



백서

Delphi와 Unicode

작성자: Marco Cantù

2008년 11월

Corporate Headquarters

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

Asia-Pacific Headquarters

L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

DEVGEAR

서울특별시 서초구
반포동 743-14
데브기어 4층

소개: DELPHI 2009 와 유니코드

Delphi 2009의 가장 중요한 새 기능 중 하나는 유니코드 문자셋에 대한 완벽한 지원입니다. 26개의 알파벳 문자를 기반의 영어로만 개발된 Delphi 애플리케이션은 이전 버전에서도 잘 작동하며 Delphi 2009에서도 잘 동작하지만, 전 세계의 언어를 위해 개발된 애플리케이션의 경우 Delphi의 이런 변경으로 뚜렷한 이점을 갖게 됩니다.

서유럽이나 남미 지역에서 개발되어 해당 지역에서만 제대로 작동했던 애플리케이션의 경우 유니코드 지원으로 많은 이점을 얻게 되며, 다른 지역에서 개발된 애플리케이션에도 큰 이점을 제공합니다. 영어로 애플리케이션을 개발하더라도 번역 및 지역화를 하기가 쉽게 되었으며, 이제 아랍어, 중국어, 일본어, 키릴 문자 등의 텍스트가 포함된 데이터베이스 메모 필드를 포함한 모든 언어의 텍스트 데이터를 처리할 수 있습니다.

API 수준에서 유니코드에 대한 광범위한 지원을 제공하는 Windows 운영 체제와 함께, Delphi는 빈틈을 메워 기존의 프로그램의 판매와 새로운 애플리케이션 개발 모두에 새로운 시장을 열어줍니다.

이 백서에서 설명하겠지만 여기에는 배워야 할 새로운 몇가지 개념과 몇몇 유의 사항들이 있습니다. 그러나 그런 변경이 많은 기회를 열어줍니다. 또, 호환성을 높여야 하는 경우 코드의 어떤 부분에서는 기존 문자열 포맷을 계속 사용할 수도 있습니다. 하지만 저는 이런 다양한 주제들을 조급하게 다루기 보다는 기본적인 내용에서부터 설명하려고 합니다. 마지막으로, 유니코드와 Delphi 2009가 제공하는 일부 새 기능을 뒷받침하는 개념을 배우는 데에는 시간이 걸리지만, 모든 어려운 세부 사항에 대해 이해하지 않더라도 Delphi 2009를 사용하여 기존 Delphi 애플리케이션 변환을 시작할 수 있습니다. Delphi 2009에서의 유니코드 사용은 보기보다 훨씬 쉽습니다!

유니코드란 무엇인가?

유니코드(Unicode)는 전세계의 모든 문자 기호들을 포괄하는 국제 문자 셋으로, 여기에는 현재 및 과거의 문자들이 모두 포함되고 약간 더 있습니다. 유니코드에는 문자의 일부가 아니지만 텍스트 작성에 사용되는 기술적인 기호, 문장 기호 및 기타 문자들도 포함됩니다. 유니코드 표준(일반적으로 "ISO/IEC 10646"라고 함)은 유니코드 컨소시엄(Unicode Consortium)에 의해 정의 및 문서화되며 100,000개 이상의 문자들을 포함하고 있습니다. 메인 웹 사이트는 <http://www.unicode.org>입니다.

유니코드의 채용이 Delphi 2009의 중심적인 요소이므로 설명해야 할 사항들이 많습니다.

유니코드를 간단하게 이해할 수 있는 기본 배경 아이디어는, 모든 단일 문자가 고유한 번호(정확한 유니코드 용어로는 코드 포인트)를 가진다는 것입니다. 여기서는 유니코드의 전체 이론을 깊이 있게 다루지 않고 핵심 사항만 강조하도록 하겠습니다.

Unicode TRANSFORMATION FORMAT

유니코드를 이해하기 어렵게 하는 것은 실제 저장소 또는 물리적 바이트라는 측면에서 동일한 코드 포인트(유니코드 문자의 숫자 값)를 표시할 수 있는 방법이 여러 가지라는 것입니다. 모든 유니코드 코드 포인트를 간단히 통일되게 표현할 수 있는 유일한 방법이 각 코드 포인트에 대해 4 바이트를 사용하는 방법뿐이라면, 대부분의 개발자는 유니코드가 메모리 및 처리 측면에서 너무 많은 대가를 요구한다고 생각할 것입니다. (Delphi에서는 유니코드 코드 포인트를 UCS4Char 데이터 타입으로 나타낼 수 있습니다)

잘 알려진 “UTF”라는 용어가 Unicode Transformation Format의 약어라는 것을 아는 사람은 별로 없습니다. 이는 유니코드 표준의 일부인 알고리즘 매핑으로서 각 코드 포인트(문자의 절대적인 숫자 표현)를 지정된 문자를 표현하는 바이트의 고유 시퀀스에 매핑합니다. 매핑은 양 방향으로 사용되어, 다른 표현으로 서로 전환할 수 있습니다.

표준은 세트의 처음 부분(처음 128개의 문자)을 표현하는 데 몇 바이트를 사용하느냐에 따라 이러한 3개의 인코딩 또는 형식(8, 16 또는 32)을 정의합니다. 인코딩의 3가지 형식 모두가 각 코드 포인트에 대해 최대 4바이트의 데이터를 필요로 한다는 것은 흥미로운 사실입니다.

- UTF-8은 문자를 1 ~ 4바이트의 변수-길이 인코딩으로 변환합니다. UTF-8은 HTML 등의 프로토콜에서 자주 사용되는데, 그것은 대부분의 문자가 ASCII 집합에 포함되는 경우(HTML의 마커처럼) 아주 컴팩트하기 때문입니다.
- UTF-16은 대부분의 운영 체제(Windows 포함)나 Java, .NET과 같은 환경에서 자주 사용됩니다. 대부분의 문자가 2바이트에 맞추어지므로 사용이 편리하며 간결하고 처리 속도가 빠릅니다.
- UTF-32는 처리 작업에 가장 유리하지만(모든 코드 포인트가 동일한 길이를 가지므로), 메모리가 많이 사용되며 실제 사용은 그리 많지 않습니다.

멀티바이트 표현(UTF-16 및 UTF-32)과 관련된 또 다른 문제는 바이트 중 어떤 부분이 먼저 오느냐 하는 것입니다. 표준에서는 모든 형태가 허용되므로 UTF-16 BE(big-endian) 또는 LE(little-endian)를 사용할 수 있으며 UTF-32에도 마찬가지입니다.

BOM

유니코드 문자를 저장하는 파일은 종종 바이트 순서 형태(BE 혹은 LE)와 사용 중인 유니코드 형식을 표시하는 표시(signature)로서 BOM(Byte Order Mark)이라고 하는 초기 헤더를 사용합니다. 다음 표는 다양한 BOM에 대한 요약으로서, 2, 3 또는 4바이트 등 다양한 길이를 갖습니다.

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

WIN32의 유니코드

Win32 API에서는 Windows NT부터 유니코드 문자에 대한 지원이 포함되었습니다. 대부분의 Windows API 함수들은 ASCII 버전(A 문자로 표시)과 와이드 문자열 버전(W 문자로 표시)의 2개 버전을 가지고 있습니다.

다음은 그 예로서 Delphi 2009의 Windows.pas의 코드 일부입니다.

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

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

거의 비슷한 선언이지만, 문자열을 나타낼 때 **PAnsiChar** 또는 **PWideChar**로 달라집니다. 문자열 포맷 표시가 없는 일반 버전은 내부적으로 이 두가지 중의 하나를 의미하게 됩니다. Delphi의 이전 버전에서는 계속 'A' 버전을 사용했지만 위에서 볼 수 있듯이 Delphi 2009에서는 'W' 버전이 기본값입니다.

CHAR는 이제 WIDECHAR

상당히 오랫동안 Delphi에는 문자를 나타내기 위해 다음의 두 가지 개별적인 문자 타입들을 가지고 있었습니다.

AnsiChar: 8비트 표현을 사용 (256개의 기호들을 표시), 코드 페이지에 따라 다르게 해석됨

WideChar: 16비트 표현을 사용 (64K의 기호 표시)

이 측면에서 Delphi 2009에 변화된 점은 없습니다. 달라진 점은, **AnsiChar**의 별칭이었던 **Char** 타입이 이제 **WideChar**의 별칭이 되었다는 것입니다. 컴파일러가 코드에서 **Char**를 발견할 때마다 **WideChar**로 해석합니다. 이런 새 컴파일러 디폴트를 변경할 수 있는 방법은 없습니다. (문자열 타입에 대해, **Char** 타입은 고정된 하드 코딩 방법으로 특정한 데이터 타입에 매핑됩니다. 개발자들dms 컴파일러 지시어(directive)로 전환할 수 있기를 요청했지만, 이는 QA, 지원, 패키지 호환성 등의 측면에서 심각한 문제를 일으키게 됩니다. 물론 코드를 수정하여 **AnsiChar**와 같은 특정 타입을 사용할 수는 있습니다.)

이는 많은 소스 코드에 영향을 미치고 많은 결과를 양산하는 상당히 큰 변화입니다. 예를 들어 **PChar** 포인터는 이제 이전의 **PAnsiChar**가 아닌 **PWideChar**의 별칭입니다.

서수 타입으로서의 CHAR

이 새 Char 타입도 역시 서수 타입이므로, **Char** 타입의 카운터를 가진 **for** 루프에서 **Inc** 및 **Dec**를 사용할 수 있습니다.

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

약간의 문제가 발생할 수 있는 유일한 경우는 전체 Char 타입의 셋을 선언하는 때입니다.

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

이 경우, 컴파일러는 기존 코드를 Delphi 2009로 포팅하는 것이라고 추정하게 되어, Char를 AnsiChar로 간주하고(셋은 최대 256개의 요소만을 가질 수 있으므로) 경고 메시지를 보여줍니다.

```
w1050 wideChar reduced to byte char in set expressions. Consider using
'CharInSet' function in 'SysUtils' unit.
```

이 코드는 예상한 대로 동작할 것이지만, 모든 문자 셋을 가질 수 없으므로 기존의 일부 코드가 간단히 매핑되지 않습니다. 모든 기존 코드를 간단히 매핑되게 하려면 경고에서 권고하는 내용에 따라 알고리즘을 변경해야 합니다.

이 방법 대신에 단순히 경고가 나타나지 않도록 하기 원한다면(위의 코드를 컴파일하면 경고 2개가 발생됨) 다음과 같이 코딩할 수 있습니다.

```
var
  CharSet: set of AnsiChar; // 경고가 나타나지 않도록 함
begin
  CharSet := ['a', 'b', 'c'];
  if AnsiChar('a') in CharSet then // 경고가 나타나지 않도록 함
  ...
```

CHR를 사용한 변환

숫자 값을 AnsiChar 또는 WideChar로 타입 캐스트를 하면 문자로 변환할 수 있으며, 전통적인 Pascal 방식인 Chr 함수(보통 Ord의 반대로 인식됨)로 숫자 값을 문자로 변환할 수도 있습니다. 이 표준 함수는 매개 변수로 바이트가 아닌 워드를 받도록 확장되었습니다.

그렇지만 문자 리터럴과 달리 Chr에 대한 호출은 항상 유니코드 영역에서 해석됩니다. 따라서 다음과 같은 코드를 Delphi 2009로 포팅할 경우, 결과에 놀랄 수도 있습니다.

```
chr(128)
```

이 코드 대신 #128을 사용하면 다른 결과를 얻을 수 있을 것이며 코드 페이지에 따라 다를 수 있습니다.

32 비트 문자

기본 Char 타입이 이제는 WideChar로 매핑되는 것과 별도로, Delphi가 4바이트 문자 타입으로 UCS4Char도 정의하고 있다는 사실을 알아둘 필요는 있습니다. (System 유닛)

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

이 타입의 정의와 **UCS4String**(UCS4Char의 배열로 정의됨)에 대한 정의가 이미 Delphi 2007에 이미 있긴 하지만 Delphi 2009의 **UCS4Char** 데이터 문자 타입은 새 **Character** 유닛(다음 섹션에서 설명)의 RTL 루틴을 포함하여 몇몇 RTL 루틴에서 자주 사용되므로 매우 중요합니다.

새 CHARACTER 유닛

새 유니코드 문자와 유니코드 문자열을 보다 잘 지원하기 위해 Delphi 2009는 RTL에 완전히 새로운 유닛인 **Character** 유닛을 추가했습니다. 이 유닛은 **TCharacter** 실드(sealed) 클래스를 정의하는데, 이 클래스는 기본적으로 정적 클래스 함수의 모음이며, 추가적으로 이 정적 클래스의 **public** 및 일부 **private** 함수에 매핑된 글로벌 루틴들이 포함되어 있습니다.

이 유닛에는 또한 두 개의 재미있는 열거 타입들이 정의되어 있습니다. 첫 번째 열거 타입은 **TUnicodeCategory**로서, 컨트롤, 공백, 대문자 또는 소문자, 소수, 문장 기호, 수학 기호 등과 같은 다양한 범주로 다양한 문자를 매핑합니다. 두 번째 열거 타입은 **TUnicodeBreak**라고 하며 다양한 공백, 하이픈 및 줄 바꿈 등의 계열을 정의합니다.

TCharacter 클래스에는 40개 이상의 메소드가 있으며, 단독 문자나 문자열 내의 문자를 대상으로 동작합니다.

- 문자의 숫자 표현을 리턴합니다. (**GetNumericValue**)
- 문자의 종류를 알아내고(**GetUnicodeCategory**) 특정 종류에 속하는 문자인지 확인합니다 (**IsLetterOrDigit**, **IsLetter**, **IsDigit**, **IsNumber**, **IsControl**, **IsWhiteSpace**, **IsPunctuation**, **IsSymbol** 및 **IsSeparator**).
- 문자가 대문자인지 소문자인지 확인하고(**IsLower** 및 **IsUpper**) 대문자 또는 소문자로 변환합니다(**ToLower** 및 **ToUpper**).
- 문자가 UTF-16 서로게이트 쌍의 일부인지 확인합니다(**IsSurrogatePair**, **IsSurrogate**, **IsLowSurrogate**, **IsHighSurrogate**).
- UTF32로 변환하거나 UTF32로부터 변환합니다(**ConvertFromUtf32**, **ConvertToUtf32**).

Character 유닛의 글로벌 함수는 이들 정적 클래스 메소드와 거의 일치하며, 그 중 일부는 이름은 다르지만 기존의 Delphi RTL 함수에 해당합니다. 문자에 대해 동작하는 일부 기본 RTL 함수는 적절한 유니코드 지원 코드를 호출하는 확장 버전들이 있습니다. 예를 들어 악센트 문자를 대문자로 변환하기 위해 다음과 같은 코드를 작성할 수 있습니다.

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

```

Memo1.Lines.Add('ToUpper ù: ' + ToUpper(ch1));

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

```

일반적인 Delphi 코드(AnsiChar 버전의 **UpCase**)는 ASCII 문자만 처리하기 때문에 문자를 변환하지 않습니다. 이것은 ASCII 만 처리하는 **UpperCase** 함수의 경우에도 마찬가지입니다. 반면 **AnsiUpperCase** 함수는 이름과는 달리 유니코드로 된 모든 문자를 처리합니다. **WideChar** 를 전달하는 경우에도 마찬가지입니다(하위 호환성 목적). **ToUpper** 함수도 제대로 동작합니다 (이제는 Windows API 의 **CharUpper** 함수를 호출하지 않습니다). 다음은 위의 코드를 실행한 결과입니다.

```

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

```

Char에 **UpCase** 호출을 사용함으로써 기존 Delphi 코드를 유지할 수 있으며, 이 코드는 일반적인 Delphi 방식을 유지하고 있다는 것에 주목하십시오.

다음 코드는 Character 유닛에서 도입된 특정 유니코드 관련 기능들에 대한 좋은 데모로서, 음악 기호 *G clef*인 유니코드 코드 포인트 \$1D11E를 포함한 문자열을 정의합니다.

```

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

```

위 코드에 이어 다음의 코드로 위 문자열의 문자들에 대해 다음과 같이 테스트를 할 수 있습니다. (모두 True를 리턴합니다)

```

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

```

마지막으로, SysUtils의 **IsLeadChar** 함수는 유니코드 서로게이트를 처리할 수 있도록 수정되었으며, 또한 문자열의 다음 문자로 이동하는 데 사용되는 기타 관련 함수 등을 처리할 수도 있습니다.

문자열과 UNICODESTRING

Char 타입 정의에서의 변경 사항은 문자열 타입 정의에서의 변경 사항과 밀접한 관련이 있기 때문에 매우 중요합니다. 문자와는 다르게 문자열은 이전에는 없었던 새로운 데이터 타입(**UnicodeString**이라고 함)에 매핑됩니다. 내부 구조도 **AnsiString** 타입의 표현과는 상당히 다릅니다. (여기서 *classic AnsiString* 타입이라는 특정 용어는 Delphi 2에서부터 Delphi 2007까지 작동하는 문자열 타입을 언급하기 위해 사용하며, **AnsiString** 타입은 Delphi 2009에서도

사용되지만 동작이 수정되었기 때문에 이전 구조를 언급할 경우 *classic AnsiString*이라는 용어를 사용합니다.)

WideChar 타입에 기반한 문자열을 나타내는 **WideString** 타입이 언어에 이미 있는데, 왜 힘들게 새 데이터 타입을 정의해야 할까요? **WideString**은 참조가 카운트되지 않으며 성능과 유연성의 측면에서 매우 취약합니다. 예를 들어 **WideString**은 네이티브 **FastMM4**가 아닌 **Windows** 글로벌 메모리 할당자(**allocator**)를 사용합니다.

AnsiString과 마찬가지로 **UnicodeString**은 참조가 카운트되며 **copy-on-write** 의미를 사용하고 성능이 우수합니다. **AnsiString**과 달리 **UnicodeString**은 문자별로 2바이트를 사용하며 **UTF-16**을 기반으로 합니다. 사실상 **UTF-16**은 변수 길이 인코딩이며, 때때로 **UnicodeString**은 두 개의 **WideChar** 서로게이트 요소(즉, 4바이트)를 사용하여 단일 유니코드 코드 포인트를 나타냅니다.

문자열 타입은 이제 **Char** 타입에서와 마찬가지로 하드 코딩된 방식으로 **UnicodeString**에 매핑됩니다. 이를 변경할 수 있는 컴파일러 지시어나 다른 방법은 없습니다. 계속해서 **AnsiString** 문자열 타입을 사용해야 하는 코드가 있으면 해당 코드를 **AnsiString** 타입의 명확한 선언으로 교체해야 합니다.

문자열의 내부 구조

새 **UnicodeString** 타입과 관련된 주요 변경 사항 중 하나는 내부 구조입니다. 새로운 구조는 참조 카운트되는 문자열 타입 모두(**UnicodeString** 및 **AnsiString**)에 해당되며, 참조 카운트되지 않는 문자열 타입들(**ShortString** 및 **WideString**)에는 해당되지 않습니다.

기존의 **AnsiString** 타입의 구조는 다음과 같았습니다.

-8	-4	문자열 참조 주소
참조 카운트	길이	문자열의 첫 번째 문자

첫 번째 요소(문자열 자체의 시작 부분부터 역으로 카운트)는 **Pascal** 문자열 길이며 두 번째 요소는 참조 카운트입니다.

Delphi 2009에서는, 참조 카운트된 문자열의 구조는 다음과 같습니다.

-12	-10	-8	-4	문자열 참조 주소
코드 페이지	요소 크기	참조 카운트	길이	문자열의 첫 번째 문자

길이와 참조 카운트 외에, 새로운 필드는 요소 크기와 코드 페이지를 나타냅니다. 요소 크기는 **AnsiString**과 **UnicodeString** 사이에 식별하는 데 사용되며, 코드 페이지의 경우는 **UnicodeString** 타입이 고정 코드 페이지 1200을 가지므로 **AnsiString** 타입에서만 의미를 갖습니다.

해당 지원 데이터 구조는 다음과 같은 **System** 유닛의 임플멘테이션 섹션에서 선언되어 있습니다.

```

type
  PStrRec = ^StrRec;
  StrRec = packed record
    codePage: word;
    elemSize: word;
    refCnt: Longint;
  
```

```
length: Longint;
end;
```

이 지원 데이터 구조는 임플멘테이션 섹션에 있으므로 여러분의 코드에 사용할 수는 없으며, 이는 내부 데이터 구조가 구현 관련 문제이고 변경될 수 있다는 면을 고려할 때 타당하다고 볼 수 있습니다. 일반적으로 사용할 필요가 있는 정보는 그에 액세스할 수 있는 헬퍼 함수들이 있습니다.

새 필드가 기존 필드보다 더 간결하긴 하지만 8 ~ 12 바이트의 문자열 오버헤드가 발생하게 되므로 그러한 간결한 표현이 호환성이라는 대가를 치르고 얻어야 할 만큼 더 효과적인지에 대해서는 의문을 가질 수도 있을 것입니다. 이는 메모리와 속도 간에 일반적으로 발생하는 트레이드 오프(trade-off) 차원의 문제인데, 즉 단일 위치의 일부를 사용하지 않고 각기 다른 메모리 위치에 데이터를 저장하면 런타임에 더 빠른 속도를 얻을 수 있지만, 대신 생성하는 각각의 문자열에 대해 추가로 메모리를 소요하게 됩니다.

과거에는 저수준의 포인터 기반 코드를 사용하여 참조 카운트에 액세스해야 했지만 Delphi 2009 RTL에서는 몇가지 편리한 함수를 추가하여 문자열의 여러 메타데이터에 액세스할 수 있습니다.

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

SysUtils 유닛에는 **ByteLength** 라는 새로운 헬퍼 함수도 있습니다. 이 함수는 **StringElementSize** 속성을 무시하고 **UnicodeString** 의 크기(바이트 단위)를 리턴합니다. 따라서 좀 이상하게 느껴질 수도 있겠지만 **UnicodeString** 외의 문자열 타입에는 동작하지 않습니다.

예를 들어 다음과 같이 문자열을 생성하고 해당 문자열에 대한 정보를 요청할 수 있습니다.

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

이 프로그램은 다음과 같은 결과를 보여줄 것입니다.

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: -1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

UnicodeString 에 의해 리턴되는 코드 페이지는 1200 이며, 이것은 글로벌 변수인 **DefaultUnicodeCodePage** 에 저장되어 있는 숫자 값을 리턴한 것입니다. 위의 코드와 결과에서 **Length** 는 문자 수를 리턴하므로 문자열 길이를 바이트 단위로 확인할 수 있는 직접적인 호출은 없습니다.

물론 다음 표현식을 사용하여, 길이 값에 각 문자의 바이트 크기를 곱하여 문자열 길이의 바이트 값을 알 수 있습니다.

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

문자열의 정보를 알아낼 수 있을 뿐만 아니라 그 정보 일부는 변경할 수도 있습니다. 문자열을 변환할 수 있는 저수준의 방법은 **SetCodePage** 프로시저를 호출(뒤에서 살펴볼 **RawByteString** 타입에만 해당)하는 것입니다. 이 프로시저는 코드 페이지를 실제 코드페이지에 맞게 조정하거나 전체 문자열 변환을 수행할 수 있습니다. 이 프로시저는 “문자열 변환” 섹션에서 사용해볼 것입니다.

UNICODESTRING과 유니코드

새 문자열 타입, 더 정확히 말해서 새로운 **UnicodeString** 타입은 당연히 유니코드 문자셋에 매핑됩니다. 그러나 “어떤 유니코드가 사용되는가”의 문제가 남습니다.

새 문자열 타입이 **UTF-16**을 사용한다는 것은 그리 놀랄만한 일이 아닙니다. 더 정확히 말해 **UnicodeString** 타입은 리틀 엔디안(little-endian) 표현을 사용하는 **UTF-16** 문자열, 즉 **UTF-16 LE**로 메모리에 저장됩니다. 여기에는 여러 가지 이유가 있는데, 가장 중요한 것은 새로운 문자열 타입이 최신 버전의 윈도우 운영 체제에서 **Windows API**가 관리하는 네이티브 문자열 타입이라는 것입니다.

Delphi 2009의 **WideChar** 타입을 다뤘던 섹션에서 설명했듯이 새 **TCharacter** 지원 클래스 (**WideChar**뿐만 아니라 **UnicodeString** 처리를 위해서도 사용됨)는 **UTF-16** 및 서로게이트 쌍을 완벽히 지원합니다. 그 섹션에서 언급하지 않았던 것은, 단일 유니코드 코드 포인트를 서로게이트 쌍(즉, 두 개의 **WideChar**)으로 나타낼 수 있기 때문에 문자열의 **WideChar** 개수가 문자열에 포함된 유니코드 코드 포인트 수와 달라지는 결과가 발생한다는 것입니다.

서로게이트 쌍을 가지는 문자열을 생성하는 방법은 다음과 같이 적절한 상황에서 서로게이트 쌍(두 개의 **WideChar**)을 포함한 문자열을 리턴하는 **ConvertFromUtf32** 함수를 사용하는 것입니다.

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

이제 문자열 길이를 요청하면 **WideChar**의 개수인 8이 리턴되지만 문자열에 있는 논리적 유니코드 코드 포인트의 개수는 리턴되지 않습니다. 문자열을 출력해보면 적절한 결과를 얻게 됩니다(적어도 **Windows**는 서로게이트 쌍의 자리표시자로 두 개가 아닌 하나의 사각형을 표시해줄 것입니다).

또한 **ConvertFromUtf32**의 코드(더 정확히 말하면 이 함수가 호출하는 **TCharacter** 클래스의 **ConvertFromUtf32** 클래스 메소드)를 살펴보면, 유니코드 코드 포인트를 서로게이트 쌍에 매핑하는 데 사용되는 실제 알고리즘을 볼 수 있습니다. 세부 동작에 관심이 있다면 이 코드를

살펴보면 흥미로울 것입니다.

관련된 다른 문제 하나는, 문자열의 각 문자에서 루핑을 하면 무슨 일이 일어나느냐는 것입니다. **for** 루프 또는 **for-in** 루프의 표준에 따르면 각 논리적 유니코드 코드 포인트가 아닌 문자열의 각 **WideChar** 요소에 대해 작업하게 됩니다. 따라서 **NextCharIndex** 함수를 기반으로 하여 **while** 루프를 사용하거나 혹은 다음과 같이 서로게이트를 확인하는 **for** 루프를 적용해야만 합니다.

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

하지만, 대부분의 경우 **BMP(Basic Multilingual Plane)**에서 작업할 수 있으므로 유니코드 문자열의 각 **WideChar**를 단일 코드 포인트로 처리할 수 있습니다.

UCS4STRING 타입

일련의 유니코드 코드 포인트들을 처리하는 데 사용할 수 있는 또 다른 문자열 타입으로 **UCS4String** 타입이 있습니다. 이 데이터 타입은 4바이트 문자(**UCS4Char** 타입)의 동적 배열을 나타냅니다. 따라서 참조 카운트 또는 **copy-on-write** 지원이 없으며 **RTL**도 거의 지원되지 않습니다.

이 데이터 타입(**Delphi 2007**에서 도입됨)은 특정 상황에서 사용될 수 있지만, 일반적인 상황에는 적당하지 않습니다. 문자열이 문자당 4바이트를 사용할 뿐만 아니라 메모리에 여러 사본을 복사해야 하기 때문에 메모리가 낭비될 수 있습니다.

여러 문자열 타입

새 **UnicodeString** 타입의 도입과 함께, 모든 문자열 타입(**AnsiString** 타입도 포함)에 의해 공유되는 변경된 내부 구조는 문자열 관리에 있어 개선의 여지를 제공합니다. **Delphi R&D** 팀은 여러 데이터 타입과 새 문자열 타입 정의 메커니즘을 제공하기 위해 이 새로운 내부 구조를 활용하여 문자열 관리를 향상시키기 위한 컴파일러 수준의 모든 작업을 수행했습니다.

UnicodeString 외에 이미 정의되어 있는 문자열 타입으로는 다음과 같은 것이 있습니다.

AnsiString은 문자당 1바이트 문자열 타입으로, 운영 체제의 현재 코드 페이지를 기반으로 하며, **Delphi** 이전 버전의 기존 **AnsiString**과 거의 일치합니다.

UTF8String은 문자의 길이가 가변인 **UTF8** 형식의 문자열입니다.

RawByteString은 코드 페이지 정보가 없는 문자 배열로서, 시스템에 의해 문자 변환이 수행되지 않습니다. 따라서 **RawByteString**은 순수하게 문자 배열로 사용할 경우에는 기존의 **AnsiString**과 비슷한 부분이 있습니다.

이런 새 문자열 타입의 정의를 살펴보면 타입 정의 메커니즘의 감을 잡을 수 있을 것입니다.

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

다음 섹션에서는 `AnsiString`과 사용자 정의 문자열 타입을 설명한 다음 `UTF8String` 타입을 설명하도록 하겠습니다. 일반적으로 변환을 피하기 위해 `RawByteString`을 사용하므로, 문자열 변환을 설명하는 다음 섹션에서 이 문자열 타입에 중점을 두고 설명하겠습니다.

새 ANSISTRING 타입

기존의 Delphi 버전들에서와 달리, 새 `AnsiType` 문자열은 추가 정보, 즉 문자열에 있는 문자들에 대한 코드 페이지를 제공합니다. `DefaultSystemCodePage` 변수는 기본값으로 현재 윈도우 코드 페이지인 `CP_ACP`로 되어 있지만, 특수 프로시저인 `SetMultiByteConversionCodePage`를 호출하여 수정할 수 있습니다. 전체 프로그램을 강제로 지정된 코드 페이지의 문자와 동작하도록 하기 위해 이 방법을 사용할 수 있습니다(단, 물론 운영 체제 설치가 지원되어야 함).

일반적으로는, 이런 방법 대신 현재 코드 페이지를 그대로 사용하거나, 문자와 코드 페이지에 대해 설명할 때 소개했던 `SetCodePage` 프로시저를 호출하여 개별 문자열에 대해 현재 코드 페이지를 변경합니다. 이 프로시저를 호출하는 경우는 두 가지 상황이 있습니다. 첫 번째 경우는 문자열의 형식을 알고 있으므로 해당 코드 페이지를 변경하는 것입니다(파일 또는 소켓에 의해 로드된 문자열의 경우). 두 번째 경우는 지정된 문자열을 변환하기 위해 호출하는 것입니다(나중에 설명하겠지만 이는 문자열을 다른 코드 페이지의 문자열에 대입할 때 자동으로 발생합니다).

더 컴팩트한 메모리 문자열 구조 때문에 `AnsiString` 타입을 계속 사용할 수도 있지만, 대부분의 경우 새 `UnicodeString` 타입을 사용하도록, 즉 문자열이 보통의 `String` 타입으로 선언된 상태 그대로 두고 코드를 변환하기 원할 것입니다. 특정 문자열 타입을 사용해야 하는 상황도 여전히 있습니다. 그러한 예로는 파일 로드나 저장, 데이터베이스로의 데이터 입출력, 코드를 문자당 8비트 형식으로 유지해야 하는 인터넷 프로토콜 사용과 같은 경우를 들 수 있습니다. 이러한 모든 경우 `AnsiString`을 사용하도록 기존 코드를 수정해야 합니다.

사용자 정의 문자열 타입 만들기

새 `AnsiString` 타입은 애플리케이션 컴파일 시 사용되는 기본 코드 페이지로 연결되는데, 이 새 `AnsiString` 타입을 사용하는 방법 외에도 동일한 메커니즘으로 사용자 정의 문자열 타입을 정의할 수 있습니다. 예를 들어 다음과 같은 코드로 `Latin-1` 문자열 타입을 정의할 수 있습니다.

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

procedure TFormLatinTest.btnNewTypeClick(Sender: TObject);
var
  str1: Latin1String;
begin
  str1 := 'a string with an accent: Cantù';
  Log('String: ' + str1);
```

이 문자열 타입은 다른 문자열 타입처럼 사용할 수 있지만 특정 코드 페이지에 연결됩니다. 따라서 이 문자열 타입을 사용할 경우 `Latin1String`을 `UnicodeString`으로 변환(예: 이 문자열 타입을 위의 `Log`에 대한 호출로 표시)하면 Delphi 컴파일러에서 변환 호출을 추가합니다. 위와 같은 코드의 마지막 줄에는 `_UStrFromLStr`에 대한 숨겨진 호출이 있으며, 이 호출은 `MultiByteToWideChar` Windows API가 실제 변환 작업을 실행할 때까지 `System` 유닛의 내부 함수를 호출합니다. 다음은

호출의 시퀀스입니다.

```

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

```

Windows API는 제대로 변환을 수행할 수 있지만 다양한 Windows 코드 페이지에서 사용 가능한 일부 문자를 Latin1으로 나타낼 수 없기 때문에 이러한 변환에는 손실의 가능성이 있습니다. 이에 대한 예로는 유로 통화 기호와 스마트 따옴표를 들 수 있습니다.

위의 **btnNewTypeClick** 메소드는 다음과 같은 문자열의 일부 세부 사항을 계속해서 표시합니다.

```

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

```

이 코드를 실행하면 다음과 같은 출력이 생성됩니다.

```

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

```

이 새로운 사용자 정의 문자열 타입이 표준 **AnsiString** 타입과 다르게 처리된다는 것을 보여주기 위해, 적어도 제 컴퓨터에서 제 로컬을 사용했을 경우 다르게 처리된다는 것을 보여주기 위해 저는 동일한 상위 끝 문자(#128 ~ #255)를 **AnsiString**과 **Latin1String**에 추가하는 테스트 메소드를 코딩하여 그룹으로 메모에 표시했습니다.

```

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

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

```

출력에서 처음 부분은 두 세트 간의 차이점을 강조하여 보여 줍니다. (표시되는 결과는 로캘에 따라 다양할 수 있습니다.)

```

128 - 135
Lati: ?E,f".??
Ansi: €E,f,,...†‡
136 - 143
Lati: ^?S<OEZE Ansi: ^%OS<E<E
144 - 151
Lati: E'""".--- Ansi: E'""".---
152 - 159
Lati: ~™š>œEžŸ
    
```

좀더 흥미로운 예는 키릴 자모와 같이 라틴 알파벳이 아닌 알파벳의 코드 페이지를 사용하는 것입니다. 그 예로 저는 두 번째 사용자 정의 문자열 타입을 다음과 같이 정의했습니다.

```

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

이전 코드와 매우 유사한 방법으로 이 문자열을 사용할 수 있지만 흥미로운 것은 127 이상의 숫자 값을 가진 문자와 같은 높은 순서의 문자를 사용하는 것입니다. 저는 for loop를 사용하는 몇 개의 문자를 선택했습니다.

```

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

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

이 메소드의 출력은 다음과 같이 표시됩니다.

```

String: a string with an accent: Cantu
Last char: 117
ElemSize: 1
Length: 30
CodePage: 1251
High end chars: --_™љ>њќћџ ŸŷЈѠГ!§Ë@€«~--
®İ°±Іігµ¶·ё№€»ЈSsіАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмно
прстуфхцчщъ
    
```

원래 값을 사용할 수 없기 때문에 악센트 부호가 있는 문자는 악센트 부호가 없는 해당 버전으로 *변환*된다는 것을 알 수 있습니다. (변환 뒤에 **WideCharToMulfiByte** 는 특정 상황에서 정상적으로 실패를 시도합니다. 예를 들어 스마트 따옴표는 물음표 대신에 곧은 따옴표로 낮춰지고 샘플 코드의 악센트 부호 표시 문자는 악센트를 잃게 됩니다.) 문자열 상수는 유니코드 문자열이며

`str1` 에 대한 대입은 암시적인 변환을 수행합니다. 사실상, 마지막 문자의 숫자 값은 다릅니다.

또한 이번에는 하이엔드 문자가 완전히 다릅니다. 원하는 결과를 얻으려면 다음과 같이 이중 캐스트를 코딩해 보십시오.

```
┆ CyrillicString(AnsiChar(I))
```

문자열에 있는 문자를 연결하여 변환하면 이러한 문자는 유니코드 문자로 처리됩니다.

UTF-8 문자열 관리

문자열 타입에 대한 새로운 내부 구조의 파생 효과 중 하나는 네이티브 방식으로 **UTF-8** 형식의 문자열을 관리할 수 있다는 것입니다. **UTF8String**이 문자열 타입의 별칭이었던 과거와는 다르게 이제 새로운 타입이 완벽히 인식됩니다. 변환은 자동으로 수행되며 모든 기존 **UTF-8** 문자열 조작 루틴이 새 특정 타입을 사용하기 위해 포팅되었습니다.

다음의 간단한 코드를 고려해 볼 수 있습니다.

```
┆ var
┆   str8: Utf8String;
┆   str16: string;
┆ begin
┆   str8 := 'Cantù';
┆   Memo1.Lines.Add('UTF-8');
┆   Memo1.Lines.Add('Length: ' + IntToStr(Length(str8)));
┆   Memo1.Lines.Add('5: ' + IntToStr(Ord(str8[5])));
┆   Memo1.Lines.Add('6: ' + IntToStr(Ord(str8[6])));
┆
┆   str16 := str8;
┆   Memo1.Lines.Add('UTF-16');
┆   Memo1.Lines.Add('Length: ' + IntToStr(Length(str16)));
┆   Memo1.Lines.Add('5: ' + IntToStr(Ord(str16[5])));
```

예상할 수 있듯이 **str8** 문자열은 6의 길이(6바이트를 의미)를 가지지만 **str16** 문자열은 5의 길이(10바이트를 의미)를 가집니다. **Length**는 변수 길이 표현이 문자열에 의해 표현된 유니코드 코드 포인트의 수와 일치하지 않는 경우 항상 문자열 요소의 수를 리턴합니다. 다음은 프로그램의 출력입니다.

```
┆ UTF-8
┆ Length: 6
┆ 5: 195
┆ 6: 185
┆
┆ UTF-16
┆ Length: 5
┆ 5: 249
```

그 이유는 처음 7비트 ANSI 공백을 제외한 문자가 최소 2개의 문자를 사용하도록 **UTF-8** 문자열이 변수 길이 구현을 사용하기 때문입니다. 앞에서 나온 *accented u*도 바로 이러한 경우입니다. 동일한 **UTF-8** 문자열을 **AnsiString** 변수에 대입하고 유사한 코드를 실행하면 다음과 같은 출력이 제공됩니다.

```
┆ ANSI Length: 5
┆ 5: 249
```

그러나 이번에는 문자열 길이 5가 5개 문자가 아닌 5바이트를 의미합니다.

UTF-8 형식에 대한 지원은 Delphi 2009에 대한 네이티브 문자열 구현인 UTF-16에 대한 지원만큼 완벽하지는 않지만 눈에 띄게 향상되었습니다. WideStrUtils 유닛에 UTF-8 조작에 대한 특정 루틴이 있으며 이 형식의 스트리밍 텍스트 파일에 대해 완벽한 지원이 제공됩니다(TEncoding 클래스 및 텍스트 파일 변환에 대해서는 “스트림 및 인코딩” 섹션에서 설명하도록 하겠습니다.) 핵심적인 사항은 이런 문자열에서 작업을 하여 명시적인 변환을 수행할 필요 없이 또한 변환을 수행할지 여부와 그 시기를 기억할 필요 없이 제어된 방법으로 표시할 수 있다는 사실이며 이는 작업에 많은 도움이 됩니다.

UnicodeString 타입 간의 변환 때문에 UTF-8 문자열의 일부 작업이 느려질 수는 있지만 컴파일러에 의해 강제로 수행되지 않는 별칭 타입 외의 특정 데이터 타입을 사용하는 것은 이 인코딩을 처리해야 하는 Delphi 개발자에게 중요한 차이를 만들게 됩니다.

이 특정 문자열 타입을 사용하는 기존 루틴 또는 새 루틴의 오버로드된 버전을 자유롭게 작성하여 추가 변환을 방지할 수도 있습니다.

문자열 변환

이제까지 UnicodeString 값을 AnsiString 또는 UTF8String 에 대입하면 적절한 변환이 이루어지는 것을 살펴보았습니다. 그와 유사하게, 지정된 코드 페이지를 포함한 AnsiString을 다른 코드 페이지에 기반한 다른 AnsiString에 대입하면 변환이 이루어집니다. 문자열을 변환이 발생하도록 요청하는 다른 코드 페이지에 대입하여 해당 문자열을 변환할 수도 있습니다.

```

type
  Latin1String = type AnsiString(28591);

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

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

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

```

위의 두 경우 모두에서 유로 기호는 Latin1 코드 페이지에서 표현될 수 없으므로 변환 시 손실이 발생하게 됩니다. SetCodePage 루틴 사용은 RawByteString 매개 변수에만 적용할 수 있으므로

결과적으로는 대입이 됩니다. 다음과 같은 출력이 표시됩니다.

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

변환은 코드의 속도를 저하시킬 수 있습니다

작업 화면 뒤에서 이루어지는 자동 변환은 시스템에서 개발자를 위해 많은 작업을 수행하기 때문에 매우 편리하지만 수행하고 있는 작업에 대해 주의를 기울여 고려하지 않으면 지속적인 변환과 문자열 복사 작업으로 인해 코드 속도가 저하되는 결과를 초래할 수 있습니다. 다음 코드를 고려해 보십시오.

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

두 문자열의 실제 문자열 타입에 따라 알고리즘은 매우 빠르거나 매우 느릴 수 있습니다. 데모에서는 첫 번째 실행으로 문자열(UnicodeString)을, 두 번째 실행으로 AnsiString과 UTF8String의 조합(최악의 경우, 각 대입에 대해 그러한 문자열과 조합을 UnicodeString 타입으로 변환하거나 그 반대로 변환해야 함)을 사용합니다. 다음은 10,000번 반복을 실행한 결과입니다.

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

10,000이 너무 많은 것 같겠지만 이 숫자는 10의 1000배이거나 10을 세번 곱한 수치입니다! 자, 그럼 50,000번 반복한다면 무슨 일이 일어날지 생각해볼 수 있습니다.

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

다시 한 번 10을 곱해야 합니다! 이 수는 점점 더 많은 문자열을 메모리에 여러 번 다시 할당해야 하기 때문에 기하급수적으로 증가하게 됩니다. 코드 속도는 변환에 의해 저하되기도 하지만 대부분의 경우 현재 문자열 크기가 지속적으로 증가하기 때문이라기보다 큰 용량의 새 임시 문자열을 작성해야 하기 때문에 저하됩니다. 다시 말해 간헐적으로 발생하는 암시적인 변환은 괜찮지만 변환이 루프 또는 재귀 루틴 내에서 발생하도록 해서는 절대 안됩니다!

중요한 것은 문자열 변환 경고를 활성화한 상태(사실상 기본값임)에서 프로그램을 컴파일하고 컴파일러가 변환 코드를 추가하는 위치를 확인할 수 있다는 사실입니다. 다른 타입의 문자열을 연결하는 데 사용되는 코드의 단일 줄에 대해 다음과 같은 경고가 표시됩니다.

```
w1057 Implicit string cast from 'UTF8String' to 'string'
w1057 Implicit string cast from 'AnsiString' to 'string'
w1058 Implicit string cast with potential Data loss from 'string'
to 'UTF8String'
```

모든 문자열을 모든 형식으로 표현할 수는 없기 때문에 “데이터가 유실될 수 있는” 문제가 발생합니다. 예를 들어 UnicodeString을 AnsiString에 대입하면 해당 작업을 하지 못하게 될 가능성이 있습니다. 문자열 변환 작업은 일반적이기 때문에 두 개의 해당 경고(암시적인 문자열

캐스트 및 데이터 유실 가능성이 포함된 암시적인 문자열 캐스트)는 기본적으로 해제되어 있습니다.

이러한 경고를 설정하게 되면 잠재적인 위험에 대해 알리는 많은 경고가 표시되지만 일반적인 프로그램에서도 그러한 경고는 많이 발생할 수 있으며 명시적인 타입 캐스트는 이러한 경고를 제거하는 것이 아니라 다른 경고 세트(명시적인 문자열 캐스트 및 잠재적인 데이터 손실을 포함한 명시적인 문자열 캐스트)로 변경하기만 합니다. 확인을 완료하면 이 경고를 해제하십시오!

문자열 상수를 문자열에 대입하면 일부 문자를 변환할 수 없는 경우 5번째 유사한 경고가 발생합니다. 이 경우 경고가 약간 다릅니다.

```
[DCC warning] StringConvertForm.pas(63): w2455 Narrowing given wide string constant lost informafion
```

이는 작업이 현실적으로 거의 불가능하기 때문에 제거해야 할 경고입니다.

프로그램 실행 속도를 저하시키는 암시적(다소 숨겨진) 변환의 또 다른 예로서 다음의 코드를 고려해 볼 수 있습니다.

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

str1 변수가 UnicodeString인 경우는 별 문제가 없지만 str1 변수가 AnsiString인 경우는 두 개의 변환이 발생할 수 있습니다. 문자열이 짧고 문자열 사본이 필요하기 때문에 이전 경우에서만큼은 나쁘지 않지만 백만번 반복하게 되면 약간의 오버헤드를 보입니다.

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

RAWBYTESTRING 사용

AnsiString을 매개 변수로 사용하여 루틴에 전달해야 하는 경우 어떻게 해야 할까요? 매개 변수가 인코딩을 사용하는 특정 문자열 타입에 대입되면 이 매개 변수는 데이터 손실 가능성을 포함한 적합한 타입으로 변환됩니다.

이것이 바로 Delphi 2009에서 사용자 정의 문자열 타입을 도입한 이유입니다. 이 문자열 타입은 RawByteString이라고 하며 다음과 같이 정의됩니다.

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

이 정의는 인코딩하지 않고 또는 더 정확히 말해 “인코딩하지 않음”이라고 표시한 자리표시자 \$ffff를 사용하여 문자열 타입을 생성하는 정의입니다. RawByteString은 AnsiString을 대입할 때 자동 변환하는 경우 첨부된 인코딩을 무시하는 바이트 문자열로 간주될 수 있습니다. 다시 말해 RawByteString 매개 변수로서 문자당 1바이트 문자열을 전달하면 다른 AnsiString 파생 타입과 달리 변환이 이루어지지 않습니다. 앞의 “문자열 변환” 섹션에서 설명한 바와 같이 SefCodePage 루틴을 호출하여 특정 변환을 수행할 수 있습니다.

이러한 경우 그러한 변환은 제네릭 및 문자 표현별로 1바이트를 유지하려는 사용자 정의 데이터 처리를 위해 문자열을 사용하는 코드의 문자열 또는 AnsiString 타입에 대한 편리한 대체 대안이 됩니다. (문자당 1바이트 Ansi 호환 문자열을 이 확장 지원과 혼동해서는 안됩니다. 지금까지

알려진 해결 방법은 문자열 처리 코드를 **UnicodeString** 타입으로 마이그레이션하는 것입니다. 이러한 새로운 추가 문자열 타입에 지나치게 현혹되지 마십시오.)

실제 문자열을 저장하기 위해 **RawByteString** 타입의 변수를 선언하지는 않아야 합니다. 정의되지 않은 코드 페이지가 제공되는 경우 이로 인해 정의되지 않은 동작이 발생하고 데이터가 손실될 위험이 있습니다. 반면에 문자열 같은 메모리 할당 및 표현을 사용하여 이진 데이터를 저장하는 게 목표라면 Delphi 이전 버전에서 **AnsiString**을 사용한 방식과 동일하게 **RawByteString**을 사용할 수 있습니다. **AnsiString**을 사용하는 비문자열 코드를 **RawByteString**으로 교체하는 것은 흥미로운 마이그레이션 경로가 될 것입니다.

지금부터는 **RawByteString** 타입을 매개 변수로 사용할 수 있는 일반적인 예에 중점을 두고 설명하겠습니다. 8비트 문자열에 대한 일부 정보를 표시하려면 다음 2개 선언 중 하나를 작성할 수 있습니다. (이 선언은 **RawTest** 데모의 주 형식 메소드입니다.)

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

두 메소드의 코드는 동일합니다(여기에서는 둘 중 하나의 코드만 나열함).

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

문자열을 적절하게 표시하기 위해 사용되는 **UnicodeString**으로 캐스트한 부분을 유의해서 보십시오. 이는 컴파일 시 리터럴 문자열이 코드 페이지가 정의되지 않은 문자열과 연결되어 데이터가 일반 **AnsiString** 처럼 처리되지 않도록 하기 위해 필요합니다. (**Log(str)**를 직접 사용하면 연결되지 않은 것처럼 동작합니다.)

스트림 및 인코딩

RTL 및 VCL로 작업할 때 Windows API를 호출하면서 애플리케이션 내 모든 문자열을 유니코드로 이동하는 것은 어렵지 않지만 파일에서 문자열을 읽고 쓰는 경우 문제가 다소 복잡해질 수 있습니다. 예를 들어 **TStrings** 파일 작업에 무슨 일이 일어나게 될까요?

Delphi 2009는 **TEncoding** 이라는 새 클래스를 사용하여 파일 인코딩을 처리합니다. 이 클래스는 .NET 프레임워크의 **System.Text.Encoding** 클래스를 약간 모방한 것입니다. **SysUtils** 유닛에 정의된 **TEncoding** 클래스에는 다음과 같이 Delphi에서 자동으로 지원되는 인코딩을 표현하는 몇 개의 하위 클래스가 있습니다. 이러한 클래스는 개발자 자신의 클래스를 추가할 수 있는 *표준 인코딩*입니다.)

```
type
  TEncoding = class
    TMBCEncoding = class(TEncoding)
      TUTF7Encoding = class(TMBCEncoding)
        TUTF8Encoding = class(TUTF7Encoding)
          TUnicodeEncoding = class(TEncoding)
```

```

| TBigEndianUnicodeEncoding = class(TUnicodeEncoding)

```

이러한 각 클래스의 한 개체는 클래스 데이터로서 **TEncoding** 클래스 내에서 사용 가능하며 해당 **getter** 함수와 클래스 속성을 포함합니다.

```

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

```

TEencoding 클래스에는 바이트 스트림에 대한 문자 읽기 및 쓰기를 위한 메소드가 있어 변환뿐만 아니라 **GetPreamble**이라고 하는 **BOM**을 처리하기 위한 특수 함수를 실행할 수 있습니다. 따라서 코드의 어느 위치에서나 다음과 같이 작성할 수 있습니다.

TSTRINGS 스트리밍

TStrings 클래스의 **ReadFromFile** 및 **WriteToFile** 메소드는 인코딩을 사용하여 호출할 수 있습니다. 특정 인코딩을 제공하지 않고 문자열 목록을 텍스트 파일에 작성하면 클래스는 **TEencoding.Default**를 사용하게 되는데, **TEencoding.Default**는 현재 Windows 코드 페이지에 의해 처음 발생 시 차례로 추출된 내부 **DefaultEncoding**을 사용합니다. 다시 말해 파일을 저장하면 이전과 동일한 ANSI 파일을 얻게 됩니다.

물론 강제로 파일을 다른 형식(예: UTF-16 형식)으로 쉽게 변환할 수 있습니다.

```

| Memo1.Lines.SaveToFile('test.txt', TEencoding.Unicode);

```

이렇게 하면 유니코드 **BOM** 또는 머리말을 사용하여 파일이 저장됩니다. 해당 **LoadFromFile** 작업을 수행하면 인코딩을 지정하지 않는 경우 로딩 메소드가 **TEencoding** 클래스의 **GetBufferEncoding** 메소드를 호출하게 되며, 이 메소드는 **BOM**의 존재 유무에 따라 인코딩을 결정합니다(**BOM**이 없는 경우 기본 ANSI 인코딩 사용).

LoadFromFile에서 인코딩을 지정하는 경우 어떻게 될까요? 개발자가 제공하는 인코딩은 파일의 실제 **BOM**과 무관하게 파일을 읽는 데 사용되며 에러를 일으키는 경우가 종종 있습니다. 개발자의 실수로 한 코드 페이지로 파일을 저장하고 다른 코드 페이지를 사용하여 파일을 강제로 업로드하게 하는 것과 같은 불일치가 있는 경우 예외가 발생할 수 있습니다. 인코딩된 파일이 **BOM** 없이 저장되었고 **ASCII** 파일이 아니라 **UTF** 파일로 인식되어야 하는 경우에는 예외가 발생하지 않는 것이 도움이 될 수 있습니다.

하지만 여기서는 파일을 저장하는 작업에 초점을 맞추겠습니다. 기존 Delphi 코드를 변경하지 않으면 프로그램은 파일을 ANSI로 저장합니다. 기존 프로그램이 유니코드 데이터를 처리하지 않으면 프로그램과 해당 파일은 이전 버전과 완벽하게 호환됩니다. 하지만 프로그램이 유니코드 데이터를 처리하는 경우는 어떻게 될까요? 다음 설계 시간 양식에서와 같이 다른 언어로 작성된 줄을 포함한 문자열 목록이 있다고 가정해 보겠습니다.

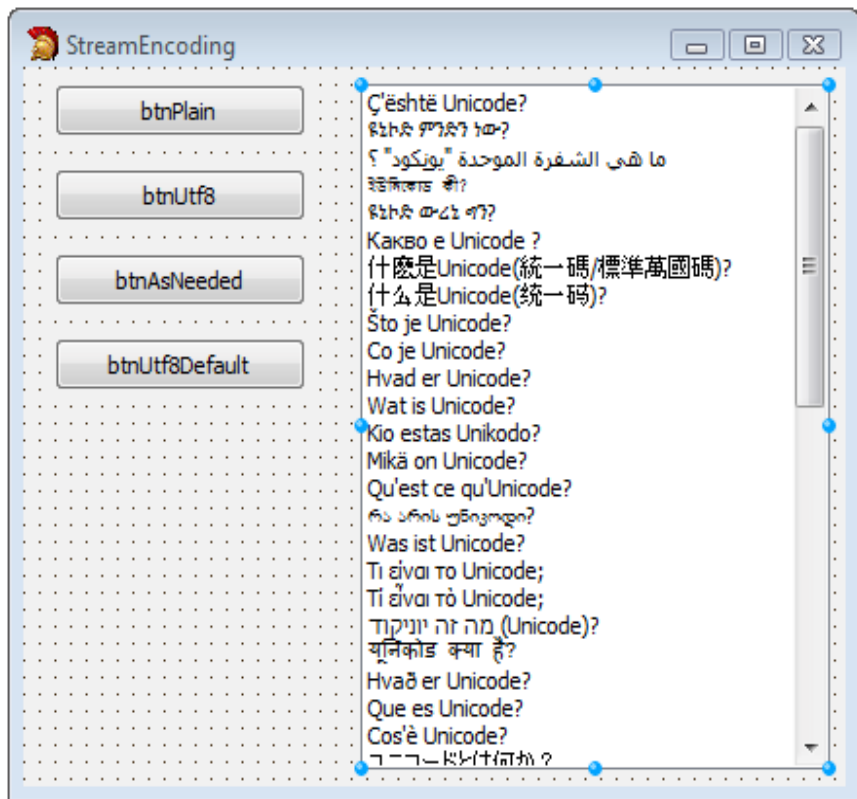


그림 1 문자열을 다른 언어로 표시하는 설계 양식

문자열 목록을 파일에 저장하고 다시 로드하는 Delphi 코드가 기존에 있다고 하면 다음과 같이 표시될 것입니다.

```

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

말할 필요도 없이 사용된 문자의 일부분만 ANSI 표현을 가지기 때문에 목록 상자에 수 많은 물음표가 표시되는 끔찍한 결과를 얻게 될 것입니다. 간단한 대안은 프로젝트의 두 번째 버튼 이벤트 처리에서처럼 코드를 변경하는 것입니다.

```

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

문자열 목록 로드 시 Delphi가 BOM에서 인코딩을 선택하므로 인코딩을

지정하지 않아도 됩니다. 필요하지 않는 한 데이터를 ANSI로 저장하는 것을 선호하는 경우 문자열 목록 내용을 확인하여 ASCII 또는 UTF-8로 저장할지를 결정할 수 있습니다.

```

procedure TFormStreamEncoding.btnAsNeededClick(Sender: TObject);
var
    strFileName: string;
    
```

```

    encoding1: TEncoding;
begin
    strFileName := 'AsNeededText.txt';
    encoding1 := TEncoding.Default;

    if ListBox1.Items.Text <>
        UnicodeString (AnsiString(ListBox1.Items.Text)) then
        encoding1 := TEncoding.UTF8;
    ListBox1.Items.SaveToFile(strFileName, Encoding1);

```

이 코드는 문자열을 `AnsiString`으로 변환하고 내용의 손실 없이 `UnicodeString`으로 다시 변환할 수 있는지 여부를 확인합니다. 매우 긴 문자열의 경우 비교와 함께 이러한 이중 변환을 하면 꽤 비용이 많이 들기 때문에 다음과 같은 대체 코드를 사용할 수 있습니다. 이 대체 코드는 특정 코드 페이지를 사용하므로 명확하지는 않지만 거의 유사합니다.

```

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

```

유사한 코드를 사용하여 상황에 따라 어떤 형식을 사용할지 결정할 수 있습니다. 실제 데이터에 상관 없이 모든 파일을 유니코드 인코딩(**UTF-8** 또는 **UTF-16**)으로 이동하는 것이 나올 것입니다. **UTF-16**을 사용하면 파일이 커질 수 있지만 저장 및 로드 시 변환이 줄어들게 됩니다.

그렇지만 기본 변환을 지정할 수 있는 방법이 없기 때문에 파일을 유니코드 인코딩으로 설정하면 클래스의 표준 동작을 변경하는 트릭을 사용하지 않는 한 모든 파일 저장 작업을 각각 변경해야 합니다. 그러한 *해킹*은 클래스 헬퍼의 형식으로 표시됩니다. 다음 코드를 고려해 보십시오.

```

type
    TStringHelper = class helper for TString
        procedure SaveToFile (const strFileName: string);
    end;

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

```

여기에서 상속된이라는 말은 기본 클래스를 호출하는 것을 의미하는 것이 아니라 클래스 헬퍼에 의해 도움을 받는 클래스를 호출한다는 의미입니다. 이제 다음과 같이 간단하게 코드를 작성하거나 유지할 수 있습니다.

```

    ListBox1.Items.SaveToFile(strFileName);

```

UTF8 또는 선택에 따라 다른 인코딩으로 저장할 수 있습니다.

결론: 유니코드와 VCL

Delphi 언어에서 유니코드 문자열이 지원되는 것은 획기적인 일로, Win32 API를 Wide 버전에 다시 매핑함으로써 마이그레이션을 훨씬 손쉽게 할 수 있게 되었지만 가장 중요한 근본적인 변화는 전체 RTL 및 VCL(Visual Component Library)이 유니코드를 완벽하게 지원한다는 것입니다. 구성 요소가 관리하는 모든 문자열 및 문자열 목록이 문자열로 선언되기 때문에 이제 새 UnicodeString 타입과 일치합니다.

그러나 일부 저수준 RTL 내부 영역은 다른 형식을 사용합니다. 예를 들어 속성 이름은 UTF-8에 기반을 두며 TypInfo 유닛에서 지원하는 RTTI 지원의 일부입니다. 일부 특수한 예외를 제외하고는 모든 영역이 UnicodeString 및 UTF-16으로 마이그레이션되었습니다.

유니코드 지원은 핵심적인 요소입니다. 그러나 그것이 국제적인 애플리케이션 빌드를 위한 지원을 향상하는 데 도움이 되는 유일한 기능은 아닙니다.

소스 코드 파일과 관련하여 원하는 형식으로 파일을 저장할 수 있지만 소스 코드에서 식별자 이름, 문자열, 주석 등을 위해 255 이상의 코드 포인트를 사용하는 경우 유니코드 형식을 사용해야 한다는 것을 명심해야 합니다. 편집기는 필요한 경우 특정 형식을 사용하도록 메시지를 표시하지만 개발자는 유니코드 소스 파일을 사용할 수 있습니다.

필자에 대하여

이 문서는 베스트 셀러 시리즈인 Mastering Delphi의 저작자 Marco Cantù가 Embarcadero Technologies 를 위해 작성하였습니다. 이 문서의 내용은 그의 최근 저서인 “Delphi 2009 핸드북” (<http://www.marcocantu.com/dh2009>)에서 발췌한 것입니다. Marco Cantù에 대한 정보는 그의 개인 블로그(<http://blog.marcocantu.com>)에서 읽을 수 있으며 전자 메일(marco.cantu@gmail.com)을 통해 연락할 수 있습니다.



EMBARCADERO
TECHNOLOGIES®

Embarcadero Technologies Inc.는 애플리케이션 개발자 및 데이터베이스 전문가가 자신이 선택한 환경에서 소프트웨어 애플리케이션을 설계, 빌드 및 실행하는 도구를 사용할 수 있도록 합니다. 전 세계 3백만 이상의 커뮤니티와 Fortune지 선정 100대 기업 중 90개 기업이 Embarcadero의 CodeGear™ 및 DatabaseGear™ 제품군을 기반으로 하여 생산성을 향상시키고 개방적인 협업 및 자유로운 혁신을 추구하고 있습니다. Embarcadero는 1993년에 설립되어 캘리포니아 샌프란시스코에 본사가 있으며 전 세계에 사무소를 두고 있습니다. Embarcadero의 온라인 주소는 www.embarcadero.com입니다. Embarcadero의 주요 제품인 DatabaseGear의 도구에는

ER/Studio®, DBArtisan®, Rapid SQL® 및 Embarcadero® Change Manager™가 있습니다.



데브기어는 미국 Embarcadero Technologies Inc.와 기존의 코드기어 한국 지사의 협력으로 전략적으로 설립된 엠바카데로 솔루션 전문 공급 기업입니다. 데브기어는 Delphi, C++Builder, JBuilder, Delphi Prism 등 개발툴 제품들과 ER/Studio, PowerSQL, DB Artisan, EA/Studio 등의 데이터베이스 툴 제품들에 대한 한국 시장에 공급은 물론 기술지원 및 교육을 제공합니다. 데브기어 웹 사이트는 <http://www.devgear.co.kr/> 이며 제품에 대한 문의는 ask@embarcadero.kr 로 하면 됩니다.