テクニカル ノート

Delphi における一般的な Unicode への移行テクニック：最前線からの体験談と助言

Jensen Data Systems, Inc.　Cary Jensen

2009 年 12 月

本社
100 California Street, 12th Floor
San Francisco, California 94111

EMEA 本部
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

日本
東京都千代田区飯田橋 4-7-1
ロックビレイビル 8F

概要

Embarcadero Technologies がリリースした Embarcadero® RAD Studio 2010 により（Delphi 2009 のリリースからですが）、開発者は Delphi® および C++Builder® を使って、Unicode 対応した一流のアプリケーションをそれぞれの顧客に届けられるようになりました。この重要な開発によってソフトウェアの新しい市場が開けはしましたが、既存のアプリケーションや開発手法に課題が突きつけられる場合もあります。特に顕著なのは、文字列のサイズを想定して処理を行っているコードです。

このホワイト ペーパーは、既に同じ道を歩んできた数多くの Delphi 開発者の経験や知識を共有することで、Unicode 移行作業の指針を示すことを目的としています。最初に問題の概略を紹介し、それから Unicode の基礎知識を簡単に説明します。その後、注意が必要なアプリケーションのさまざまな側面について系統的に検討し、現実の経験を踏まえて例や提案を示します。ホワイト ペーパーの最後には、Unicode 移行作業に役立ちそうな参考文献の一覧を挙げています。

はじめに

Embarcadero は、2008 年 8 月に初めて、RAD Studio で完全に Unicode をサポートしました。それにより、今後長期間にわたって Delphi と C++Builder が Windows プラットフォーム上でのネイティブ アプリケーション開発の最前線にとどまることが確実になりました。

しかし、これまで何年もの間に Delphi に導入されてきた主要な機能強化の多く、たとえばバリアントやインターフェイス（Delphi 3）、フレーム（Delphi 5）、関数のインライン化やクラスのネスト（Delphi 2005）、ジェネリックス（RAD Studio 2009）などとは違って、Unicode 対応は、これまで Delphi でサポートされてきた機能に単に新しい機能を追加するだけのものではありません。逆に、ほとんどすべての Delphi アプリケーションで使われている基本データ型のいくつかが、根本的に変更されています。具体的には、String 型、Char 型、PChar 型の定義が変更されました。

これらの変更は、軽い気持ちで行われたのではありません。変更によって既存アプリケーションにどのような影響が出るか、今後の開発にどのような影響が出るかを、幅広く検討した上での導入です。それだけでなく、Embarcadero では、Delphi をサポートし普及させてくれている技術パートナーから情報や助言を募りました。

実際のところ、Unicode サポートを実装しようとすると、いくらかの不都合が生じるのは避けられません。このホワイト ペーパーへの寄稿者の 1 人（本人の希望で Steve と呼ぶことにします）は、次のように言っています。「PChar や String の意味を変えるべきではなかったと思います。ただ、Delphi の設計・開発チームがどのような選択をしたとしても批判は免れなかったでしょう。言わば勝ち目のない状況だったのです。」

最終的に、String と Char と PChar の意味を変更するのが、影響がないわけではありませんが最も混乱が避けられる方法だと判断されました。プラス面を見ると、グラフィカル インターフェイスとそこで操作するデータの両方をグローバル化を意識したやり方で扱う、世界に通用するアプリケーションを RAD Studio 開発者がすぐに構築できるようになるため、ますますグローバル化する市場でアプリケーションを構築し配置することを阻む大きな障壁が取り除かれます。

しかし、マイナス面もあります。String、Char、PChar が変更されたことで、Delphi や C++Builder のこれまでのバージョンで作られてきたアプリケーションや、ライブラリ、共有ユニット、実績のある手法などの移行に、大小さまざまな問題が起きる可能性が生じています。

現実的に考えましょう。既存アプリケーションをアップグレードするときには大抵、移行に関わる問題が生じる可能性があり、結果として既存コードを変更したりサードパーティのコンポーネントやライブラリを新バージョンにアップグレードしたりする必要が生じる可能性があります。RAD Studio 2009 以降へのアップグレードについても同じことが言えます。アップグレードには、簡単なものもあれば困難なものもあります。

そしてここがこのホワイト ペーパーで本当に言いたいことです。Delphi 1（Char および PChar）や Delphi 2（String）の頃から頼っている基本データ型が変更されるのですから、既存アプリケーションを RAD Studio 2009 以降に移行させるのは、これまでの移行に比べて手間がかかってもおかしくありません。

寄稿者である Innova Solutions Pty Ltd の Roger Connell は、「[Delphi チームは Unicode を新しくサポートするという] 立派な仕事を成し遂げたと私は思いますが、Delphi の移行の中で [今回が] 最も困難な移行でした（実のところ本当に困難だったのはこれが初めてでした）」という意見を述べています。幸いなことに、どのような課題にも解決策は存在します。このホワイト ペーパーはそのお手伝いをするためのものです。

私は手始めに、RAD Studio コミュニティに情報の提供を呼びかけました。具体的には、RAD Studio 2009 以降への既存アプリケーションの移行を成功させた開発者の方々に、Unicode 移行についてわかったことや、助言や、体験談などを伝えてくれるよう依頼しました。受け取った回答は非常に興味深いものでした。

考え得るほとんどすべてのタイプの開発者から回答を頂きました。独立した個人の開発者の人もいれば、開発チームに属している人もいました。垂直市場製品を作成している人も、社内アプリケーションを構築している人も、アプリケーション開発者向けに非常に有名なサードパーティ コンポーネントやツールを公開している人もいました。また、Delphi の世界で尊敬を集めている著名人や、会議で講演をしたり私たちのほとんどが読んでいる本を執筆している開発者からも回答を頂きました。

同じように、その体験談や助言や取り組み方もさまざまでした。移行プロジェクトは予想以上に簡単だったという人もいますし、困難だったという人もいます。特に、長い間使われてきたアプリケーションや、幅広い手法やソリューションを使用しているアプリケーションの場合は困難だったようです。

個々の移行が簡単だったか困難だったかとは別に、共通する取り組み方、実践的なソリューション、検討が必要な事柄がいくつか見えてきました。ここではそれを皆さんに紹介したいと思います。

しかし、このホワイト ペーパーを公開したらすべてが終わるわけではありません。Unicode 移行の成功事例の収集は引き続き行って、いずれこのホワイト ペーパーを更新したいと思っています。ですから、これを読んで何かを思いついた方、このホワイト ペーパーを補間/拡張したい方は、ぜひ寄稿者として参加頂けたらと思います。これについては、ホワイト ペーパーの最後で詳しくお知らせします。

次のセクションでは、基本的な Unicode の定義と説明を簡単に行います。Unicode のことを既によく知っていて、UTF-8 および UTF-16 の基本を理解し、コード ページとコード ポイントの違いがわかる人は、このセクションを飛ばすか、見慣れない用語がないかをざっと確認していただくとよいでしょう。

しかし、先に進む前に言っておきたいことがもう 1 つあります。RAD Studio の Unicode サポートには、構築するアプリケーションについて、はっきりと区別はできるけれども補完的な 2 つの意味合いが含まれています。1 つ目は、Delphi 2009 以降で書かれたコードと Delphi のそれより前のバージョンのコードでの、文字列の扱いの違いに関するものです。2 つ目はローカライズ、つまり市場の言語や文化に合わせてソフトウェアを変更する処理に関するものです。

このホワイト ペーパーでは、2 つのうち特に前者を取り上げる予定です。複数の言語や文字セットのサポートを実現する方法は、このホワイト ペーパーの範囲外であり、この後では扱いません。

UNICODE とは

Unicode とは、世界中のすべての文字言語のすべての文字や記号をエンコードして、デジタル コンピュータで格納、検索、表示できるようにするための標準仕様です。制御文字（タブ、改行、改ページなど）と 26 文字のラテン アルファベットからなる印刷可能文字とを表す ANSI（American National Standards Institute：米国規格協会）コードの標準文字セットと同様に、Unicode でもそれぞれの文字に少なくとも 1 つの一意の番号が割り当てられています。

また、これも ANSI コード標準と似ているのですが、Unicode では、通貨、科学表記や数学表記、その他さまざまな外国の文字など、多様なシンボルを表現できます。それだけ多くのシンボル（現段階で 100 万を超えています）を参照できるようにするため、Unicode 文字には最大で 4 バイト（32 ビット）のデータが必要になります。それに対して ANSI コード標準は 8 ビットでエンコードされているため、一度に扱える文字の種類は 255 に限定されます。

Unicode の制御文字、文字、記号にはそれぞれに数値が割り当てられていて、それを**コード ポイント**と呼びます。文字のコード ポイントは、Unicode 技術委員会で割り当てられた後は決して変わりません。たとえば、'A' のコード ポイントは 65（16 進数で $0041。Unicode 表記では U+0041）です。また、どの文字にも一意の

不変の名前が割り当てられます。この場合は 'LATIN CAPITAL LETTER A'（ラテン大文字 A）です。どちらも変更されることは決してないため、現在のエンコードを永久に使い続けることができます。

各コード ポイントは、1 バイト、2 バイト、4 バイトのいずれかで表現でき、一般的なコード ポイントの大半（64K 分）は 2 バイト以下で表現することができます。Unicode の用語で、最初のこの 64K の記号を**基本多言語面**（basic multilingual plane：BMP）と呼びます（BMP という頭字語はこのホワイト ペーパーで何度も使いますので覚えておいてください）。

しかし、Unicode 標準では一部の文字を 2 つ以上の連続したコード ポイントで表現できるため、事が複雑になります。こういった文字のことを合成文字または分解可能文字と呼びます。

たとえば ö という文字は $00F6 と表すことができます。この文字を合成済み文字と呼びます。しかしそれ以外に、o という文字（$006F）の後に分音記号（¨）文字（$0308）を付けて表すこともできます。Unicode の処理規則でこの 2 つの文字を組み合わせて、1 つの文字を作成するのです。

これを以下のコードで実証します。

```
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);
```

合成文字の目的は、Unicode ファイルの内容をより細かく分析できるようにすることです。たとえば、研究者が、どの文字に付いているかに関係なく、分音記号（¨）が使われている頻度を数えようと思った場合には、分音記号を使っている文字をすべて分解して、簡単に数えることができます。

現在割り当てられているすべてのコード ポイント（と今後考え得るすべてのコード ポイント）は十分に 4 バイトで表現できますが、各文字にそれだけ多くのメモリを費やして表現することがどんな場合にも妥当だというわけではありません。たとえば、英語圏のほとんどの人は、かなり少ない文字（100 文字以下程度）しか使いません。

そのため、Unicode では、コード ポイントを表すための異なるエンコード標準もいくつか仕様化していて、それぞれが一貫性や処理や格納などの要件のトレードオフを提供しています。その中でも Delphi でよく見かけるのが UTF-8、UTF-16、UTF-32 です。（UTF とは、人によって主張は異なりますが、Unicode Transformation Format または UCS Transformation Format の略です。）また、時折、UCS-2 や UCS-4（UCS は Universal Character Set の略）を見かけることもあります。

UTF-8 では、コード ポイントを表す整数のサイズに応じて、1、2、3、4 バイトのいずれかでコード ポイントを格納します。これは、サイズが重要な意味を持つ HTML や XML などの標準でよく使われる形式です。具体的に言うと、1 バイトで表現でき、（少なくとも大部分の Web ページでは）HTML の大部分を占めるラテン アルファベットなどの文字は、1 バイトしか使いません。7 ビットで表現できないコード ポイントだけが残りのバイトを使用します（コード ポイントの値が 127 を超えた時点で、UTF-8 は値のエンコードに少なくとも 2 バイトを使うようになります）。余分な処理が必要ではありますが、こうすることで、テキストを表すために必要なメモリ量と、ひいてはネットワーク経由でこの情報を転送するために必要な帯域幅を、最小限に抑えることができます。

UTF-16 は中間的なものです。物理メモリや帯域幅が処理ほど重要でない環境では、BMP 文字はすべて 2 バイト（16 ビット）のデータ（これを**コード単位**と呼びます）で表されます。言い換えれば、BMP のコード ポイントは 1 つのコード単位で表されるということです。

先ほどこのセクションで、UTF-8 が 1、2、3、4 バイトのいずれかで 1 つの Unicode コード ポイントをエンコードできると書きました。UTF-16 に関しては、似ているけれども異なる状況が起こります。アプリケーションで BMP に含まれない文字を表現する必要がある場合です。そういったコード ポイントには 2 つのコード単位（4 バイト）が必要で、この 2 つが一緒になってサロゲート ペアというものを構成します。UTF-16 では、サロゲート ペアを使って 16 ビットに収まらないコード ポイントを表すことができ、この一組のコード単位で 1 つのコード ポイントを一意に識別します。

もう予想できるでしょうが、UTF-32 ではすべてのコード ポイントを 4 バイトを使って表します。物理記憶域の点では最も無駄が大きくなりますが、処理は最も少なくて済みます。さらに、UTF-16 と UTF-32（それから UCS-2 と UCS-4 も）には、ビッグエンディアン（big-endian：BE）とリトルエンディアン（little-endian：LE）の 2 種類があります。ビッグエンディアンのエンコードでは最上位バイトから、リトルエンディアンでは最下位バイトから順に使用します。どちらの方法が使われているかは、通常、エンコードされたファイルの先頭にあるバイト オーダー マーク（Byte Order Mark：BOM）で特定できます。BOM によって UTF-8 と UTF-16 と UTF-32 を区別することもできます。

UTF-16 では 1 文字分が 2 バイトまたは 4 バイトになるのに対して、UCS-2 では必ず 2 バイトです。そのため、UCS-2 では BMP に含まれる文字しか参照できません。別の言い方をすれば、BMP に関しては UCS-2 と UTF-16 は同等です。しかし UCS-2 はサロゲート ペアを認識しないので、BMP 以外の文字を表現することができません。

それに対して UCS-4 は、長さが 4 バイトあり、UTF-32 と同じだけの Unicode コード ポイントを表現できます。ただし、UTF-32 標準ではその他にも Unicode 機能を定義していて、事実上 UCS-4 に代わって使われています。

技術的なことはこれで十分でしょう。次のセクションでは、私たち Delphi 開発者が受ける影響を検討します。

UNICODE 移行と DELPHI アプリケーション

Delphi での Unicode のサポートは Delphi 2009 で始まったのではなく、このリリースで全体に広がっただけです。たとえば Delphi 2007 では、Unicode 対応のサーバーを扱う dbExpress ドライバの多くで Unicode をサポートしています。また、Delphi 2005 以降では、ソース ファイルを UTF-8 形式で保存してコンパイルすることが可能になっています。そして WideString 型（2 バイトの文字列型）も Delphi 3 から含まれています。

実際に、このホワイト ペーパーの寄稿者の 1 人である Steve は、次のように言っています。「[Delphi 2009 へ移行するにあたっての] 最大の問題は、WideString や TNT コントロールを使っているためにアプリケーションが既に Unicode 互換になっていることでした。それにより、まだ String や PChar を使っているアプリケーションよりも困難になっていたと思います。」

Delphi 2009 では状況が根本的に変化しています。たとえば、コンポーネント名や、メソッド名、変数名、定数名、文字列リテラルなどで Unicode 文字列を使うことができます。しかし、ほとんどの開発者にとって、最も大きい変化は文字列データ型と文字データ型でしょう。このセクションでは、まず、文字列型および文字型に対して行われた変更の概要を説明します。その後、変更によって影響を受ける Delphi アプリケーション開発の具体的な部分を取り上げます。

STRING、CHAR、PCHAR

String 型が、UTF-16 文字列である UnicodeString 型で定義されるようになりました。同様に、Char 型は 2 バイトの文字型である WideChar 型、PChar 型は 2 バイトの Char のポインタである PWideChar 型になっています。

この基本データ型の変更で重要なのは、それぞれの文字が少なくとも 1 つのコード単位（2 バイト）で、場合によってはそれ以上で表現されるという点です。

この変更と同時に、文字列のバイト数と文字列中の文字数とが一致しなくなりました（ただし、中国語などのマルチバイト文字セットを使っていた場合を除きます。その場合には、Delphi の新しい Unicode 実装によって実際に状況が単純化されています）。同様に、Char 型の値が 1 バイトではなく 2 バイトになりました。

これまで愛用してきた古い文字列型の AnsiString はまだ存在しています。AnsiString 値は、これまでどおり、1 文字が 8 ビットの ANSI 値を含み、参照がカウントされ、コピーオンライトのセマンティクスが使われます。また、8 ビットの文字型や 8 ビットの文字ポインタが必要な場合には、これまでどおり AnsiChar 型と PAnsiChar 型を使うことができます。

従来の Pascal String すら利用できます。参照がカウントされないこの文字列には、最大で 255 バイトを格納することができます。この文字列は ShortString データ型で定義され、要素内の文字を 1 バイト目から 255 バイト目に、文字列の長さを 0 バイト目の 8 ビットに格納します。

AnsiString 変数を使い続けたいならそれは可能です。従来の多くの文字列操作関数（UpperCase や Trim など）の AnsiString 版が含まれる、AnsiStrings.pas という特別なユニットさえあります。さらに、標準的な文字列関数の多くがオーバーロードされ、AnsiString 版と UnicodeString 版の両方が提供されています。このホワイト ペーパーの何人もの寄稿者の話からわかりますが、実際に、既存の String の宣言を AnsiString の宣言に変換する方法は、従来のコードを移行するのに効果的な手法です。

次のコードを見てください。変数 s が AnsiString として宣言されています。

```
var
  s: AnsiString;
…
```

Delphi 2009 とそれより前のバージョンで異なるのは、次の宣言です。

```
var
  s: String;
…
```

こちらのコードの変数は UnicodeString 型です。UnicodeString 型と AnsiString 型には共通する機能がいくつかありますが、非常に大きな違いもあります。主な共通点は、参照がカウントされ、コピーオンライトの動作をすることです。

参照がカウントされるとは、どのコードが文字列を参照しているかを Delphi が内部的に追跡するという意味です。コードで文字列が参照されなくなると、文字列で使われていたメモリの割り当てが自動的に解除されます。

コピーオンライトによっても効率が向上します。コピーオンライトをサポートしている型（Delphi では動的配列も該当します）では、1 つの値を 2 つ以上の変数で参照している場合、すべての変数はメモリ上の同じ位置を指します。1 つの変数から参照先の値を変更しようとしない限り、それは変わりません。ただし、1 つの変数から参照先の値を変更すると、コピーが作成され、変更はそのコピーだけに対して適用されます。

String と違って、WideString 型は最初に Delphi に導入されたときと同じままです。2 バイト文字参照を表しますが、参照のカウントもされませんし、コピーオンライトもサポートしません。Delphi の FastMM メモリマネージャを使用していないので、パフォーマンスの効率もあまり良くありません。一部の開発者は Delphi 2009 より前のバージョンで WideString を使って Unicode サポートを実装していましたが、WideString は COM 開発のサポートが主な目的で、COM の BSTR データ型に対応するものでした。

AnsiString クラスは、2009 より前の String 型と似た動作をしますが、従来の AnsiString とは 1 つの点で大きく異なっています。内部構造が違うのです。従来の AnsiString をメモリ上に割り付けると、文字列中の 1 文字ごとに 1 バイトと、それ以外に余分に 8 バイトが使われました。余分なバイトのうちの 4 バイトには AnsiString の長さが含まれ、残りの 4 バイトは参照カウントに使われました。

それに対して、新しい AnsiString 型と UnicodeString 型では、文字データを保持するのに必要なメモリの他に、従来の AnsiString よりも 4 バイト多い 12 バイトを追加で使用しています。従来の AnsiString と同様に、最後

の 8 バイトは文字列の長さ（AnsiString の場合は文字数、UnicodeString の場合はコード単位数）と参照カウントに使われます。AnsiString と UnicodeString の両方で使われている追加の 4 バイトのうち、2 バイトは文字の要素サイズを表し、残りの 2 バイトは文字列のコード ページを参照します。

AnsiString の要素サイズは 1 で、UnicodeString の要素サイズは現在のところ 2 です（これは将来的に変わる可能性があります。内部構造に余裕を持たせてあるのはそのためです）。一方、コード ページはさらに複雑な問題なので、この後で文字列変換の問題と併せて説明します。

ファースト ステップ

好材料から始めましょう。既存アプリケーションの中には、変更をほとんど、あるいはまったくしなくても Delphi 2009 以降に変換できるものがあります。VCL コンポーネント（ほとんどの場合、VCL の Unicode サポートは注意深く検討されています）や、Unicode サポートの意味を時間をかけてしっかりと理解しているサードパーティ ベンダのコンポーネントを主に使っている限り、あまり問題はないでしょう。

寄稿者の 1 人で、プログラマでありソフトウェア アーキテクトでもある Rej Cloutier は、自分のアプリケーションで実際に完全な変換を行ったことはないけれども、Delphi 2010 へのテスト移行をしてみたそうです。彼は次のように書いてきました。「難なく移行できたというのが結果です。1 つの理由は、[私たちの] すべての文字列関数が 1 つのユニットにカプセル化されていたことです。[その結果、] 詳しく調査しなければならないユニットは 1 つだけでした（3、4 か所にちょっとした変更が必要でした）。8 つ程の DBMS は問題なくコンパイルできました（それぞれ 100k 行から 185k 行程度のコードが含まれています）。」

もう 1 つ別の例を挙げましょう。私はトレーニング資料で使用するための 数百の Delphi プロジェクトを持っています。その中には、もともと Delphi 1 の頃に記述したプロジェクトもあれば、最近になって Delphi の最新機能のデモ用に作成したプロジェクトもあります。これらのプロジェクトを、私は何年にもわたって、トレーニング資料と一緒に最新状態にアップデートしてきました。そのため、ほとんどのプロジェクトはごく最近、BDS 2006 や RAD Studio 2007 でコンパイルしています。

Delphi 2009 がリリースされてから、私はこれらのプロジェクトのうち 100 以上を Delphi 2009 や Delphi 2010 に移行してきました。その中で変更が必要だったプロジェクトはたったの 5 つ程度で、その変更も主に、DLL 内のルーチンにデータを渡す部分や、テキスト ファイルの読み書きの部分でした。

これは、既存の Delphi アプリケーションを簡単に変換できるという正当な説明になるでしょうか。いいえ、ならないでしょう。寄稿者の Steve が「サンプル コードと実世界の複雑なアプリケーションは比較できないと思います」と主張したのは当然のことです。

しかし、ここからも学ぶべきことはあります。デモ用プロジェクトは、Delphi の機能の使い方を説明するために構築したものであり、パッケージや、DLL、コンポーネント、Delphi の Open Tools API、COM、

DataSnap、ユーザー インターフェイス設計、スレッドとスレッド同期など、さまざまなトピックを扱っています。言い換えれば、これらアプリケーションのほとんどでは、Delphi の RTL（ランタイム ライブラリ）、VCL（ビジュアル コンポーネント ライブラリ）、コンパイラ/デバッガのオプション、統合開発環境、Delphi のエディタといったもののデモを行っていました。そして、その大部分は問題なく動いたのです。つまり、Delphi 環境の Unicode への移行は、一貫していて凝集度が高いということです。

困難になるのは Delphi が直接に管理する領域から外に手を伸ばしたときであり、デモ プロジェクトについての Steve の意見が正しいというのもこれが理由です。実世界のアプリケーションは、通常は機能が豊富であり、オペレーティング システムの機能を直接使うだけでなく、外部のライブラリや、パッケージ、ストリーム、ファイル、コードなどにも依存しています。結果的にそこで問題が発生してしまうのです。

デモ コードと実世界のアプリケーションとのもう 1 つの違いは、移行する価値のある既存アプリケーションのほとんどは、既にいくらかの期間稼働しているという点です。そのため、大抵の場合、パフォーマンスや機能を実現するために当初は重要だったけれども、現在ではもっと優れた代わりの選択肢が存在するような手法が使われています。同様に、これらのアプリケーションは時間が経つ中で、異なる開発者によって異なる方法で書かれてきています。また、元から使用しているサードパーティのツールやライブラリがもうサポートされなくなっている場合もあります。数え上げればきりがありません。

Unicode 移行がどの程度複雑になりそうかの客観的な尺度を探しているなら、寄稿者の Steffen Friismose が Unicode Statistics Tool を検討するよう提案しています。このツールは Embarcadero の Code Central からダウンロードできます。Unicode Statistics Tool は、ソース コードを調査し、Unicode 移行が相対的にどの程度複雑になるかの評価を出力します。このツールと解説は

http://cc.embarcadero.com/item/27398 にあります。

このホワイト ペーパーの多くの寄稿者から受け取った情報を踏まえると、Delphi 2009 以降に既存アプリケーションを移行する際には、次のような問題や手法を検討する必要があります。

- String と Char のサイズ
- AnsiString に戻す
- 文字列変換
- Char の集合
- ポインタ操作とバッファ
- 外部データの読み書き
- 外部ライブラリとサードパーティ コンポーネント
- データベース関連の問題

この後、これらのトピックそれぞれについて検討していきます。

文字列と文字のサイズ

Delphi 2009 より前の時代には、文字列のサイズ（バイト単位）は簡単にわかりました。しかし現在はそれほど簡単ではありません。UnicodeString 型は UTF-16 なので、文字列のバイト数は文字列に含まれる文字数の 2 倍である（Char の長さは 2 バイトなので）という結論になるかもしれません。コードで表現すると次のとおりです。

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

そしてこれは、ほとんどの場合に正しくなります。さらに言うと、次のコードの方がより正確です。

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

こちらの例の方が正確だというのは（ドラム ロールをお願いします）、文字列のサイズについての想定が減っているからです。具体的には、Char のバイト数が 2 であると想定するのではなく、StringElementSize 関数を使って計算しています。

しかし、知りたいのがその文字列に含まれる文字数であった場合には、それほど簡単ではありません。Length 関数で文字列中の文字数が返されると考えたくなるかもしれませんが、それは正しくありません。Length が返すのは UnicodeString 中のコード単位の数です。

この問題を最もよく表しているのが、寄稿者である Unit 4 Agresso の Jasper Potjer の次の疑問です。「5 文字の UTF-16 文字列があり、そこに [1 つの] サロゲート ペアが含まれているとします。Length は、文字 [コード ポイント] の数（5）を返すのですか。それとも 16 ビット ワード [コード単位] の数（6）を返すのですか。」彼は他にも関連する質問をいくつかしてきました。ここでは勝手ながら、彼の質問全体の本質部分を次のように整理してみます。

1. Length 関数は、コード ポイント数を返すのでしょうか、コード単位数を返すのでしょうか。
2. UnicodeString の最初の文字がサロゲート ペアで表現されている場合、MyString[1] にはコード ポイント（文字）が含まれるのでしょうか、コード単位（サロゲート ペアの半分）が含まれるのでしょうか。
3. Char 型はサロゲート ペアを保持することができるのでしょうか。言い換えれば、Char はコード ポイントを保持するのでしょうか、コード単位を保持するのでしょうか。

4. Length 関数がコード ポイントではなくコード単位を返すのであれば、UnicodeString に何文字が含まれているかをどうやって判断すればよいでしょうか。

奇妙なことに、このホワイト ペーパーのための調査をしている間、私はこの点に関する議論をほとんど目にしませんでした。唯一の例外は Jacob Thurman のブログです（http://www.jacobthurman.com/?p=30 で見ることができます）。その他に私は、Delphi 開発チームで働いている Seppy Bloom と Thom Gerdes にも意見を求めました。

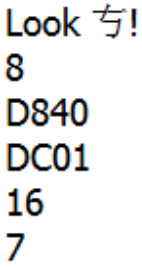
そしてここでも整理すると、上記の質問に対する答えは次のようになります。

1. UnicodeString の各要素はコード単位です。そのため、文字列のサイズ（バイト数）は、長さに要素サイズ（StringElementSize か 2 の好きな方）を掛けたものになります。UnicodeString の長さ（文字数）は、大抵の場合、コード ポイント単位での長さと同じですが、UnicodeString にサロゲート ペアが含まれる場合には同じになりません。
2. MyString[1] にはコード単位が含まれます。これはコード ポイントと同じ場合と同じでない場合があります。
3. 1 つの Char にサロゲート ペアを保持することはできません。Char に保持することができるのは 1 つのコード単位です。
4. UnicodeString の文字数を正確に知りたい場合には、SysUtils ユニットに含まれるヘルパ関数を使用します。たとえば、UnicodeString 中に BMP 文字とサロゲート ペアが混在している場合には ElementToCharLen 関数を使用してください。（つまり、Delphi 2009 より前のバージョンでマルチバイト文字セットを使っていた場合に必要だった方法と似たアプローチです。）

この解答を以下のコードで実証します。

```
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))));
```

結果として ListBox1 に出力される内容は次のようになります。



出力された文字列は 7 文字ですが、UnicodeString には、Length 関数が返すように 8 つのコード単位が含まれます。UnicodeString の 6 番目と 7 番目の要素を調べた結果、上位サロゲート値と下位サロゲート値が表示され、そのそれぞれがコード単位になっています。また、UnicodeString のサイズは 16 バイトですが、ElementToCharLen からは、文字列に合計で 7 つのコード ポイントが含まれることが正確に返されます。

サロゲート ペアについてはこの解答で十分ですが、残念ながら合成文字の場合にはまったく同じではありません。具体的に言うと、UnicodeString に合成文字が少なくとも 1 つ含まれている場合には、実際に文字列中に表示される文字が 1 つだったとしても、その合成文字が 2 つ以上のコード単位を占めている可能性があります。さらに、ElementToCharLen は、合成文字を処理するためではなく、サロゲート ペア専用に設計されたものです。

実際に、合成文字によって文字列の正規化の問題が発生しますが、Delphi の RTL（ランタイム ライブラリ）では現在のところその問題に対処していません。これについて Seppy Bloom に尋ねると、Microsoft が Windows® の新しいバージョンの一部（Windows® Vista、Windows® Server 2008、Windows® 7 など）に正規化 API（アプリケーション プログラミング インターフェイス）を最近追加したと教えてくれました。

Seppy は親切なことに、合成文字を含む UnicodeString の中の文字数を数える方法のサンプル コードまで示してくれました。読者の皆さんのお役に立てるよう、そのコードをここに掲載しますが、いくつか注意点があります。まず、このコードは完全にテストされているわけではなく、保証されていません。ご自身の責任でお使いください。2 つ目に、このコードは Windows XP より前の Windows では動きませんし、Microsoft Internationalized Domain Names（IDN）Mitigation API 1.1 をインストールしている場合には Windows XP でしか動きません。

次がそのコードです。

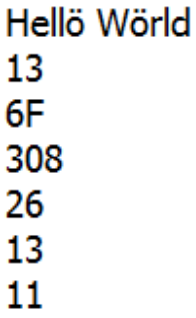
```
const
  NormalizationOther   = 0;
  NormalizationC       = 1;
  NormalizationD       = 2;
  NormalizationKC      = 5;
  NormalizationKD      = 6;

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;
```

次のコードは、2 つの合成文字を含む UnicodeString を使って、NormalizedStringLength 関数の使い方を実証するものです。

```
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)));
```

結果として ListBox1 に出力される内容は次のようになります。



ご覧のとおり、表示される文字列は 11 文字ですが、Length は 13 コード単位を返します（これは正確です）。さらに、UnicodeString の 5 番目と 6 番目の要素には、最初の合成文字の構成要素が含まれています。最後に、ElementToCharLen では 13 文字が存在すると返しているのに対し、NormalizedStringLength では 11 文字を表示していると返します。

これをどう判断するべきでしょうか。ElementToCharLen は不正確なのでしょうか。いいえ、不正確なのではありません。実際に UnicodeString は 13 のコード ポイントを含んでいます。ただ、Unicode の規則で 2 つのコード ポイントを組み合わせて 1 つの合成コード ポイントにすることを 2 度行っていて、その結果が表示される文字数になっただけのことです。（文字列を正規化しています。）

これを先ほどのサロゲート ペアの例と比較してみましょう。1 つのサロゲート ペアには 2 つのコード単位が必要ですが、その 2 つのコード単位が 1 つのコード ポイントを表します。ElementToCharLen が数えるのはコード ポイントです。

このホワイト ペーパーで合成文字を紹介したときに言及した例を思い出してください。どの文字に付いているかに関係なく、分音記号が使われている頻度に研究者が興味を持つかもしれないという可能性を示唆しまし

た。その場合には、分音記号が数える対象として別個の文字となります。いずれにせよ、通常のアプリケーションでは合成文字はめったに見かけないものであり、上記の研究者の例のように特別な場合にだけ使われます。

このセクションを終える前に、Delphi 2009 で初めて登場した Character.pas ユニットについて少し説明しておきたいと思います。このユニットには、TCharacter クラスの他に、UnicodeString 内の個々の文字に関する情報を特定するために利用できる数多くのクラス関数が含まれています。このクラス関数のそれぞれについて、クラス関数を直接に呼び出すスタンドアロンの関数も用意されています。

たとえば、ある文字が大文字か小文字か、記号かどうか、句読記号文字かどうか、制御文字かどうかを特定する関数などがあります。個々の文字を UTF-32 との間で互いに変換する関数もあります。

どうしたわけか、このホワイト ペーパーでは Character.pas ユニットはこの後一度だけ、しかもついでに触れられているだけです。ユニットを覗いてみてください。素敵なものが含まれています。

ANSISTRING に戻す

寄稿者の方々から頂いたフィードバックには、Delphi 2009 以降のバージョンに既存アプリケーションを移行させる方法として一般的なものが 2 つありました。1 つ目は、String、Char、PChar の宣言はそのままにしておいて、新しい Unicode 型では問題が起きる部分（バイト配列を渡さなければならない外部プロシージャに PChar を渡している呼び出しなど）に注意を集中するという方法です。

2 つ目は、String の宣言を AnsiString に、Char の宣言を WideChar に、PChar の宣言を PWideChar に変換するという方法です。この "よく知っているものを使い続ける" 方法では、ANSI 文字と UTF-16 文字の間の不整合を最小化できる傾向があります。

既存アプリケーションを移行する際には Unicode を採用するべきだという強い意見もあります。『Delphi 2009 Handbook』の著者である Marco Cantù は、「ほとんどの場合には、新しい UnicodeString 型を使うようにコードを変換した方が確実に良いでしょう」と書いていて、それによって、Windows API（最近ではほとんどが Unicode を使っています）の呼び出しの多くで速度が向上するなど、多くの効果が得られると言及しています。

しかし、実際には大抵の場合、すべて Unicode にするかすべて AnsiString に戻すかという問題ではなく、2 つのアプローチを混在させる必要があります。Marco は、「ファイルに対するロードや保存、データベースとのデータのやり取り、1 文字 8 ビットの形式にしておかなければならないインターネット プロトコルの使用 ... そのような場合には必ず AnsiString を使うようにコードを変換してください」と述べています。

しかし、この問題には純粋に実用的な面もあります。変換を手早く終わらせる必要があり、他のリファクタリングがほとんど行われていない場合には、シングルバイトの Char 型を使い続ける方が得策かもしれません。逆に、アプリケーションに大がかりな改造を行っていたり、今後長期間にわたって頻繁に保守作業を行う予定

であったり、変更を行う時間を確保する余裕がある場合には、完全に Unicode に変換する方法も説得力を持ちます。

最初に AnsiString を使ったアプローチを見てみましょう。Roger Connell は次のように書いてきました。「[既存コードを Unicode に移行するために] しなければならないことの一覧を [Embarcadero が] 提供していますが、[注意が必要な] コード行が多すぎて負荷を容認できませんでした。私は自分のコードの文字列を AnsiString のまま残すことにし、そのための変換プログラムを作成しました。Unicode サポートは [後で時間があるときに] ... ゆっくりと組み込むつもりです。」

この方法には何の問題もありません。そして、Australian Delphi User Group（ADUG）のメンバでもある Roger は、親切にもその変換ユーティリティを公開してくれました。このユーティリティは、使用するための必要条件の注意書きを添えて、次の場所に置かれています。

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

Delphi 2009 以降への近道を提供してくれた他に、Roger は次のようにも書いています。「[この方法では] コードは D6 でも D7 でも D2009 でもコンパイルできるものになります。UI でパフォーマンスがいくらか悪くなるかもしれませんが、私の考えでは ... [パフォーマンスが悪くなるのは] D7 においてです。」

しかし、ただ AnsiString に戻すことが常に正解であるとは限りません。寄稿者である MVU Technologies LLC の Mariano Vincent de Urquiza は次のように書いてきました。「String を AnsiString に、Char を AnsiChar に大量に置き換えましたが、うまくいきませんでした。プロシージャごとに確認とテストをする必要があり、それが複数のユニットに広がっていて、本当にいらいらしました。終わる気がしませんでした。」

Daintel ApS のソフトウェア開発マネージャである Lars Dybdahl は、コードの種類に応じて文字列の扱い方を変えられるよう、さまざまな方法を提唱しています。たとえば、彼は次のように書いています。「私たちの手元には非常に古いコードがあり、そこではポインタや外部コンポーネントが使われているため、変換が非常に困難でした。しかし、このコードで入出力している文字列データの量は少なかったので、String を RawByteString に、PChar を PAnsiChar に名前変更し、Windows API の ANSI 版を使用するという解決方法が簡単でした。プログラムのこの部分は Unicode をサポートしないことになりますが、状況によってはそれで問題ありません。たとえば、バイナリ データを文字列変数で扱っている暗号化モジュールなどが該当します。」

Lars はさらに提案をしてくれています。「[データ型の] 名前を変更してユニットが正しく動くようになったら、多くの場合、String（UnicodeString）を使うようにインターフェイスを変更した方がよいでしょう。UnicodeString を使っている別のユニットがこのユニットを使ったときに、警告が出ないようにするためです。こうすることで、基本的に、UnicodeString との間の変換を Implementation 部にカプセル化できます。」

匿名希望の別の寄稿者が同様のことを書いています。「私は D2007 の CharInSet [関数] を書いて、必要なところで使用しました。また、あちこちの Char を AnsiChar に変更しました（Windows API [呼び出し]、サードパ

ーティの DLL インターフェイス、ファイル フォーマット定義など）。[その他に] "テキスト" のファイルを取り除きました。D2009 が届いたときに、私はデモ バージョンを試してみましたが、1、2 日でそのすべてがうまく動くようになりました。PChar を使うのは少し怖いと常に思っていたのが役に立ったのかもしれません。」

Roger Connell は、String の宣言を AnsiString に変更すると旧バージョンの Delphi との下位互換性が保たれると言っていますが、実際にはそのような変換は不必要です。開発者の中には、String の宣言を元のまま使ってうまくいった人もいます（この宣言は、Delphi のバージョンにより、AnsiString または UnicodeString としてコンパイルされます）。

Delphi 用の有名なレポート作成ツール ReportBuilder を提供している Digital Metaphors の Nard Moseley は、自分たちの移行について次のように述べています。「ReportBuilder のコードは、ソース コードが 700,000 行を超える大規模で複雑なものです。ReportBuilder を Unicode に移行するにあたって、私たちは Delphi のチーフサイエンティスト Allen Bauer が推奨する戦略を採用しました。要するに、アプリケーションを 1 つの箱であると考えたのです。箱の中ではすべての文字列が Unicode になっています。箱の外側の縁はアプリケーションの外側の縁を表し、アプリケーションはその部分で、異なる文字エンコードを使っているかもしれない他のシステムやファイルなどと通信します。」

彼はまた次のようにも言っています。「その他に私たちが直面した課題は、同じコードで古い ANSI VCL と新しい Unicode VCL の両方をサポートしなければならないという要件です。私たちは、条件付きコンパイルをできるだけ少なくして、ソース コードをできるだけきれいに保つことを常に目標としています。重点を置いた戦略は、TCharacter や TEncoding といった Unicode VCL クラスのファサードを構築することでした。つまり、クラスを直接呼び出す代わりに、内部クラス群を呼び出し、その中で条件付きで TCharacter や TEncoding クラスを呼び出すわけです。

もう 1 人、"String を String のままにしておく" 方法に賛成しているのが David Berneda です（彼の会社 Steema Software では、Delphi に同梱され、Professional Edition としても提供されているチャート作成ツール TeeChart を提供しています）。David は次のように書いています。「何も特別なことはしていません。ただ、すべてが String であることを確認しただけです（どのバージョンの Delphi でもすべてのコンパイルが成功するように）... Unicode 非対応の API を呼び出すときには ShortString を使用し ... PAnsiChar（テキスト）キャストを行います。」

文字列変換

ある型の文字列を別の型の文字列に代入すると、文字列変換が行われます。文字列を元とは異なるデータ型にキャストしたときにも、文字列変換が行われます。文字列変換は、Delphi 2009 以降を使って新規アプリケー

ション開発をするときに必要になることは多くありませんが、移行対象のアプリケーションではよく見られます。

しかし、先に進む前に、Jan Goyvaerts が自分のブログに書いている意見を紹介したいと思います。彼は、$STRINGCHECKS コンパイラ オプションをオン（デフォルト）にすると、Delphi が余分な文字列型検証コードを大量に挿入すると述べています。Delphi は厳密に型指定されているため、このコンパイラ指令をオフにしても安全であり、同時にパフォーマンスが向上すると指摘しています。しかし、C++Builder で開発を行う場合には、このコンパイラ指令をオンにしておくべきです。

文字列変換に関しては、良い点と悪い点があります。良い点は、文字列型は他の文字列型と代入互換性があり、文字型は他の文字型と代入互換性があるということです。代入時に変換が必要になったとしても、文字列から文字列への代入の場合には、自動的に変換が行われます。

悪い点は、変換の中にはデータが失われるものがあるということです。なぜ失われるかを理解することは、Unicode の達人になるためのステップのうちでも困難なものの 1 つです。まずは、このホワイト ペーパーでこれまでに軽く触れたコード ページについて、さらに詳しく検討しましょう。

コード ページ

コード ページという用語は、もともと 7 ビットだった ASCII 文字セット（#$00 - #$7F）を拡張するために使われたメカニズムを指します。最初の MS-DOS では、#$80 から #$FF までの範囲の値は主に罫線文字に使われていました。（MS-DOS で使われていたコード ページは、OEM（original equipment manufacturer）コード ページと呼ばれていました。）

古いバージョンの Windows では、**Windows コード ページ**とも呼ばれるさまざまなコード ページが導入され、Windows で表示しなければならない数多くの言語をサポートしていました。これらのコード ページでは、#$80 から #$FF までの範囲の文字は、ほとんどが言語や文化に固有の文字や記号です。

個々のコード ページはコード ページ識別子で区別します。たとえば、米国のほとんどのコンピュータではコード ページ 1252 を使っています。コード ページ 437 は当初の IBM PC で使われていた OEM コード ページを指します。コード ページ 950 は繁体中国語の ANSI および OEM コード ページを指します。

優に 128 を超える文字が追加で必要な日本語や中国語などの言語に対応するため、Windows はシングルバイトのコード ページとマルチバイトのコード ページの両方をサポートしています。マルチバイトのコード ページでは、1 バイトまたはそれ以上を使って文字を特定します。たとえばダブル バイト文字セット（DBCS）では、**第 1 バイト**（必ず #$7F 以上の値）と**第 2 バイト**を組み合わせて 1 つの文字を特定します。コード ページ 932（日本語）とコード ページ 950（繁体中国語）はダブル バイト文字セットです。

インストールされた Windows ごとにデフォルトのコード ページがあり、それによって、そのマシン上で Unicode 以外の文字に対してデフォルトで使われる文字セットが決まります。また、デフォルトの AnsiString 型の文字セットも決まります（これはどのバージョンの Delphi でも同じです）。

先ほど、Delphi 2009 以降の String のレイアウトでは、2 バイトを使ってコード ページを保持していると説明しました。UnicodeString 型の場合のコード ページは 1200 で、これによって Windows は文字列が UTF-16LE であると判断します。AnsiString の場合、コード ページは、デフォルトの Windows コード ページか、カスタム AnsiString 型用に定義されているコード ページになります。

しかし、気を付けてください。デフォルトで、アプリケーション内のすべての AnsiString 変数のコード ページは、インストールされた Windows のデフォルト コード ページと同じになります。同じアプリケーション内でコード ページの異なる 2 つの AnsiString を作成するには、特定のコード ページを使用すると明示的に Delphi に指定する必要があります（次のセクションのコードにこの例がいくつか含まれています）。

このホワイト ペーパーの寄稿者の数人は、草稿の技術チェックもしてくれました。その 1 人、Lars Dybdahl は、2 段落前の、UnicodeString にコード ページがあるという事実に言及している部分を削除するよう依頼してきました。私はその部分を残すことにしましたが、彼が言ってきた内容をここに引用しておきます。誰かが移行を行うときに彼の考え方が役に立つかもしれないからです。

Lars は次のように書いてきました。「Delphi 2009 を使うときや移行の際に、UTF-16 のコード ページ番号をまったく使ったことがありません。困難だったのは、UnicodeString が常に同じ UTF-16LE エンコードを使っていると把握することでした。それが判明した後、ものごとはずっと簡単になりました。可能な限り UnicodeString だけを使っていれば、すべてが完璧にうまくいくからです。[Unicode 移行を始める前に] 私が読んだドキュメントで UnicodeString のコード ページのことに [触れていなくて]、UnicodeString は AnsiString と違って簡単で速いという事実だけを書いてあれば、私は貴重な時間を無駄にしなくて済んだはずです。」

文字列変換とデータ損失

先ほど述べたように、1 つの文字列型（コード ページ）を別の文字列型に変換するときに、データ損失が起きる可能性があります。データ損失が起きるのは、変換先文字列のコード ページに存在しない文字が変換元文字列に含まれている場合です。

1 つの簡単な例でこれを実証します。次のコードを見てください。

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

```
//Use this line if 1252 is not already your default code page
DefaultSystemCodePage := 1252;
s := #$B4;
a := s;
ListBox1.Items.Add(s);
ListBox1.Items.Add(a);
s := a;
ListBox1.Items.Add(s);
```

このコードを実行した後、ListBox1 には次の値が含まれています。

OEM コード ページ 437 の文字 #$B4 は Unicode 文字 U+2524 を指します。その名前は BOX DRAWINGS LIGHT VERTICAL AND LEFT（細線素片右）です。（#$B4 は AnsiChar リテラルであり、WideChar リテラルと同じではないことに触れておいた方がよいでしょう。#$2524 は WideChar リテラルです。）この文字は、コードが実行されている Windows のデフォルト コード ページ（ここでは 1252）に存在しません。その結果、データが失われ、間違った文字が出力されます。

コードの最後の 2 行は、2 つのコード ページの #$B4 が異なる文字であるというだけの問題ではないことを示しています。この行では、AnsiString の値が IBMAnsi 変数に戻され、その後表示されています。ご覧のとおり、元の文字には戻っていません。

変数 **a** を AnsiString ではなく String（UnicodeString）として宣言すると、データ損失は起きません。具体的には U+2524 の文字が両方の文字列型で表示されます。このことは、**a** を String として宣言した場合に生成される次の出力で実証されます。

RawByteString

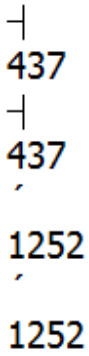
AnsiString の特別な型に RawByteString というものがあります。RawByteString 型はそれ自体のコード ページを持たないため、文字列を RawByteString 型に代入しても暗黙的な変換は行われません。RawByteString に代入された値のコード ページが、代入されたもののコード ページになります。そのため、RawByteString 型は、AnsiString パラメータを渡すための理想的なデータ型になります。他のデータ型を使って AnsiString パラメータを渡すと、2 つのパラメータのコード ページが異なる場合に暗黙的な型変換が行われ、データ損失が起きる可能性が生じます。

このように RawByteString が AnsiString の元のコード ページを "採用" する様子を、次のコードで実証します。IBMAnsi 値を RawByteString に代入したときに、RawByteString がコード ページ 437 を採用していることに注

目してください。同じ RawByteString 変数にデフォルト コード ページを使用している AnsiString の値を代入すると、コード ページ 1252（このコードが動いているコンピュータ上の Windows のデフォルト コード ページ）が採用されます。

```
type
  IBMAnsi = type AnsiString(437); //IBM OEM
var
  i: IBMAnsi;
  a: AnsiString;
  r: RawByteString;
begin
  //Use this line if 1252 is not already your default code page
  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)));
```

このコードを実行した後、ListBox1 は次の図のようになります。



ほとんどの Unicode 移行では、暗黙的な変換の問題を気にする必要はありません。逆に、アプリケーションが暗黙的コード ページ変換の影響を受ける場合には、相当複雑な Unicode 変換に直面している可能性があります。

「事はきわめて複雑です」と Lars Dybdahl は書いています。「プログラマはすぐに混乱してしまうでしょう。どういうメカニズムになっているかを理解できなければ、プログラマは移行プロセスに不満を持ってしまいます。私は、移行に取り掛かる前に、Delphi で実験をして理解しなければなりませんでした。特に、AnsiString（1250）変数に別のコード ページの文字列を格納できるという事実が時々厄介なことになります。含まれるバイト値が文字列のコード ページと一致することが保証されないからです。」

Lars は、自分の論点、特に AnsiString および UnicodeString のリテラルと、RawBytesString、暗黙的変換について実証するためのコード例も提供してくれました。この例は、他のコード例よりも長いけれども非常に興味深いものなので、読者の皆さんが自分で試してみるときの出発点にできるよう、ここに記載しておきます（コメントを少しだけ編集しましたが、それ以外はほとんど彼が送ってくれたままです）。

```
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;
// The expressions on the right side look similar, but they are not
as1 := #$C0;                    // AnsiChar literal that has no code page
as1b := #$C0#$C0;               // UnicodeString literal
as1c := RawByteString(#$C0#$C0);// UnicodeString literal with conversion
                                //at runtime to local code page -
                                //it's not a RawByteString with $C0 values (!)
as2 := #$C0;                    // AnsiChar literal that has no code page

// Both AnsiChar literals got byte value preserved. The UnicodeString didn't.
Assert (ord(as1[1])=$C0);
Assert (ord(as1b[1])<>$C0);
Assert (ord(as1c[1])=$C0);  // as1c now has the 1252 code page
Assert (ord(as2[1])=$C0);

// Now here is a demonstration how things can be seriously confusing.
// Both as1 and as1c are String1250, but as1c now has 1252 as codepage
// because RawByteString() was used to create it. This means that both
// strings only contain $C0 values, but they don't contain the same
// characters.
Assert (length(as1)=1);
Assert (as1[1]+as1c[2] =  as1c[1]+as1c[2]);
Assert (as1   +as1c[2] <> as1c[1]+as1c[2]);

// And because of the different code pages, none of these are the same
character
s1:=as1;
s2:=as2;
Assert (s1<>s2);
end;
```

明示的変換

文字列変換が自動的に行われるのなら、明示的に変換する必要はあるのでしょうか。答えは Yes です。よく見かける変換は、たとえば Windows API 呼び出しにデータを渡すために AnsiString 値を使う必要があるけれども、情報源から受け取ったデータが AnsiString でない場合です。

次の例は、有名な Delphi 専門家の Bob Swart（通称 Dr. Bob）が寄せてくれたものです。彼は次のように書いています。「これは、私が持っている中でも非常にうまく実世界を単純化できた例です。結果として

PChar を返す "昔の" Win32 DLL エクスポート関数を使用しています。この昔の PChar は、現在 PAnsiChar と呼ばれているものです。」

外部 DLL については後で詳しく説明しますので、ここでは明示的変換に着目してください。これは AnsiCharDLL.dll という DLL に含まれる単純なルーチンの静的インポート文です。

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

ご覧のとおり、このルーチンは PAnsiChar を受け取り、PAnsiChar を返します（実際には受け取った PAnsiChar を返します）。

「この関数をインポートし、PAnsiChar を使用すると指定できます。これは問題ありません」と Bob は書いています。「しかし、[入力] 値に TEdit（UnicodeString 値を持つ）を指定して、PAnsiChar 値を受け取る [このような] 関数を呼び出すと、1 度のみならず、2 度の明示的文字列キャストが必要になります。」

これを実証するのが次のコードです。

```
ShowMessage(
  EchoAnsiString(
    PAnsiChar(AnsiString(Edit1.Text)))); // double cast !!!
```

EchoAnsiString に渡そうとしている値がもともと AnsiString であれば、2 番目のキャスト（AnsiString へのキャスト）は必要なく、次の例のようなコードになるでしょう。

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

同様の例に、やはり有名な Delphi 専門家の Marco Cantù のものがあります。98 ページから取った次の変換例では、AnsiString へのキャストを行って GetProcAddress（外部 DLL に含まれるルーチンのエントリ ポイントを動的に取得するための Windows API 呼び出し）を呼び出しています。インポート対象のルーチン名は strFnName に格納されていますが、これは UnicodeString 変数です。

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

Marco は、Delphi の「文字列変換に関する警告（一部はデフォルトでオフになっています）」をすべてオンにすることも提唱しています。次のリストは彼の本の 88 ページから転載したものです。

Explicit string cast（文字列の明示的なキャスト）

Explicit string cast with potential data loss（データ損失の可能性がある文字列の明示的なキャスト）

Implicit string cast（文字列の暗黙的なキャスト）

Implicit string cast with potential data loss（データ損失の可能性がある文字列の暗黙的なキャスト）

Narrowing given wide/Unicode string constant lost information（指定されたワイド/Unicode 文字列定数を縮小変換した結果、情報が失われました）

Narrowing given WideChar constant to AnsiChar lost information（指定された WideChar 定数を

AnsiChar に縮小変換した結果、情報が失われました）
WideChar reduced to byte char in set expression（set 式で WideChar がバイト char に縮小されました）
Widening given AnsiChar constant to WideChar lost information（指定された AnsiChar 定数を WideChar に拡大変換した結果、情報が失われました）
Widening given AnsiString constant lost information（指定された AnsiString 定数を拡大変換した結果、情報が失われました）

同様のことを寄稿者の Lars Dybdahl も勧めています。「文字列に関する警告は、無視するのではなく修正して取り除いてください。そのほとんどは非常に簡単に修正できます。」また、次のようにも提案しています。「あまりに頻繁に UnicodeString から AnsiString への変換が行われることがないよう注意してください。たとえば、AnsiString ユニットの TStringList を使用すると、TStringList との間で代入を行うたびに文字列が変換されます。そうすると、アプリケーションの速度が大きく低下します。」

さて、このセクションの最後に、これも Lars から寄せられた興味深い意見を紹介しましょう。彼は次のように述べています。「文字列の比較の問題は、実際は非常に複雑です。たとえば、次のコードがあるとします。

```
var
  line: String;
const
  myconstant: String='<something with strange unicode chars>';
...
ReadLn (file,line);
if line=myconstant then...
```

「これは Delphi で動くでしょうか。実際のところ、私にも見当がつきません。"if line=myconstant then" の行をコンパイルすると機械語の UStrEqual の呼び出しになることはわかりますが、これがバイナリ比較なのか、それとも、2 つのまったく同じ文字列で異なるバイト値（合成済み文字と分解済み文字）が使われているかもしれないという事実に対処できる正しい Unicode 文字列比較なのかは、まったくわかりません。」

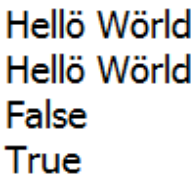
「Windows API は使われていないようなので、私はバイナリ比較なのではないかと推測しています。つまり、上記のコードが必ず正しく動くとは限らないということです。この答えを Google で検索してみましたが、見つかりませんでした。おそらく Delphi の UnicodeString は通常、文字列を正規化したものを格納していますが、TStream.Read を使って UnicodeString 値を読み取った場合にはどうなるでしょうか。その場合、文字列は正規化されないため、正規化された Unicode 文字列とバイト単位で比較することはできません。」

覚えていらっしゃるかと思いますが、前のセクションで String と Char のサイズについて正規化の問題が発生したときに、RTL でまだ正規化を直接サポートしていないことを学びました。ですから、Lars の疑問に対する答えは、2 つの正規化された文字列がまったく同じ値を含んでいる場合でも、line=myconstant という式を評価すると結果が False になる可能性がある、ということだと私は思います。

一方、SysUtils ユニットのいくつかの文字列比較関数では CompareString Windows API を呼び出していて、この関数（実際には ANSI 版と Unicode 版の 2 つの関数があります）は正規化された文字列を基に比較を行っています。次のコードは、この問題と解決策を示すものです。

```
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));
```

ListBox1 の内容を次の図に示します。



CHAR の集合

思い出してください。このホワイト ペーパーの前の方で、変換を簡単に行うことができた匿名の寄稿者が「私は D2007 の CharInSet [関数] を書いて、必要なところで使用しました」と言ったことを取り上げました。ここで暗示されているのは、Char の集合が実際に意味をなさなくなったという事実です。

理由は単純です。Delphi の集合には最大で 256 の要素を含むことができますが、Char は 1 バイトではなく 2 バイトになっています。

実際に Char の集合を宣言しようとすると、次の警告が出力されます。

> W1050 set 式で WideChar がバイト char に縮小されました。'SysUtils' ユニットの 'CharInSet' 関数の使用を検討してください。

選択肢は 2 つあります。宣言を AnsiChar の集合に変更するか、コンパイラが推奨するとおりに CharInSet を使用するかです。実際に、Delphi の多くの文字列関数と同様、CharInSet はオーバーロードされていて、シングルバイト版と WideChar 版が提供されています。SysUtils ユニットでは次のように宣言されています。

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

宣言を見るとわかるように、CharInSet は、渡された文字が渡された集合に含まれているかどうかを示す Boolean 値を返します。この 2 番目のパラメータ TSysCharSet は次のように宣言されています。

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

ポインタ操作とバッファ

おそらく、Unicode 移行で最も重要かつ困難なものの 1 つが、ポインタ操作で文字を使っていたり、文字の配列をバッファとして使っているコードでしょう。このホワイト ペーパーの寄稿者の皆さんから頂いたコメントに何度も書かれていたことからもそれがわかります。（バイト配列として使われている文字列に暗黙的な文字列変換が適用されたら、どれほどの大惨事になるか、想像してみてください。）

一例を挙げると、Delphi Experts（www.DelphiExperts.net）の Olaf Monien は次のように書いています。「PChar [ではポインタ演算ができるため] ... は、コードのすべての行を調べ直さなければならないので、大抵はコストがかかります。」明らかにこの考えが正しいことを示しているのが、移行がきわめて簡単に済んだという匿名の寄稿者の言葉です。「PChar を使うのは少し怖いと常に思っていたのが役に立ったのかもしれません。」

Lars Dybdahl の次の言葉にも複雑なコードの問題が現れています。「私たちの手元には非常に古いコードがあり、そこではポインタや外部コンポーネントが使われているため、変換が非常に困難でした。」この種のコードは 1 行ずつ確認する必要があります。

ポインタやバッファをあまり使わない人であれば（私もその 1 人です）、開発者がそのような構文要素をそもそもなぜ使用するのかを不思議に思うかもしれません。理由は、速度と機能です。

これらの手法を使う人の多くは、速度が最重要視される複雑な演算を行っているか、他に方法がない（なかった）タスクを実行しているか、Delphi（または Turbo Pascal）の初期に開発された手法を使っているかです。（旧式のコーディングに代わる新しい手法やクラスが現れても、開発者によっては、保守性が犠牲になるにも関わらず、惰性で元の手法を使い続けることがあります。これは単なる個人的な所見であり、価値判断ではありません。）

実際に、この種のコードの複雑さは、ポインタやバッファ自体とは関係ありません。問題は Char がポインタとして使われる点です。String と Char のバイト数が変わってしまったので、1 つの Char の長さは 1 バイトであるという、ほとんどのコードで前提とされていた基本事項が無効になりました。

この種のコードを Unicode 変換するのは（概して保守するのも）困難であり、細かい調査が必要なため、可能な限りコードをリファクタリングするべきだという議論も説得力があります。要するに、そのような操作から Char を取り除き、別の適切なデータ型に変更するということです。例を挙げると、Olaf Monien は次のように書いています。「Char（または String）型に対してバイト単位の操作をするのは勧められません。バイトバッファが必要なら、buffer: array[0..255] of Byte; のように 'Byte' をデータ型として使うべきです。」

たとえば、過去に次のようなことをしていたとします.

```
var
  Buffer: array[0..255] of AnsiChar;
begin
```

```
  FillChar(Buffer, Length(Buffer), 0);
```

ただ Unicode に変換したいだけなら、次のような変更になるかもしれません。

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

一方、Olaf が提案するように、Char の配列をバッファとして使うのをやめ、Byte の配列を使用するべきだという意見も妥当です。その場合のコードは次のようになります（最初のコードには似ていますが、2 番目のコードと同じではありません。バッファのサイズが異なります）。

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

より望ましいのは、FillChar の 2 番目の引数を次のようにすることです。これなら配列のデータ型に関係なく正しく動きます。

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

最後の 2 つの例が優れている点は、もともと欲しかったもの、つまりサイズが 1 バイトの値を保持できるバッファが手に入ることです。（さらに、扱う対象がコード単位ではなくバイトになるため、Delphi が何らかの暗黙的文字列変換を行うことはありません。）また、ポインタ演算をしたい場合には、PByte を使うことができます。PByte は Byte のポインタです。

ただし、文字のポインタや文字配列を受け取る外部ライブラリを使用している箇所では、このような変更が不可能な場合があります。そこでは実際に文字のバッファが必要とされているのであり、この文字は通常は AnsiChar 型です。

Byte の配列を使う方法だけでなく、Byte の動的配列である TBytes 型も検討してください。TBytes は SysUtils ユニットで次のように宣言されています。

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

次の話題に移る前に、Embarcadero Technologies の主任研究員である Allen Bauer の言葉をお伝えしたいと思います。彼は自分のブログ（http://blogs.embarcadero.com/abauer/2008/01/24/38852）で次のように書いています。「PChar [ではポインタ演算ができる] ため、... ある型のポインタを PChar にキャストしてポインタ演算を行うといったおかしなことを、（私たち自身も含めて）多くの開発者が行っていました。その結果、ポインタから PChar へのキャストが多用されているコードや、操作対象のデータが 1 バイトのサイズの文字でもないのに PChar ポインタが直接頻繁に使われているコードなどが作成されました。つまり、任意のデータのバイトバッファを操作するのに PChar が使われたということです。」

「[RAD Studio 2009] の開発時に、私たち自身のコードが同じことをしているのに気付きました。（皆が経験があると言ったとおりです。）PChar でごまかす方法は唯一の選択肢でしたし、それによって多くのコードが単純化され、少しは読みやすくなっていました。コードを見ると、バイトの配列としてこのデータ バッファにアクセスしようとしていたこと、配列へのアクセスや単純な演算の実行に便利だから PChar を使っていただけだということが明らかです。」

「[$POINTERMATH コンパイラ指令を] オンにして型付きポインタを宣言すると、その型の変数では、ポインタの先のサイズに応じた代数演算や配列のインデックス処理を好きなだけ行うことができます。PByte の宣言ではこの指令がオンになっています。つまり、バイト ポインタやバイト配列にアクセスするだけの目的で PChar 型を使っているすべての場所で、PChar 型の代わりに PByte 型を使用できるようになったため、既存コードのステートメントやロジックを変更する必要がないということになります。単純にソースを検索して置き換えるだけです。もちろん、PChar の参照を PAnsiChar に変更することもできましたが、それではコードの実際の意図が不明確なままになってしまいます。」

外部データの読み書き

その他、Unicode 移行で注意が必要な分野に、外部ファイルや外部ストリームがあります。Raize Software（受賞に輝く Delphi 開発者向けのコンポーネントやツールを作成している会社）の Ray Konopka がこの問題の真相を明らかにしています。彼が対象としているのはリスト コントロール用の SaveToFile と LoadFromFile ですが、彼のコメントはファイルやストリームの読み書きが行われている多くの状況に当てはまります。

ファイル入出力とテキストのエンコード

「ほとんどのリスト コントロールには、SaveToFile メソッドと LoadFromFile メソッドが準備されています」と Ray は書いています。「これらのメソッドは、通常、RAD Studio 2009 で何の問題もなくコンパイルできます。実行時にも、正しく動いているように見えます。ただし、項目のどれかに Unicode 文字を入れなければの話です。」

「SaveToFile の呼び出しは正しく動くようにすら見えますが、SaveToFile で作成されたファイルはデフォルトでは ANSI でエンコードされているため、Unicode 文字はファイルに正しく格納されません。LoadFromFile を呼び出して、保存されたファイルからリストに情報を取り出そうとしても、実際の Unicode 文字のデータが失われるため、正しく動きません。」

Ray は続けてこう言っています。「解決方法はもちろん、テキスト ファイルで使用するエンコードを指定することです。そのためには、コンポーネント作成者が、SaveToFile と LoadFromFile（さらには SaveToStream と LoadFromStream も）をオーバーロードしてエンコード用のパラメータを受け取るようにしたものを提供しなければなりません。」

「この解決方法に問題はありませんが、コンポーネントを使用する開発者の側で適切なエンコードを選択する必要があります。すべてのファイルで UTF-8 などの Unicode ベースのエンコードを使うことに決めてそれで済ませることも可能です。しかしそれでは、ANSI ベースの文字しか含まないリストも UTF8 ファイルに格納されることになりますし、実際にそこまでする必要はありません。本当に適切なのは、必要な時だけ UTF-8 エンコード [またはその他のエンコード] を使ってファイルを保存する方法です。」

Ray がここで言及しているのは、SaveToFile、LoadFromFile、SaveToStream、LoadFromStream の呼び出し（および類似の呼び出し）はほとんどすべて、任意指定の 2 番目のパラメータとしてエンコードを受け取るようになったということです。特にエンコードを定義しなければ、デフォルトのエンコードが使われます。

エンコードの定義は、SysUtils ユニットにある TEncoding クラスのプロパティまたはクラス関数を使って行います。利用可能な TEncoding クラスには、ASCII、UTF8、Unicode などがあります。

ファイルやストリームのエンコードを管理する必要があるという点は、ある匿名の寄稿者も口にしています。「私たちは構成情報をすべてテキスト ストリームに保存していますが、2009 でコンパイルすると、Unicode [のエンコード] になったことでファイル サイズが 90k から 130k に増加しました」と書いています。「TChart テキスト（TeeChart クラス）がシングルバイト文字を使って保存されていることに気付いたので、マルチバイト文字をシングルバイトに [エンコードし、]」それによって不必要なディスク領域を使わずに済むようになりました。

Ray Konopka はさらに踏み込んで次のように言っています。「テキスト ファイルを常に UTF-8 ファイルとして保存したいとは思いませんでした。そうではなく、Delphi IDE と同じようにファイルを扱いたいと思ったのです。つまり、ユニットに Unicode 文字が含まれている場合にだけ、ファイルを UTF-8 ファイルとして保存するということです。ユニットの内容が ANSI 文字だけであれば、デフォルトのエンコードを使ってファイルを保存します。」

次のコードは、Ray がその方法を示すために寄稿してくれたサンプルです。

```
{$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}
```

Ray はこう説明しています。「リストに Unicode 文字が含まれている場合、Items.Text を AnsiString に変換して、さらにそれを UnicodeString に変換すると、元の Unicode 文字列とは異なる [文字列が返されます]。つま

り、ファイルを UTF-8 でエンコードする必要があるということです。文字列を変換してもデータが失われない場合には、文字列は一致し、ファイルを ANSI として保存できることになります。」

Embarcadero は、TEncoding の参照を受け取ることができる（つまり、使用するエンコードを指定することができる）入出力ルーチンの一覧を公開しています。これは次の（長い）URL に置かれています（途中で改行が入ってしまいました）。

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

Sybase iAnywhere が提供する Advantage Database Server の研究開発プロジェクト マネージャである J.D. Mullin もまた、以前に永続化した情報を復元するために彼が使っている手法を、親切にも教えてくれました。ファイルを ANSI 形式で保存しようとした匿名の寄稿者と同様、J.D. も、以前に保存した ANSI データを必ず正確に復元できるようにしたいと考えました。「ANSI 文字列のデータをストリームから文字列バッファに読み込む方法ではうまくいきません」と彼は書いています。「[データを] まず、一時的な ANSI バッファに明示的に読み込む必要があります。」

次のコードは、その方法を示すために彼が作成したサンプルです。

```
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;
```

J.D. はこれについて次のように説明しています。「一時的な ANSI バッファが必要なのは、読み込み先のバッファの型を ReadBuffer が自動的に判断し、それに合わせて処理を行うためです。ファイルやストリームに ANSI データを格納している場合は、Unicode バッファではなく ANSI バッファに読み込む必要があります。ReadBuffer に Unicode バッファを渡してしまうと、ReadBuffer は Unicode データが保存されていると考えて、それに合わせた読み取り処理を行います。」

Lars Dybdahl もファイルの読み書きのことをよく理解していて、次のように書いています。「私たちの既存の入出力ルーチンの多くは、Delphi 2007 で UTF-8 エンコードを処理するように設計されたものです。そのため、多くのロジックやデータ ストレージは AnsiString で UTF-8 を操作するようになっています。」

「解決策として、アルゴリズム中の UTF-8 変換をすべて取り除き、入出力部分だけに適用して、すべてのテキスト処理を UnicodeString で行うようにしました。たとえば TStringList は、Delphi 2007 では UTF-8 を使ってうまく動きますが、Delphi 2009 では UnicodeString を使用しています。できるだけ早く UTF-8 を UnicodeString に変換する必要があります。必ず TStringList に含める前に行ってください。」

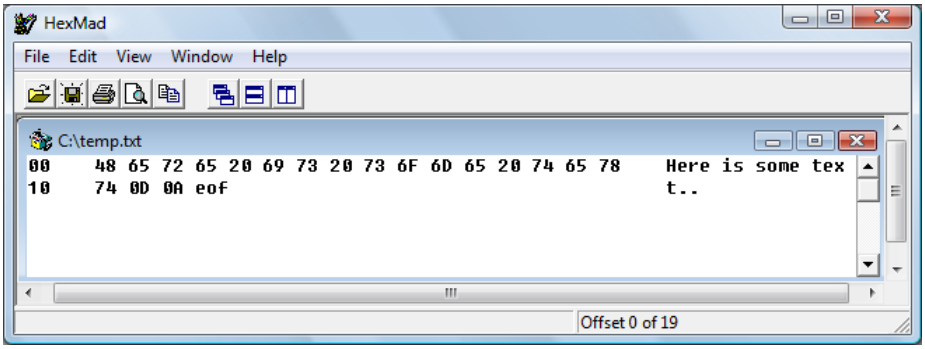
データの読み書きに関するこのセクションを終わらせる前に、あと 2 つの手法について説明しなければなりません。しかしその前に、このホワイト ペーパーでここまでに少しだけ触れた 1 つのトピックについて、もう少し説明が必要です。そのトピックとはバイト オーダー マーク（BOM）です。

バイト オーダー マーク

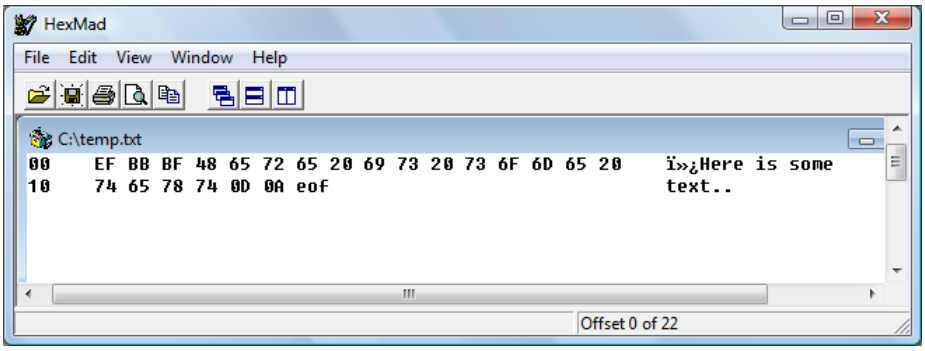
バイト オーダー マークとは、テキスト ファイル中に含まれる可能性があるプリアンブル部であり、含まれる場合にはファイルのエンコードを特定する役割をします。Delphi の SaveToFile メソッド（またはエンコードを指定できる同様のメソッド）を使用すると、必要に応じてファイルの最初の数バイトに BOM が書き込まれます。これは次のサンプル コードで実証することができます。

```
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;
```

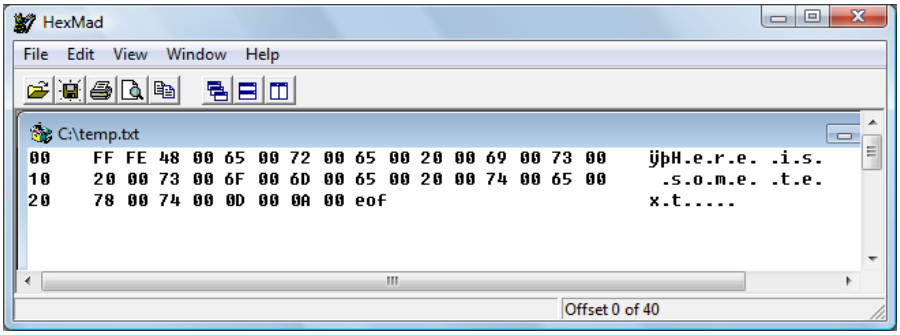
このファイルを低レベルの 16 進ファイル ビューア（ここでは HexMad を使用しています）で ASCII エンコードを使って開くと、次のように表示されます。



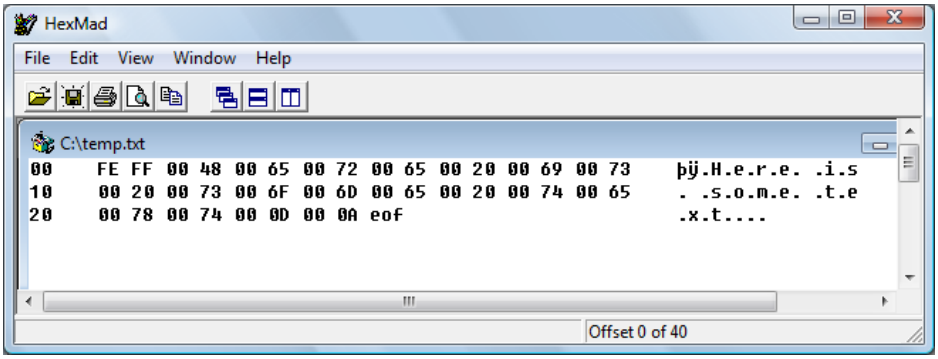
UTF-8 エンコードでは、次のように表示されます。



EF BB BF がバイト オーダー マーク（BOM）で、この例ではファイルが UTF-8 Unicode ファイルであることを特定しています。それに対して、Unicode エンコード（UTF-16）を選択した場合（デフォルトでリトルエンディアンになります）、ファイルは次のようになります。（BOM は TEncoding.GetPreamble メソッドで読み取ることができます。）



そして、UTF-16 BE（ビッグエンディアン）を選択すると次のようになります。



Unicode（UTF-16）のプリアンブル部の長さが 2 バイトしかないことがわかります。リトルエンディアン（デフォルト）のプリアンブル部は FF FE であり、ビッグエンディアンでは逆順の FE FF になります。

BOM は任意指定である

Marco Cantù は、著書『Delphi 2009 Handbook』の 91 ページで次のように書いています。「最も推奨できる方法は、ファイルに保存するときは必ず BOM を保存して、形式を明確に示すことです。これを行うのは、メモリ中で作業している場合にはまったく難しくありません。実際の形式を覚えていなくても、Delphi のストリーム処理のコードが、メモリ ストリームに対してすら適切な BOM を追加してくれるからです。」

この章で見てきたように、エンコードを定義しさえすれば、適切な BOM が書き込まれるようになります。彼は続けて、ファイルやストリームを読み取る場合には、データを書き込んだ時に作成された BOM を基に Delphi が推測するため、エンコードを指定する必要はない、と言っています。また、Delphi では、あるエンコードを使ってデータを書き込んで、そのデータを別のエンコードで読み取っても例外が発生しないため（ただしおそらくデータは正しく読み取れません）、特別なエンコードで作成したファイルを読み取るときには、エンコードを指定しないことが特に重要である、とも言っています。

Unicode データ ファイルでは BOM が必須ではないため、事は少しややこしくなります。これについて Lars Dybdahl は次のように指摘しています。「多くのツールやアプリケーションでは BOM を含むファイルを読み取ることができません。良い例が、アプリケーションで Linux サーバー アプリケーション用の構成ファイルを作成する場合です。BOM を読み取るとサーバー アプリケーションの処理は失敗します。また、BOM が含まれていない XML ファイルは UTF-8 エンコードでなければ読み取れない XML リーダーを使ったことがある人もいるでしょう。同様に、データベースの BLOB フィールドなどのバイナリ構造にテキストを書き込んだ場合、リーダーが BOM を理解しないこともあります。」

そのような理由から、他で作成されたファイルで、Unicode 形式だけれども BOM を含んでいないファイルに遭遇することがあります。問題の原因になりかねないこの点については、寄稿者の J.D. Mullin や Louis Kessler も言及しています。Louis は、Stack Overflow において、「BOM（バイト オーダー マーク）がない場合には、エンコードをどうやって推測するのが一番良いでしょうか」という単刀直入の質問を投稿しました。受け取った回答を基に、彼は CharSet Detector を使うようになりました。これは、Nikolaj Yakowlew が作成したソース モジュールで、ファイル内の文字のパターンを調べてエンコードを予測するものです。私自身は CharSet Detector を試したことがありませんが、こういったものが必要なら、http://chsdet.sourceforge.net/ で詳細を調査することができます。

データの読み書きを取り上げたこのセクションの終わりに、Delphi 2009 以降でストリーム読み書き用の一対のクラスが新しく使えるようになったことだけでも触れておかなければなりません。この TStreamWriter クラスと TStreamReader クラスは、機能的には .NET の対応するクラスと同等です。重要なのは、どちらも、ストリームを読み書きするときに使用したいエンコードを定義できるということです。

外部ライブラリおよびサードパーティ コンポーネントの利用

ここまでの議論で明らかになったように、自分自身のソース コードを移行する際に注意が必要な部分は多種多様です。幸い、その作業に取り組む時には、有利な点がいくつもあります。おそらくはコードを熟知しているでしょうし、コードは自分のスタイルか会社の方針で決められたスタイルで書かれていますし、ソース コードすべてにアクセスすることができます。

しかし、外部ライブラリやサードパーティ コンポーネントに関しては、そういった利点は得られません。ソース コードすらなく、制御の及ばない力に翻弄されることになるかもしれません。

このセクションでは、自分の制御の及ばないコードに関して、3 つの事柄を具体的に取り上げます。まず、Windows API について検討します。それから、サードパーティ コンポーネントについて説明します。最後のセクションでは、ソース コードが実際に手元にある外部ライブラリ、自分や自分のチームで作成した外部ライブラリ、オープン ソース リソースから取得した外部ライブラリなどについて説明します。

WINDOWS API

自分のコードが孤立して動作するのでないことは既定の事実です。必ず "世の中" の他のコードに依存しています。最低でも、オペレーティング システムやそれを動かすためのモジュールが必要です。Windows に関しては、そういったライブラリを Windows API（アプリケーション プログラミング インターフェイス）と呼びます。

Unicode サポートに関しては状況は悪くありません。少なくとも Windows 2000 以降の Windows では完全に Unicode をサポートしていますし、それより前のバージョンでもマルチバイト文字に対応しています。さらに、文字列に関わる Windows API のほとんどは、各ルーチンについて少なくとも 2 つのバージョン、つまり、Windows コード ページ（ANSI）を使う文字列のバージョンとワイド文字列のバージョンが実装されています。これは、windows.pas などのインポート用ユニットを調べてみるとすぐにわかります。そこでは文字列関連ルーチンの A バージョンと W バージョンの両方がインポートされます。

次に示すのは、Windows ユニットに含まれる外部宣言の典型的な例です。

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

ここからわかるのは、実行可能ファイル（コンソール アプリケーション、ウィンドウ アプリケーション、DLL のいずれも）の完全修飾名を返す関数 GetModuleFileName に 3 つのバージョンがあることです。ANSI 文字バッファ（パスを格納するためのもの）を受け取る GetModuleFileNameA（ANSI）版と、UTF-16 文字バ

ッファを受け取る GetModuleFileNameW（ワイド）版、W 版のエイリアスである GetModuleFileName です。上のコードでわかるように、GetModuleFileName 版は実際に W 版を呼び出します。

興味深いことに、Delphi 2009 より前の windows.pas では次のように宣言されています。

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

ここからわかるのは、ほとんどの文字列関連の関数呼び出しについて、しばらく前から ANSI 版とマルチバイト版が利用できるようになっていて、単にデフォルトがワイド版に変わっただけだということです。

一般にこれは、Windows API 呼び出しに ANSI 版とワイド版が揃っていて、自分のコード内でネイティブの String 型、Char 型、PChar 型（Unicode 対応したもの）を使っている場合には、既存コードの移行が問題なく行えることを意味します。次のサンプル コードを見てください。

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

このコードは、Delphi 2009 以降のバージョンでも、それより前の Delphi でも、正しく動き、コンパイルできます。Delphi 2009 以降ではワイド版が呼び出され、それより前のバージョンでは ANSI 版が呼び出されます。

逆に、変換時に何らかの理由で AnsiString に戻す方法（String の宣言を AnsiString に、Char の宣言を AnsiChar に置き換えるなどの方法）を取ることになった場合、Windows API 呼び出しを洗い出して、明示的に ANSI 版を呼び出すように変更しなければなりません。たとえば、次のように変更します。

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

Windows API 呼び出しに適切な ANSI 版が存在しない場合には、別の方法を取らなければなりません。たとえば、互換性のないデータ型を、利用可能なルーチンでサポートされているものに変換またはキャストする方法などが考えられます。しかし正直なところ、そのような調整が必要になる Windows API 呼び出しを私は知りません。

サードパーティ ツール

Windows API が私たちのアプリケーションに不可欠であるのに対して、サードパーティ コンポーネントは、開発時間を短縮したりアプリケーションの機能を向上するために取り入れる便利な道具です。そして、もろ刃の剣でもあります。

アプリケーションの相当量の機能を作成するのにサードパーティ コンポーネントに頼ってしまうと、そのサードパーティ ベンダとの結び付きが始まることになります。自分のアプリケーションを Delphi の新バージョンに移行するときには、そのベンダと連携してアプリケーション移行を行わなければなりません。

幸いなことに、Delphi のサードパーティ コンポーネントに関しては、非常に有能で信頼できるベンダがいます。たとえば、このホワイト ペーパーの寄稿者の 4 人、David Berneda（Steema Software）、Ray Konopka（Raize Software）、J.D. Mullin（Sybase iAnywhere）、および Nard Moseley（Digital Metaphors）は、Delphi 2009 以降をサポートするようそれぞれ自社の製品をアップデートしてくれただけでなく、Delphi のそれ以前のバージョンと互換性を保てる方法で行ってくれました。これらの会社が成功し続け、固定客の気持ちをつかみ続けているのは、こういった理由からです。

一方、保証はまったくありません。何年もの間に、いくつかのサードパーティ ベンダが業界から消えていきました。それらのベンダのコンポーネントをアプリケーションで使用していた場合には、単に不便を被るだけか悲惨な状況になるか、程度は異なるにしても影響を受けたことでしょう。

ソース コードを購入することが可能であり、サードパーティ ベンダが製品をサポートできなくなった場合に開発者が利用し続けられるような形態でそのソース コードが提供されるという条件でなければ、サードパーティ コンポーネント セットを使用しないという開発者を、私は数多く知っています。

ソース コードが手元にあることが常に解決策になるとは限りませんが、非常に良い出発点にはなります。逆に、ソース コードがなければ、選択肢は 2 つしかありません。アプリケーションからコンポーネントを取り除くか、あきらめてそのコンポーネントがサポートしている一番新しい Delphi のバージョンでアプリケーションを保守し続けることにするかです。アプリケーションからコンポーネントを取り除くことは、控え目に言っても骨の折れるものです。旧バージョンのコンパイラに永遠に縛られるのは恐ろしいことであり、通常は、アプリケーション自体のライフサイクルが終わりに近付いているのでなければ受け入れられません。

非営利の外部ライブラリ

ここで取り上げる最後の種類のライブラリは、ソース コードは入手できるけれども営利的にサポートされていないライブラリです。たとえば、自分や自分のチームが作成したライブラリや、オープン ソース ライブラリがそれに当たります。

社内で作成したカスタム ライブラリの移行は、他のアプリケーションとほとんど同じですが、少し工夫が必要です。ライブラリ内部の Unicode サポートを行うだけでなく、API も対応させる必要があります。たとえば、パラメータとして String や Char のデータを渡さなければならないエクスポート関数について、ANSI 版とワイド版の両方を導入する必要があるかもしれません。

オープン ソース ライブラリの場合も問題は同じですが、もう 1 点、検討しなければならないことがあります。ほとんどの場合、ライブラリは GNU パブリック ライセンスかそれと同等の合意の下に公開されています。自

分が所有権を持っているのでない非営利ライブラリの移行に時間を費やす前に、付属の使用許諾契約書をよく読んで、その契約書に示されている条件すべてに従えるかどうかを確認してください。（寄稿者の Steffen Friismose は、ライセンスが GNU パブリック ライセンスまたはそれと同等のものであれば、移行作業をプロジェクトに寄付することで、ライセンスの問題を回避できる可能性が高いと示唆しています。）

データベース関連の問題

Unicode 文字列を格納したり表示する必要がないデータベース アプリケーションは、通常、Unicode 対応の Delphi に簡単に移行できます。Unicode 文字列を処理しなければならないアプリケーションでは、事はもう少し複雑です。このセクションではそれを取り上げます。

しかしその前に、特に Unicode に関係するわけではないものの、Delphi 2009 以降に古いデータベース アプリケーションを移行させる際に遭遇しがちな 1 つの変更について説明します。ブックマークに関する問題です。

ブックマークとは、TDataSet 内のレコードの位置を参照するもので、ナビゲーション用に使われます（ブックマークを使って後で戻ってきたいレコードを特定します）。Delphi 2009 ではブックマークに関する問題が 2 つ発生しています。まず、文字列ベースのブックマーク（TBookmarkStr）が非推奨になり、使用すべきでなくなりました。2 つ目に、TBookmark 型が変更されました。

J.D. Mullin はこれについて次のように説明しています。「[Embarcadero は] TBookmark 型をポインタから TBytes に変更しました。これは、TDataSet の GetBookmark メソッド [、GotoBookmark メソッド]、FreeBookmark メソッドを使っているだけのほとんどのアプリケーションには影響しません。しかし、GetBookmark から返されるポインタを使って "間抜けな" ことをしている場合には注意が必要です。自動テストの多くは、もっと汎用的/標準的なやり方でブックマークを使うように変更しなければなりません。」

TBookmark でどのような "間抜けな" ことをしたくなるのか、私にははっきりとわかりません。しかし、J.D. が言いたいことはよくわかります。ブックマークは、TDataSet 内のレコード位置に印を付け、GotoBookmark を呼び出すことでその位置に素早く戻れるようにするためのものです。他の目的に使っている場合には、コードを注意深くテストする必要があります。

話を Unicode への移行に戻すと、データベース アプリケーションに関しては、残念ながら少し困難です。このホワイト ペーパーの中で、「[Delphi 2009 へ移行するにあたっての] 最大の問題は、WideString を使っているためにアプリケーションが既に Unicode 互換になっていることでした」という寄稿者の言葉を引用したのを覚えていらっしゃるでしょうか。このように複雑になっているのがこの人の例だけでないのは明らかです。

Marco Cantù は、著書『Delphi 2009 Handbook』の中で、Delphi 2009 より前の TDataSet クラスおよび TFields クラスにおける Unicode サポートは TWideString 型によって実現されていると述べています。

Delphi 2009 以降では、マルチバイト文字列型は String として（または明示的に UnicodeString として）宣言されています。

この更新により、TField クラス内の Unicode データの読み書きが UnicodeString データ型と一貫したものになり、潜在的なデータ変換の問題をいくらか取り除くことができましたが、TDataSet に関するクラス名やメンバ名の中には混乱を招くものが残っています。たとえば、TUnicodeStringField 型は存在せず、TStringField クラスはその値を今でも AnsiString 値として格納しています。Unicode の TField が必要な場合には、TWideStringField（これは、前の段落で述べたように、Delphi 2009 以降では UnicodeString として格納されます）を使うことになります。

これらのほとんどは、典型的なデータベース アプリケーションには影響しません。たとえば、Delphi 2009 より前の段階で皆さんのデータベースが Unicode 対応していないなら、TWideStringField などの WideString クラスは必要なかったはずなので、それを UnicodeString に変換する必要もありません。

逆に、Delphi 2009 より前に自分のデータベースで Unicode サポートを実装している場合には、WideString 型の使い方を調査し、Delphi 2009 以降で導入された UnicodeString の定義と矛盾しない形で使われているかを確認する必要があります。

たとえば、Marco は、Delphi 2006 および 2007 の TDataSet.GetFieldNames メソッドは TWideStrings 値を返していたと言っています（『Delphi 2009 Handbook』、333 ページ）。アプリケーションでこのメソッドを呼び出して、その値を TWideStrings 変数に代入している場合、Delphi 2009 以降でコンパイルすると、代入時に WideString から UnicodeString への型変換が行われるようになります。彼は次のように提言しています。「TWideStrings クラスと TWideStringList クラスが使われている箇所をすべて洗い出し、推奨されている TStrings 型および TStringList 型に変更するようコードを書き直してください。」

データベースの移行について寄稿者からのコメントがほとんどないのは興味深いことです。データベース アプリケーションの Unicode 移行はデータベース側と比較するとあまり問題が発生しないことがその理由であるよう、私は望んでいます。実際に、データベースの Unicode 移行について最後に取り上げる 2 つのコメントを寄せてくれたのは Lars Dybdahl です。

その 1 つは提案です。Lars は「データベース フィールドを Unicode に移行する前に、データベース ツールが Unicode に対応しているかを確認してください」と言っています。言い換えると、ツールが Unicode 対応していない間は、データベースに Unicode データを入れるべきではないということです。

Unicode 文字を Unicode 非対応のデータベースに入れられないようにするには（入れるとデータ損失が生じるため）、予防のための計画を少し立てる必要があります。具体的に言うと、データベース対応コントロールを使ってユーザーがデータを編集できるようにしている場合にはどうなるでしょうか。データベース対応コントロールのほとんどでは WideString データをサポートしています。

私が推奨するのは次の方法です。アプリケーションが Delphi 2009 以降で動くようになったら、ユーザー インターフェイスから Unicode 文字をデータベースにわざと追加して、何が起きるかを確認してください。さらに、レポートに表示するなど、そのデータを何かに使った場合にどうなるかを確認してください。

現実を受け入れましょう。ユーザーのデータ入力が妥当かを常に確認することは不可能ですし、ユーザーが Unicode データを入力することで、たとえば間違った日付を入力した場合と同じように問題が生じるかもしれません。ガベージ イン、ガベージ アウト（ゴミを入れればゴミが出てくる）というわけです。

逆に、アプリケーションに Unicode データを導入することで、アクセス違反などの許容できない動作が発生する場合には、データの収集方法を変更して、データベースに実際に挿入する前にデータが有効であることを検証する必要があるかもしれません。たとえば、ユーザー インターフェイスとその下のデータベースとの間に、中間層として ClientDataSet を導入するなどです。ClientDataSet は TWideStringField をサポートしているため、検証が行われるのを待つ間、ユーザーのデータをそこにキャッシュすることができます。

データを収集した後、ClientDataSet 内のデータをデータベースに書き込む前に、文字列フィールドが有効なデータになっているかをテストすることができます。このホワイト ペーパーで先に取り上げた、Ray Konopka が教えてくれたものと同様の裏技が使えるかもしれません。TWideStringField データを AnsiString に変換し、さらに元に変換し直して、変換でデータ損失が発生していないことを検証し、発生していなければデータをデータベースに書き込んで問題ない、という方法です。

2 つ目に、Lars は、Delphi が Unicode をサポートした直接の結果として彼のチームが遭遇した、データ アクセスに固有の移行時の課題を伝えてくれています。Lars は次のように書いています。「IBX で Firebird の Unicode を扱うには、IBX にパッチを適用する必要がありますが、最も困難な部分は BLOB です。BLOB にはバイナリ データが含まれている場合もテキストが含まれている場合もありますが、Delphi 2007 ではどちらであっても問題なく、両方に AsString [メソッド] を使うことができました ...」

「しかし今では、テキスト フィールドを UTF-16 の UnicodeString として、バイナリ データを RawByteString として取得するには、この 2 つをまったく別のものとして扱う必要があります。これを解決するために、BLOB を扱うプロシージャの多くを 2 つずつ（RawByteString 用に 1 つと String 用に 1 つ）作成し、フィールドごとに適切なものを使い分けるようにしました。また、IBX から BLOB を正しく取り出すための小さな関数も導入する必要がありました。」

Lars は彼のチームが使っている Firebird と IBX ドライバについて述べているだけですが、彼の話は Unicode 移行に関する貴重な教えとなっています。前のセクションで見たように、自分のアプリケーションの範囲外のコードに依存している場合には、自分が原因でない問題に遭遇する可能性があります。それでも、移行を成功させるには、そういった不具合に対処しなければならないのです。

まとめ

Nard Moseley は次のように書いています。「Unicode への移行は、最初は怖く思えるものです。未知のものは常に恐怖を伴います。また、Unicode に移行するには、新しい概念を学ぶことが必要です。これは常に苦痛を伴います。しかし実際のところ、ほとんどのソース コードでは、修正が必要ないか、ほんの少しで済みます。データベース アクセスや多くの Windows API 呼び出しなどですら、そのまま動くでしょう。Delphi チームの提案に従い、Delphi コンパイラの警告や検索機能などのツールを利用すれば、Unicode への移行は造作ない簡単な作業で済むかもしれません。」

このホワイト ペーパーの寄稿者から頂いた情報をよく検討すると、Unicode 移行がすべて Nard が言うように簡単に進むとは限らないと警告せざるを得ません。アプリケーションが、どの程度複雑か、何をしているか、何と相互作用しているか、これまでに Unicode 移行がどの程度行われているか（古い WideString 型を使って）などに左右されます。

しかし重要なのは、アプリケーションの変換が簡単であれ困難であれ、終わった時にはアプリケーションの寿命が大幅に延びているということです。Windows 7 機能のサポートなど、アプリケーションのルック アンド フィールが更新されるだけでなく、Delphi で予定されているクロス プラットフォーム コンパイルや 64 ビット ネイティブ コードといった今後の機能強化に対応できるアプリケーションになっているはずです。

謝辞

このホワイト ペーパーは、多くの人の助けによって発想を得、支えられ、実現したものです。実現に力を貸してくださったすべての人に深く感謝します。Embarcadero Technologies では、Unicode 移行についてのホワイト ペーパーを書けばよいと提案してくれた Delphi Solutions シニア ディレクタの Mike Rozlog と、完成に向けて骨身を惜しまずに協力してくれた Del Chiaro に感謝を捧げたいと思います。また、Embarcadero Technologies の Delphi チームのメンバである Seppy Bloom と Thom Gerdes は、執筆時に技術的な支援や助言をしてくれました。ありがとうございました。

また、自分たちの Unicode 対応作業について詳しく教えてくださった数多くの寄稿者の方々にも恩義を感じています。寄稿者は次の方々です（アルファベット順）。David Berneda、Marco Cantù、Rej Cloutier、Roger Connell、Mariano Vincent de Urquiza、Lars Dybdahl、Steffen Friismose、Louis Kessler、Ray Konopka、Olaf Monien、Nard Moseley、J.D. Mullin、Jasper Potjer、Steve、Bob Swart、Jacob Thurman、その他匿名希望の何人かの方々。

また、ホワイト ペーパーの校正や編集を手伝ってくれた Jensen Data Systems の Loy Anderson にも感謝します。大切な方々が最後になりましたが、草稿の技術レビューをしてくださった Lars Dybdahl、Steffen Friismose、Embarcadero Technologies の Takeshi Arisawa に感謝の意を表したいと思います。

参考文献

読者の皆さんの役に立ちそうなさまざまな情報源を以下に挙げておきます。Unicode に関して目を通したブログや論文や投稿をすべて記録しておかなかったことを残念に思っています。しかし幸いなことに、印象的だったものの多くは次にまとめてあります。

UNICODE、DELPHI、ソフトウェアの問題に関するブログや論文

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 in a Unicode World Part I: What is Unicode, Why do you need it, and How do you work with it in Delphi?（Delphi Unicode ワールド パート I：Unicode とは？ なぜ必要なのか？ そして、Delphi でどのような作業を行うのか？）、執筆者：Nick Hodge
http://edn.embarcadero.com/article/38437

Delphi in a Unicode World Part II: New RTL Features and Classes to Support Unicode（Delphi Unicode ワールド パート II：RTL の新機能と、Unicode をサポートするクラス）、執筆者：Nick Hodge
http://edn.embarcadero.com/article/38498

Delphi in a Unicode World Part III: Unicodifying Your Code（Delphi Unicode ワールド パート III：コードを Unicode 対応にする）、執筆者：Nick Hodge
http://edn.embarcadero.com/article/38693

Delphi and Unicode（Delphi と Unicode）**（ホワイト ペーパー）**、執筆者：Marco Cantù
http://www.embarcadero.com/images/dm/technical-papers/delphi-and-unicode-marco-cantu.pdf
http://thedorictemple.blogspot.com
http://www.beholdgenealogy.com/blog

コード ページについて説明している MICROSOFT の WEB ページ

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

ユニコード コンソーシアム

http://unicode.org/

書籍

『Delphi 2009 Handbook』（2009, Marco Cantù）（邦訳『DELPHI 2009 HANDBOOK：Delphi 最新プログラミングエッセンス』）、Amazon.com および Lulu.com より入手可能。

『Delphi 2009 Development Essentials』（2009, Bob Swart）、Lulu.com より入手可能。

執筆者について

Cary Jensen は、ヒューストンに本社を置き、ソフトウェアに関するトレーニング、開発、コンサルティング、指導を行っている会社 Jensen Data Systems, Inc. の社長です。彼は、受賞に輝いたベストセラーの共著者で、Advantage Database Server、Delphi、Kylix、Oracle JDeveloper、JBuilder、Paradox などを題材とした 20 冊以上の書籍を執筆しています。Cary は、北米やヨーロッパのあちこちで行われる会議やワークショップやトレーニング セミナーでも講演をし、人気を集めています。ライス大学で人とコンピュータとの相互作用を専攻し、ヒューマン ファクター心理学の博士号を取得しています。Jensen Data Systems, Inc. の詳細は、http://www.JensenDataSystems.com を参照してください。

コメントと寄稿

今後このホワイト ペーパーを更新していきたいと私は心から望んでいます。このホワイト ペーパーを改善し拡張できるよう、コメントや訂正や寄稿を大いに歓迎いたします。ご意見がありましたら、cjensen@jensendatasystems.com まで電子メールをお寄せください。件名には "Unicode migration" と入れてください。頂いた電子メールには 1 週間以内に返事を出します。返事が届かない場合には、私が電子メールを見過ごしてしまっている可能性があります。お手数ですが、再送をお願いいたします。

エンバカデロ・テクノロジーズについて

エンバカデロ・テクノロジーズは、1993 年にデータベースツールベンダーとして設立され、2008 年にボーランドの開発ツール部門「CodeGear」との合併によって、アプリケーション開発者とデータベース技術者が多様な環境でソフトウェアアプリケーションを設計、構築、実行するためのツールを提供する最大規模の独立系ツールベンダーとなりました。米国企業の総収入ランキング「フォーチュン 100」のうち 90 以上の企業と、世界で 300 万以上のコミュニティが、エンバカデロの Delphi®、C++Builder®、JBuilder®といった CodeGear™製品や ER/Studio®、DBArtisan®、RapidSQL®をはじめとする DatabaseGear™製品を採用し、生産性の向上と革新的なソフトウェア開発を実現しています。エンバカデロ・テクノロジーズは、サンフランシスコに本社を置き、世界各国に支社を展開しています。詳細は、www.embarcadero.com/jp をご覧ください。