www.beholdgenealogy.com/blog

Веб-страница на сайте Майкрософт, посвященная кодовым страницам

http://msdn.microsoft.com/en-us/library/dd317752(VS.85).aspx

Консорциум Unicode

http://unicode.org/

Книги

Delphi 2009 Handbook (2009 г., Марко Канту), распространяется через Amazon.com и Lulu.com.

Delphi 2009 Development Essentials (2009 г., Боб Сворт), распространяется через Lulu.com.

Об авторе

Кэри Дженсен — президент компании Jensen Data Systems, Inc. из Хьюстона, которая занимается разработкой программного обеспечения, а также предоставляет образовательные и консалтинговые услуги. Он является соавтором двадцати популярных бестселлеров, включая книги, посвященные Advantage Database Server, Delphi, Kylix, Oracle JDeveloper, JBuilder и Paradox. Г-н Дженсен часто выступает на конференциях, симпозиумах и семинарах в Северной Америке и Европе. В Университете им. Райса он получил степень кандидата психологических наук в области влияния человеческого фактора и специализируется на человеко-машинном взаимодействии.

Замечания и пожелания

В мои намерения входит обновление настоящего документа в будущем. Поэтому я приглашаю всех присылать свои замечания, исправления и пожелания, которые могут быть использованы для улучшения и дополнения документа. Свои предложения присылайте на адрес cjensen@jensendatasystems.com, указав в поле темы «Unicode migration». В течение недели я обязательно отвечу на все сообщения. Если вы не получили ответ, значит, ваше сообщение где-то потерялось. Пришлите его еще раз.

Компания Embarcadero Technologies, Inc. — ведущий поставщик средств для разработки приложений и работы с базами данных, чьи продукты отмечены многочисленными профессиональными наградами. Решения Embarcadero позволяют с легкостью проектировать, оперативно разрабатывать и эффективно эксплуатировать системы вне зависимости от платформы и языка программирования. Девяносто компаний из списка Fortune 100 и более трех миллионов разработчиков по всему миру активно используют продукты Embarcadero, повышая тем самым продуктивность своего труда, сокращая расходы, упрощая управление изменениями и соблюдение нормативных требований, а также обеспечивая быстрое внедрение инноваций. Основными продуктами компании Embarcadero являются Embarcadero® Change Manager™, Embarcadero® RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® и Rapid SQL®. Компания Embarcadero была основана в 1993 году, ее штаб-квартира расположена в Сан-Франциско, а отделения открыты во многих странах мира. Сайт компании Embarcadero: www.embarcadero.com.