

Tech Notes

---

# Delphi in a Unicode World

Nick Hodges, Embarcadero Technologies

August 2008

---

**Corporate Headquarters**  
100 California Street, 12th Floor  
San Francisco, California 94111

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

**Asia-Pacific Headquarters**  
L7. 313 La Trobe Street  
Melbourne VIC 3000  
Australia

**Contents**

CHAPTER I: WHAT IS UNICODE, WHY YOU NEED IT, AND HOW TO WORK WITH IT IN DELPHI 2009.....	- 3 -
Introduction.....	- 3 -
What is Unicode?.....	- 3 -
Why Unicode?.....	- 3 -
A Word about Terminology.....	- 4 -
The New UnicodeString Type.....	- 4 -
Conclusion.....	- 6 -
CHAPTER II: NEW RTL FEATURES AND CLASSES TO SUPPORT UNICODE.....	- 7 -
Introduction.....	- 7 -
TCharacter Class.....	- 7 -
TEncoding Class.....	- 8 -
TStringBuilder.....	- 9 -
Declaring New String Types.....	- 10 -
Additional RTL Support for Unicode.....	- 10 -
StringElementSize.....	- 10 -
StringCodePage.....	- 10 -
Other RTL Features for Unicode.....	- 11 -
SetCodePage.....	- 12 -
Getting TBytes from Strings.....	- 13 -
Conclusion.....	- 13 -
CHAPTER III: UNICODIFYING YOUR CODE.....	- 14 -
Areas That Should "Just Work".....	- 14 -
General Use of String Types.....	- 14 -
The Runtime Library.....	- 14 -
The VCL.....	- 15 -
String Indexing.....	- 15 -
Length/Copy/Delete/SizeOf with Strings.....	- 15 -
Pointer Arithmetic on PChar.....	- 16 -
ShortString.....	- 16 -
Areas That Should be Reviewed.....	- 17 -
SaveToFile/LoadFromFile.....	- 17 -
Use of the Chr Function.....	- 18 -
Sets of Characters.....	- 18 -
Using Strings as Data Buffers.....	- 19 -
Calls to SizeOf on Buffers.....	- 19 -
Use of FillChar.....	- 19 -
Using Character Literals.....	- 20 -
Calls to Move.....	- 21 -
Read/ReadBuffer methods of TStream.....	- 21 -
Write/WriteBuffer.....	- 22 -
LeadBytes.....	- 22 -
TMemoryStream.....	- 22 -
TStringStream.....	- 23 -
MultiByte ToWideChar.....	- 23 -
SysUtils.AppendStr.....	- 23 -
GetProcAddress.....	- 23 -
Use of PChar() casts to enable pointer arithmetic on non-char based pointer types.....	- 24 -
Variant open array parameters.....	- 24 -
CreateProcessW.....	- 25 -

Passing in a string constant.....	- 25 -
Passing in a constant expression .....	- 25 -
Passing in a string with a Reference Count of -1: .....	- 26 -
Code to search for .....	- 26 -
APPENDICES .....	- 27 -
Embarcadero and Partner Blog Entries about Unicode .....	- 27 -
Embarcadero Developer Network Videos about Unicode.....	- 27 -
Get off your ASCII and expand your business to global markets .....	- 27 -
Migrating your Projects to Delphi 2009 – It's easy!.....	- 28 -
Additional Sources of Delphi 2009 Information .....	- 28 -
Additional Sources of Unicode Information .....	- 28 -

# CHAPTER I: WHAT IS UNICODE, WHY YOU NEED IT, AND HOW TO WORK WITH IT IN DELPHI 2009

This Chapter discusses Unicode, how Delphi Developers can benefit from using Unicode, and how Unicode is implemented in Delphi 2009

## INTRODUCTION

The Internet has broken down geographical barriers that enable world-wide software distribution. As a result, applications can no longer live in a purely ANSI-based environment. The world has embraced Unicode as the standard means of transferring text and data. Since it provides support for virtually any writing system in the world, Unicode text is now the norm throughout the global technological ecosystem.

## WHAT IS UNICODE?

[Unicode](#) is a character encoding scheme that allows virtually all alphabets to be encoded into a single character set. Unicode allows computers to manage and represent text most of the world's writing systems. Unicode is managed by [The Unicode Consortium](#) and [codified in a standard](#). More simply put, Unicode is a system for enabling everyone to use each other's alphabets. Heck, there is even [a Unicode version of Klingon](#).

This article isn't meant to give you a full rundown of exactly what Unicode is and how it works; instead it is meant to get you going on using Unicode within Delphi 2009. If you want a good overview of Unicode, Joel Spolsky has a great article entitled "[The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets \(No Excuses!\)](#)" which is highly recommended reading. As Joel clearly points out "IT'S NOT THAT HARD". This chapter will discuss why Unicode is important, and how Delphi will implement the new `UnicodeString` type.

## WHY UNICODE?

Among the many new features found in Delphi 2009 is the imbuing of Unicode throughout the product. The default string in Delphi is now a Unicode-based string. Since Delphi is largely built with Delphi, the IDE, the compiler, the RTL, and the VCL all are fully Unicode-enabled.

The move to Unicode in Delphi is a natural one. Windows itself is fully Unicode-aware, so it is only natural that applications built for it, use a Unicode string as the default string. And for Delphi developers, the benefits don't stop merely at being able to use the same string type as Windows.

The addition of Unicode support provides Delphi developers with a great opportunity. Delphi developers now can read, write, accept, produce, display, and deal with Unicode data – and it's all built right into the product. With only few, or in some cases to zero code changes, your applications can be ready for any kind of data you, your customers or your users can throw at it. Applications that previously restricted to ANSI encoded data can be easily modified to handle almost any character set in the world.

Delphi developers will now be able to serve a global market with their applications -- even if they don't do anything special to localize or internationalize their applications. Windows itself supports many different localized versions, and Delphi applications need to be able to adapt and work on machines running any of the large number of locales that Windows supports, including the Japanese, Chinese, Greek, or Russian versions of Windows. Users of your software may be entering non-ANSI text into your application or using non-ANSI based path names. ANSI-based applications won't always work as desired in those scenarios. Windows applications built with a fully Unicode-enabled Delphi will be able to handle and work in those situations. Even if you don't translate your application into any other spoken languages, your application still needs to be able to work properly -- no matter what the end user's locale is.

For existing ANSI-based Delphi applications, then opportunity to localize applications and expand the reach of those applications into Unicode-based markets is potentially huge. And if you do want to localize your applications, Delphi makes that very easy, especially now at design-time. The Integrated Translation Environment (ITE) enables you to translate, compile, and deploy an application right in the IDE. If you require external translation services, the IDE can export your project in a form that translators can use in conjunction with the deployable External Translation Manager. These tools work together with the Delphi IDE for both Delphi and C++Builder to make localizing your applications a smooth and easy to manage process.

The world is Unicode-based, and now Delphi developers can be a part of that in a native, organic way. So if you want to be able to handle Unicode data, or if you want to sell your applications to emerging and global markets, you can do it with Delphi 2009.

## A WORD ABOUT TERMINOLOGY

Unicode encourages the use of some new terms. For instance the idea of "character" is a bit less precise in the world of Unicode than you might be used to. In Unicode, the more precise term is "code point". In Delphi 2009, the `sizeof(Char)` is 2, but even that doesn't tell the whole story. Depending on the encoding, it is possible for a given character to take up more than two bytes. These sequences are called "Surrogate Pairs". So a code point is a unique code assigned an element defined by the Unicode.org. Most commonly that is a "character", but not always.

Another term you will see in relation to Unicode is "BOM", or Byte Order Mark, and that is a very short prefix used at the beginning of a text file to indicate the type of encoding used for that text file. MSDN has [a nice article on what a BOM is](#). The new `TEncoding` Class (to be discussed in Chapter II) has a class method called `GetPreamble` which returns the BOM for a given encoding.

Now that all that has been explained, we'll look at how Delphi 2009 implements a Unicode-based string.

## THE NEW UNICODESTRING TYPE

The default string in Delphi 2009 is the new `UnicodeString` type. By default, the `UnicodeString` type will have an affinity for UTF-16, the same encoding used by Windows. This is a change from previous versions which had `AnsiString` as the default type. The Delphi RTL has in the past included the `WideString` type to handle Unicode data, but this type is not reference-counted as the `AnsiString` type is, and thus isn't as full-featured as Delphi developers expect the default string to be.

For Delphi 2009, a new `UnicodeString` type has been designed, that incorporates the capabilities of both the `AnsiString` and `WideString` types. A `UnicodeString` can contain

either a Unicode-sized character, or an ANSI byte-sized character. (Note that both the `AnsiString` and `WideString` types will remain in place.) The `Char` and `PChar` types will map to `WideChar` and `PWideChar`, respectively. Note, as well, that no string types have disappeared. All the types that developers are used to still exist and work as before.

However, for Delphi 2009, the default string type will be equivalent to `UnicodeString`. In addition, the default `Char` type is `WideChar`, and the default `PChar` type is `PWideChar`.

That is, the following code is declared by the compiler:

```
type
  string = UnicodeString;
  Char = WideChar;
  PChar = PWideChar;
```

`UnicodeString` is assignment compatible with all other string types; however, assignments between `AnsiStrings` and `UnicodeStrings` will do type conversions as appropriate. Thus, an assignment of a `UnicodeString` type to an `AnsiString` type could result in data-loss. That is, if a `UnicodeString` contains high-order byte data, a conversion of that string to `AnsiString` will result in a loss of that high-order byte data. The important thing to note here is that this new `UnicodeString` behaves pretty much like strings always have (with the notable exception of their ability to hold Unicode data, of course). You can still add any string data to them, you can index them, you can concatenate them with the '+' sign, etc.

For example, instances of a `UnicodeString` will still be able to index characters. Consider the following code:

```
var
  MyChar: Char;
  MyString: string;
begin
  MyString := 'This is a string';
  MyChar := MyString[1];
end;
```

The variable `MyChar` will still hold the character found at the first index position, i.e. 'T'. This functionality of this code hasn't changed at all. Similarly, if we are handling Unicode data:

```
var
  MyChar: Char;
  MyString: string;
begin
  MyString := '世界您好';
  MyChar := MyString[1];
end;
```

The RTL provides helper functions that enable users to do explicit conversions between codepages and element size conversions. If the user is using the `Move` function on the character array, they cannot make assumptions about the element size.

As you can imagine, this new string type has ramifications for existing code. With Unicode, it is no longer true that one `Char` represents one Byte. In fact, it isn't even always true that one `Char` is equal to two bytes! As a result, you may have to make some adjustments to your code. However, we've worked very hard to make the transition a smooth one, and we are confident that you'll be able to be up and running quite quickly.

Chapters II and III will discuss further the new UnicodeString type, talk about some of the new features of the RTL that support Unicode enablement, and then discuss specific coding idioms that you'll want to look for in your code. This series should help make your transition to Unicode a smooth and painless endeavor.

## CONCLUSION

With the addition of Unicode as the default string, Delphi can accept, process, and display virtually any alphabet or code page in the world. Applications you build with Delphi 2009 will be able to accept, display, and handle Unicode text with ease, and they will work much better in almost any Windows locale. Delphi developers can now easily localize and translate their applications to enter markets that they have previously been more difficult to enter. It's a Unicode world out there, and now your Delphi apps can live in it.

In Chapter II, we'll discuss the changes and updates to the Delphi Runtime Library that will enable you to work easily with Unicode strings.

## CHAPTER II: NEW RTL FEATURES AND CLASSES TO SUPPORT UNICODE

This Chapter will cover the new features of the Delphi 2009 Runtime Library that will help handle Unicode strings.

### INTRODUCTION

In Chapter I, we saw how Unicode support is a huge benefit for Delphi developers by enabling communication with all characters set in the Unicode universe. We saw the basics of the `UnicodeString` type and how it will be used in Delphi

In this chapter, we'll look at some of the new features of the Delphi Runtime Library that support Unicode and general string handling.

### TCHARACTER CLASS

The Tiburon RTL includes a new class called `TCharacter`, which is found in the `Character` unit. It is a sealed class that consists entirely of static class functions. Developers should not create instances of `TCharacter`, but rather merely call its static class methods directly. Those class functions do a number of things, including:

- Convert characters to upper or lower case
- Determine whether a given character is of a certain type, i.e. is the character a letter, a number, a punctuation mark, etc.

`TCharacter` uses the standards set forth by the Unicode consortium.

Developers can use the `TCharacter` class to do many things previously done with sets of chars. For instance, this code:

```
uses
  Character;

begin
  if MyChar in ['a'...'z', 'A'...'Z'] then
  begin
    ...
  end;
end;
```

can be easily replaced with

```
uses
  Character;

begin
  if TCharacter.IsLetter(MyChar) then
  begin
    ...
  end;
end;
```



The `Character` unit also contains a number of standalone functions that wrap up the functionality of each class function from `TCharacter`, so if you prefer a simple function call, the above can be written as:

```
uses
  Character;

begin
  if IsLetter(MyChar) then
  begin
    ...
  end;
end;
```

Thus the `TCharacter` class can be used to do most any manipulation or checking of characters that you might care to do.

In addition, `TCharacter` contains class methods to determine if a given character is a high or low surrogate of a surrogate pair.

## TENCODING CLASS

The Tiburon RTL also includes a new class called `TEncoding`. Its purpose is to define a specific type of character encoding so that you can tell the VCL what type of encoding you want used in specific situations.

For instance, you may have a `TStringList` instance that contains text that you want to write out to a file. Previously, you would have written:

```
begin
  ...
  MyStringList.SaveToFile('SomeFilename.txt');
  ...
end;
```

and the file would have been written out using the default ANSI encoding. That code will still work fine – it will write out the file using ANSI string encoding as it always has, but now that Delphi supports Unicode string data, developers may want to write out string data using a specific encoding. Thus, `SaveToFile` (as well as `LoadFromFile`) now take an optional second parameter that defines the encoding to be used:

```
begin
  ...
  MyStringList.SaveToFile('SomeFilename.txt', TEncoding.Unicode);
  ...
end;
```

Execute the above code and the file will be written out as a Unicode (UTF-16) encoded text file.

`TEncoding` will also convert a given set of bytes from one encoding to another, retrieve information about the bytes and/or characters in a given string or array of characters, convert any string into an array of byte (`TBytes`), and other functionality that you may need with regard to the specific encoding of a given string or array of chars.

The TEncoding class includes the following class properties that give you singleton access to a TEncoding instance of the given encoding:

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

The Default property refers to the ANSI active codepage. The Unicode property refers to UTF-16.

TEncoding also includes the

```
class function TEncoding.GetEncoding(CodePage: Integer): TEncoding;
```

that will return an instance of TEncoding that has the affinity for the code page passed in the parameter.

In addition, it includes following function:

```
function GetPreamble: TBytes;
```

which will return the correct BOM for the given encoding.

TEncoding is also interface compatible with the .Net class called Encoding.

## TSTRINGBUILDER

The RTL now includes a class called TStringBuilder. Its purpose is revealed in its name – it is a class designed to “build up” strings. TStringBuilder contains any number of overloaded functions for adding, replacing, and inserting content into a given string. The string builder class makes it easy to create single strings out of a variety of different data types. All of the Append, Insert, and Replace functions return an instance of TStringBuilder, so they can easily be chained together to create a single string.

For example, you might choose to use a TStringBuilder in place of a complicated Format statement. For instance, you might write the following code:

```
procedure TForm86.Button2Click(Sender: TObject);
var
  MyStringBuilder: TStringBuilder;
  Price: double;
begin
  MyStringBuilder := TStringBuilder.Create('');
  try
    Price := 1.49;
    Label1.Caption := MyStringBuilder.Append('The apples are
      $').Append(Price).
      Append(' a pound.').ToString;
  finally
    MyStringBuilder.Free;
  end;
end;
```

TStringBuilder is also interface compatible with the .Net class called StringBuilder.

## DECLARING NEW STRING TYPES

Tiburon's compiler enables you to declare your own string type with an affinity for a given codepage. There is any number of code pages available. (MSDN has [a nice rundown of available codepages.](#)) For instance, if you require a string type with an affinity for ANSI-Cyrillic, you can declare:

```
type
  // The code page for ANSI-Cyrillic is 1251
  CyrillicString = type AnsiString(1251);
```

And the new String type will be a string with an affinity for the Cyrillic code page.

## ADDITIONAL RTL SUPPORT FOR UNICODE

The RTL adds a number of routines that support the use of Unicode strings.

## STRINGELEMENTSIZE

StringElementSize returns the typical size for an element (code point) in a given string. Consider the following code:

```
procedure TForm88.Button3Click(Sender: TObject);
var
  A: AnsiString;
  U: UnicodeString;
begin
  A := 'This is an AnsiString';
  Memo1.Lines.Add('The ElementSize for an AnsiString is: ' +
    IntToStr(StringElementSize(A)));
  U := 'This is a UnicodeString';
  Memo1.Lines.Add('The ElementSize for an UnicodeString is: ' +
    IntToStr(StringElementSize(U)));
end;
```

The result of the code above will be:

```
The ElementSize for an AnsiString is: 1
The ElementSize for an UnicodeString is: 2
```

## STRINGCODEPAGE

StringCodePage will return the Word value that corresponds to the codepage for a given string.

Consider the following code:

```

procedure TForm88.Button2Click(Sender: TObject);
type
  // The code page for ANSI-Cyrillic is 1251
  CyrillicString = type AnsiString(1251);
var
  A: AnsiString;
  U: UnicodeString;
  U8: UTF8String;
  C: CyrillicString;
begin
  A := 'This is an AnsiString';
  Mem1.Lines.Add('AnsiString Codepage: ' + IntToStr(StringCodePage(A)));
  U := 'This is a UnicodeString';
  Mem1.Lines.Add('UnicodeString Codepage: ' +
    IntToStr(StringCodePage(U)));
  U8 := 'This is a UTF8string';
  Mem1.Lines.Add('UTF8string Codepage: ' +
    IntToStr(StringCodePage(U8)));
  C := 'This is a CyrillicString';
  Mem1.Lines.Add('CyrillicString Codepage: ' +
    IntToStr(StringCodePage(C)));
end;

```

The above code will result in the following output:

```

The Codepage for an AnsiString is: 1252
The Codepage for an UnicodeString is: 1200
The Codepage for an UTF8string is: 65001
The Codepage for an CyrillicString is: 1251

```

## OTHER RTL FEATURES FOR UNICODE

There are a number of other routines for converting strings of one codepage to another. Including:

```

UnicodeStringToUCS4String
UCS4StringToUnicodeString
UnicodeToUtf8
Utf8ToUnicode

```

In addition the RTL also declares a type called `RawByteString` which is a string type with no encoding affiliated with it:

```

RawByteString = type AnsiString($FFFF);

```

The purpose of the `RawByteString` type is to enable the passing of string data of any code page without doing any codepage conversions. This is most useful for routines that do not care about specific encoding, such as byte-oriented string searches. Normally, this would mean that parameters of routines that process strings without regard for the strings code page should be of type `RawByteString`. Declaring variables of type `RawByteString` should rarely, if ever, be done as this can lead to undefined behavior and potential data loss.

In general, string types are assignment compatible with each other.

For instance:

```

MyUnicodeString := MyAnsiString;

```

will perform as expected – it will take the contents of the `AnsiString` and place them into a `UnicodeString`. You should in general be able to assign one string type to another, and the compiler will do the work needed to make the conversions, if possible.

Some conversions, however, can result in data loss, and one must watch out this when moving from one string type that includes Unicode data to another that does not. For instance, you can assign `UnicodeString` to an `AnsiString`, but if the `UnicodeString` contains characters that have no mapping in the active ANSI code page at runtime, those characters will be lost in the conversion. Consider the following code:

```
procedure TForm88.Button4Click(Sender: TObject);
var
  U: UnicodeString;
  A: AnsiString;
begin
  U := 'This is a UnicodeString';
  A := U;
  Memo1.Lines.Add(A);
  U := 'Добро пожаловать в мир Юникода с использованием Дельфи
      2009!!!';
  A := U;
  Memo1.Lines.Add(A);
end;
```

The output of the above when the current OS code page is 1252 is:

```
This is a UnicodeString
????? ??????????? ? ??? ??????? ? ????????????????? ?????? 2009!!
```

As you can see, because Cyrillic characters have no mapping in Windows-1252, information was lost when assigning this `UnicodeString` to an `AnsiString`. The result was gibberish because the `UnicodeString` contained characters not representable in the code page of the `AnsiString`, those characters were lost and replaced by the question mark when assigning the `UnicodeString` to the `AnsiString`.

## SETCODEPAGE

`SetCodePage`, declared in the `System.pas` unit as

```
procedure SetCodePage(var S: AnsiString; CodePage: Word; Convert: Boolean);
```

is a new RTL function that sets a new code page for a given `AnsiString`. The optional `Convert` parameter determines if the payload itself of the string should be converted to the given code page. If the `Convert` parameter is `False`, then the code page for the string is merely altered. If the `Convert` parameter is `True`, then the payload of the passed string will be converted to the given code page.

`SetCodePage` should be used sparingly and with great care. Note that if the codepage doesn't actually match the existing payload (i.e. `Convert` is set to `False`), then unpredictable results can occur. Also if the existing data in the string is converted and the new codepage doesn't have a representation for a given original character, data loss can occur.

## GETTING TBYTES FROM STRINGS

The RTL also includes a set of overloaded routines for extracting an array of bytes from a string. As we'll see in Part III, it is recommended that instead of using string as a data buffer, you use TBytes instead. The RTL makes it easy by providing overloaded versions of BytesOf() that takes as a parameter the different string types.

## CONCLUSION

Delphi 2009's Runtime Library is now completely capable of supporting the new UnicodeString. It includes new classes and routines for handling, processing, and converting Unicode strings, for managing codepages, and for ensuring an easy migration from earlier versions.

In Chapter III, we'll cover the specific code constructs that you'll need to look out for in ensuring that your code is Unicode ready.

## CHAPTER III: UNICODIFYING YOUR CODE

This Chapter describes what you need to do to get your code ready for Delphi 2009.

As discussed in Chapter I, we saw Delphi 2009 will use by default a UTF-16 based string. As a result, certain code idioms within existing code may need to be changed. In general, the large majority of existing code will work just fine with Delphi 2009. As you'll see, most of the code changes that need to be made are quite specific and somewhat esoteric. However, some specific code idioms will need to be reviewed and perhaps have changes made to ensure that the code works properly with `UnicodeString`.

For example, any code that manipulates or does pointer operations on strings should be examined for Unicode compatibility. More specifically, any code that:

- Assumes that `SizeOf(Char)` is 1
- Assumes that the `Length` of a string is equal to the number of bytes in the string
- Writes or reads strings from some persistent storage or uses a string as a data buffer

should be reviewed to ensure that those assumptions are not persisted in code. Code that writes to or reads from persistent storage needs to ensure that the correct number of bytes are being read or written, as a single byte no longer represents a single character.

Generally, any needed code changes should be straightforward and can be done with a minimal amount of effort.

### AREAS THAT SHOULD "JUST WORK"

This section discusses area of code that should continue to work, and should not require any changes to work properly with the new `UnicodeString`. All of the VCL and RTL have been updated to work as expected in Delphi 2009, and with very, very few exceptions, such will be the case. For instance, `TStringList` is now completely Unicode-aware, and all existing `TStringList` code should work as before. However, `TStringList` has been enhanced to work specifically with Unicode, so if you want to take advantage of that new functionality, you can, but you need not if you don't want to.

### GENERAL USE OF STRING TYPES

In general, code that uses the string type should work as before. There is no need to re-declare string variables as `AnsiString` types, except as discussed below. String declarations should only be changed to be `AnsiString` when dealing with storage buffers or other types of code that uses the string as a data buffer.

### THE RUNTIME LIBRARY

The runtime library additions are discussed extensively above, however there is no mention of a new unit added to the RTL – `AnsiString.pas`. This unit exists for backwards compatibility with code that chooses to use or requires the use of `AnsiString` within it.

Runtime library code runs as expected and in general requires no change. The areas that do need change are described below.

## THE VCL

The entire VCL is Unicode aware. All existing VCL components work right out of the box just as they always have. The vast majority of your code using the VCL should continue to work as normal. We've done a lot of work to ensure that the VCL is both Unicode ready and backwards compatible. Normal VCL code that doesn't do any specific string manipulation will work as before.

## STRING INDEXING

String Indexing works exactly as before, and code that indexes into strings doesn't need to be changed:

```
var
  S: string;
  C: Char;
begin
  S := 'This is a string';
  C := S[1]; // C will hold 'T', but of course C is a WideChar
end;
```

## LENGTH/COPY/DELETE/SIZEOF WITH STRINGS

Copy will still work as before without change. So will Delete and all the SysUtils-based string manipulation routines.

Calls to Length(SomeString) will, as always, return the number of elements in the passed string.

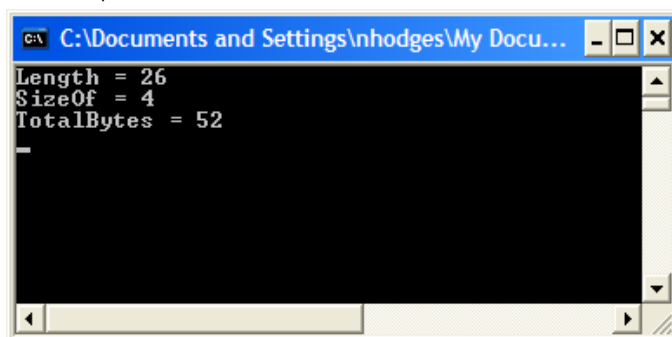
Calls to SizeOf on any string identifier will return 4, as all string declarations are references and the size of a pointer is 4.

Calls to Length on any string will return the number of elements in the string.

Consider the following code:

```
var
  S: string;
begin
  S := 'abcdefghijklmnopqrstuvwxy';
  WriteLn('Length = ', Length(S));
  WriteLn('SizeOf = ', SizeOf(S));
  WriteLn('TotalBytes = ', Length(S) * SizeOf(S[1]));
  ReadLn;
end.
```

The output of the above is as follows:



```
Length = 26
SizeOf = 4
TotalBytes = 52
```



## POINTER ARITHMETIC ON PCHAR

Pointer arithmetic on PChar should continue to work as before. The compiler knows the size of PChar, so code like the following will continue to work as expected:

```
var
p: PChar;
MyString: string;
begin
  ...
  p := @MyString[1];
  Inc(p);
  ...
end;
```

This code will work exactly the same as with previous versions of Delphi – but of course the types are different: PChar is now a PWideChar and MyString is now a UnicodeString.

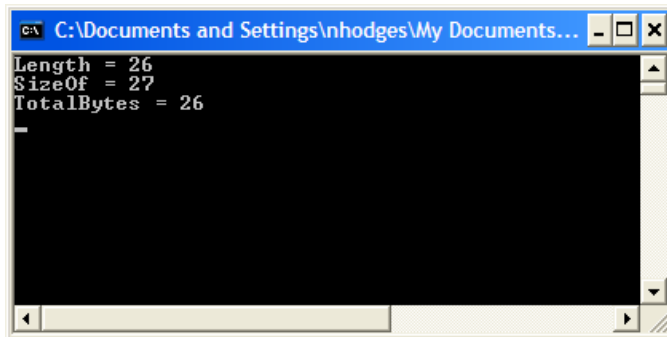
## SHORTSTRING

ShortString remains unchanged in both functionality and declaration, and will work just as before.

ShortString declarations allocate a buffer for a specific number of AnsiChars. Consider the following code:

```
var
  S: string[26];
begin
  S:= 'abcdefghijklmnopqrstuvwxyz';
  WriteLn('Length = ', Length(S));
  WriteLn('SizeOf = ', SizeOf(S));
  WriteLn('TotalBytes = ', Length(S) * SizeOf(S[1]));
  ReadLn;
end.
```

It has the following output:



```
C:\Documents and Settings\nhodes\My Documents...
Length = 26
SizeOf = 27
TotalBytes = 26
-
```

Note that the total bytes of the alphabet is 26 – showing that the variable is holding AnsiChars.

In addition, consider the following code:

```
type
TMyRecord = record
  String1: string[20];
  String2: string[15];
end;
```

This record will be laid out in memory exactly as before – it will be a record of two `AnsiStrings` with `AnsiChars` in them. If you've got a `File of Rec` of a record with short strings, then the above code will work as before, and any code reading and writing such a record will work as before with no changes.

However, remember that `Char` is now a `WideChar`, so if you have some code that grabs those records out of a file and then calls something like:

```
var
MyRec: TMyRecord;
SomeChar: Char;
begin
// Grab MyRec from a file...
SomeChar := MyRec.String1[3];
...
end;
```

then you need to remember that `SomeChar` will convert the `AnsiChar` in `String1[3]` to a `WideChar`. If you want this code to work as before, change the declaration of `SomeChar`:

```
var
MyRec: TMyRecord;
SomeChar: AnsiChar; // Now declared as an AnsiChar for the shortstring
                    index
begin
// Grab MyRec from a file...
SomeChar := MyRec.String1[3];
...
end;
```

## AREAS THAT SHOULD BE REVIEWED

This next section describes the various semantic code constructs that should be reviewed in existing code for Unicode compatibility. Because `Char` now equals `WideChar`, assumptions about the size in bytes of a character array or string may be invalid. The following lists a number of specific code constructs that should be examined to ensure that they are compatible with the new `UnicodeString` type.

## SAVETOFILE/LOADFROMFILE

`SaveToFile` and `LoadFromFile` calls could very well go under the “Just Works” section above, as these calls will read and write just as they did before. However, you may want to consider using the new overloaded versions of these calls if you are going to be dealing with Unicode data when using them.

For instance, `TStrings` now includes the following set of overloaded methods:

```
procedure SaveToFile(const FileName: string); overload; virtual;
procedure SaveToFile(const FileName: string; Encoding: TEncoding);
    overload; virtual;
```

The second method above is the new overload that includes an encoding parameter that determines how the data will be written out to the file. (See above for an explanation of the `TEncoding` type.) If you call the first method above, the string data will be saved as it always has been – as ANSI data. Therefore, your existing code will work exactly as it always has.

However, if you put some Unicode string data into the text to be written out, you will need to use the second overload, passing a specific `TEncoding` type. If you do not, the strings will be written out as ANSI data, and data loss will likely result.

Therefore, the best idea here would be to review your `SaveToFile` and `LoadFromFile` calls, and add a second parameter to them to indicate how you'd like your data saved. If you don't think you'll ever be adding or using Unicode strings, though, you can leave things as they are.

## USE OF THE CHR FUNCTION

Existing code that needs to create a `Char` from an integer value may make use of the `Chr` function. Certain uses of the `Chr` function may result in the following error:

```
[DCC Error] PasParser.pas(169): E2010 Incompatible types: 'AnsiChar' and 'Char'
```

If code using the `Chr` function is assigning the result to an `AnsiChar`, then this error can easily be removed by replacing the `Chr` function with a cast to `AnsiChar`.

So, this code

```
MyChar := chr(i);
```

Can be changed to

```
MyChar := AnsiChar(i);
```

## SETS OF CHARACTERS

Probably the most common code idiom that will draw the attention of the compiler is the use of characters in sets. In the past, a character was one byte, so holding characters in a set was no problem. But now, `Char` is declared as a `WideChar`, and thus cannot be held in a set any longer. So, if you have some code that looks like this:

```
procedure TDemoForm.Button1Click(Sender: TObject);
var
  C: Char;
begin
  C := Edit1.Text[1];

  if C in ['a'..'z', 'A'..'Z'] then
  begin
    Labell.Caption := 'It is there';
  end;
end;
```

and you compile it, you'll get a warning that looks something like this:

```
[DCC Warning] Unit1.pas(40): W1050 WideChar reduced to byte char in set expressions. Consider using 'CharInSet' function in 'SysUtils' unit.
```

You can, if you like, leave the code that way – the compiler will “know” what you are trying to do and generate the correct code. However, if you want to get rid of the warning, you can use the new `CharInSet` function:

```
if CharInSet(C, ['a'..'z', 'A'..'Z']) then
begin
  Labell.Caption := 'It is there';
end;
```

The `CharInSet` function will return a `Boolean` value, and compile without the compiler warning.

## USING STRINGS AS DATA BUFFERS

A common idiom is to use a `string` as a data buffer. It's common because it's been easy -- manipulating strings is generally pretty straight forward. However, existing code that does this will almost certainly need to be adjusted given the fact that `string` now is a `UnicodeString`.

There are a couple of ways to deal with code that uses a `string` as a data buffer. The first is to simply declare the variable being used as a data buffer as an `AnsiString` instead of `string`. If the code uses `Char` to manipulate bytes in the buffer, declare those variables as `AnsiChar`. If you choose this route, all your code will work as before, but you do need to be careful that you've explicitly declared all variables accessing the string buffer to be ANSI types.

The second and preferred way dealing with this situation is to convert your buffer from a `string` type to an array of bytes, or `TBytes`. `TBytes` is designed specifically for this purpose, and works as you likely were using the `string` type previously.

## CALLS TO SIZEOF ON BUFFERS

Calls to `SizeOf` when used with character arrays should be reviewed for correctness. Consider the following code:

```
procedure TDemoForm.Button1Click(Sender: TObject);
var
var
  P: array[0..16] of Char;
begin
  StrPCopy(P, 'This is a string');
  Memo1.Lines.Add('Length of P is ' + IntToStr(Length(P)));
  Memo1.Lines.Add('Size of P is ' + IntToStr(SizeOf(P)));
end;
```

This code will display the following in Memo1:

```
Length of P is 17
Size of P is 34
```

In the above code, `Length` will return the number of characters in the given string (plus the null termination character), but `SizeOf` will return the total number of Bytes used by the array, in this case 34, i.e. two bytes per character. In previous versions, this code would have returned 17 for both.

## USE OF FILLCHAR

Calls to `FillChar` need to be reviewed when used in conjunction with strings or a character. Consider the following code:

```

var
  Count: Integer;
  Buffer: array[0..255] of Char;
begin
  // Existing code - incorrect when string = UnicodeString
  Count := Length(Buffer);
  FillChar(Buffer, Count, 0);

  // Correct for Unicode - either one will be correct
  Count := SizeOf(Buffer);           // <<-- Specify buffer size in
                                     bytes
  Count := Length(Buffer) * SizeOf(Char); // <<-- Specify buffer size in
                                     bytes
  FillChar(Buffer, Count, 0);
end;

```

Length returns the size in characters but FillChar expects Count to be in bytes. In this case, SizeOf should be used instead of Length (or Length needs to be multiplied by the size of Char).

In addition, because the default size of a Char is 2, FillChar will fill a string with bytes, not Char as previously.

Example:

```

var
  Buf: array[0..32] of Char;
begin
  FillChar(Buf, Length(Buf), #9);
end;

```

This doesn't fill the array with code point \$09 but code point \$0909. In order to get the expected result the code needs to be changed to:

```

var
  Buf: array[0..32] of Char;
begin
  ..
  StrPCopy(Buf, StringOfChar(#9, Length(Buf)));
  ..
end;

```

## USING CHARACTER LITERALS

The following code

```

if Edit1.Text[1] = #128 then

```

will recognize the Euro symbol and thus evaluate to True in most ANSI codepages. However, it will evaluate to False in Delphi 2009 because while #128 is the euro sign in most ANSI code pages, it is a control character in Unicode. In Unicode, Euro symbol is #\$20AC.

Developers should replace any characters #128-#255 with literals, when converting to Delphi 2009, since:

```

if Edit1.Text[1] = '€' then

```

will work the same as #128 in ANSI, but also work (i.e., recognize the Euro) in Delphi 2009 (where '€' is #20AC)

## CALLS TO MOVE

Calls to `Move` need to be reviewed when strings or character arrays are used. Consider the following code:

```
var
  Count: Integer;
  Buf1, Buf2: array[0..255] of Char;
begin
  // Existing code - incorrect when string = UnicodeString
  Count := Length(Buf1);
  Move(Buf1, Buf2, Count);

  // Correct for Unicode
  Count := SizeOf(Buf1);           // <<-- Specify buffer size in
                                   bytes
  Count := Length(Buf1) * SizeOf(Char); // <<-- Specify buffer size in
                                   bytes

  Move(Buf1, Buf2, Count);
end;
```

`Length` returns the size in characters but `Move` expects `Count` to be in bytes. In this case, `SizeOf` should be used instead of `Length` (or `Length` needs to be multiplied by the size of `Char`).

## READ/READBUFFER METHODS OF TSTREAM

Calls to `TStream.Read/ReadBuffer` need to be reviewed when strings or character arrays are used. Consider the following code:

```
var
  S: string;
  L: Integer;
  Stream: TStream;
  Temp: AnsiString;
begin
  // Existing code - incorrect when string = UnicodeString
  Stream.Read(L, SizeOf(Integer));
  SetLength(S, L);
  Stream.Read(Pointer(S)^, L);

  // Correct for Unicode string data
  Stream.Read(L, SizeOf(Integer));
  SetLength(S, L);
  Stream.Read(Pointer(S)^, L * SizeOf(Char)); // <<-- Specify buffer
                                             size in bytes

  // Correct for Ansi string data
  Stream.Read(L, SizeOf(Integer));
  SetLength(Temp, L); // <<-- Use temporary AnsiString
  Stream.Read(Pointer(Temp)^, L * SizeOf(AnsiChar)); // <<-- Specify
                                             buffer size in bytes
  S := Temp; // <<-- Widen string to Unicode
end;
```

Note: The solution depends on the format of the data being read. See the new `TEncoding` class described above to assist in properly encoding the text in the stream.

## WRITE/WRITEBUFFER

As with Read/ReadBuffer, calls to TStream.Write/WriteBuffer need to be reviewed when strings or character arrays are used. Consider the following code:

```
var
  S: string;
  Stream: TStream;
  Temp: AnsiString;
begin
  // Existing code - incorrect when string = UnicodeString
  Stream.Write(Pointer(S)^, Length(S));

  // Correct for Unicode data
  Stream.Write(Pointer(S)^, Length(S) * SizeOf(Char)); // <<-- Specify
  buffer size in bytes

  // Correct for Ansi data
  Temp := S; // <<-- Use temporary AnsiString
  Stream.Write(Pointer(Temp)^, Length(Temp) * SizeOf(AnsiChar)); //
  <<-- Specify buffer size in bytes
end;
```

Note: The solution depends on the format of the data being written. See the new TEncoding class described above to assist in properly encoding the text in the stream.

## LEADBYTES

Replace calls like this:

```
if Str[I] in LeadBytes then
```

with the IsLeadChar function:

```
if IsLeadChar(Str[I]) then
```

## TMEMORYSTREAM

In cases where a TMemoryStream is being used to write out a text file, it will be useful to write out a Byte Order Mark (BOM) as the first entry in the file. Here is an example of writing the BOM to the file:

```
var
  BOM: TBytes;
begin
  ...
  BOM := TEncoding.UTF8.GetPreamble;
  Write(BOM[0], Length(BOM));
```

All writing code will need to be changed to UTF8 encode the Unicode string:

```
var
  Temp: Utf8String;
begin
  ...
  Temp := Utf8Encode(Str); // <-- Str is the string being written out to
  the file.
  Write(Pointer(Temp)^, Length(Temp));
  //Write(Pointer(Str)^, Length(Str)); <-- this is the original call to
  write the string to the file.
```

## TSTRINGSTREAM

TStringStream now descends from a new type, TByteStream. TByteStream adds a property named Bytes which allows for direct access to the bytes with a TStringStream. TStringStream works as it always has, with the exception that the string it holds is a Unicode-based string.

## MULTIBYTE TOWIDECHAR

Calls to MultiByteToWideChar can simply be removed and replaced with a simple assignment. An example when using MultiByteToWideChar:

```
procedure TWideCharStrList.AddString(const S: string);
var
    Size, D: Integer;
begin
    Size := SizeOf(S);
    D := (Size + 1) * SizeOf(WideChar);
    FList[FUsed] := AllocMem(D);
    MultiByteToWideChar(0, 0, PChar(S), Size, FList[FUsed], D);
    Inc(FUsed);
end;
```

And after the change to Unicode, this call was changed to support compiling under both ANSI and Unicode:

```
procedure TWideCharStrList.AddString(const S: string);
var
    L, D: Integer;
begin
    FList[FUsed] := StrNew(PWideChar(S));
    Inc(FUsed);
end;
```

## SYSUTILS.APPENDSTR

This method is deprecated, and as such, is hard-coded to use AnsiString and no UnicodeString overload is available.

Replace calls like this:

```
AppendStr(String1, String2);
```

with code like this:

```
String1 := String1 + String2;
```

Or, better yet, use the new TStringBuilder class to concatenate strings.

## GETPROCADDRESS

Calls to GetProcAddress should always use PAnsiChar (there is no W-suffixed function in the SDK). For example:



```

procedure CallLibraryProc(const LibraryName, ProcName: string);
var
  Handle: THandle;
  RegisterProc: function: HRESULT stdcall;
begin
  Handle := LoadOleControlLibrary(LibraryName, True);
  @RegisterProc := GetProcAddress(Handle,
    PAnsiChar(AnsiString(ProcName)));
end;

```

Note: Windows.pas will provide an overloaded method that will do this conversion.

## USE OF PCHAR() CASTS TO ENABLE POINTER ARITHMETIC ON NON-CHAR BASED POINTER TYPES

In previous versions, not all typed pointers supported pointer arithmetic. Because of this, the practice of casting various non-char pointers to PChar is used to enable pointer arithmetic. For Delphi 2009, pointer arithmetic can be enabled using a compiler directive, and it is specifically enabled for the PByte type. Therefore, if you have code like the following that casts pointer data to PChar for the purpose of performing pointer arithmetic on it:

```

function TCustomVirtualStringTree.InternalData(Node: PVirtualNode):
  Pointer;
begin
  if (Node = FRoot) or (Node = nil) then
    Result := nil
  else
    Result := PChar(Node) + FInternalDataOffset;
end;

```

You should change this to use PByte rather than PChar:

```

function TCustomVirtualStringTree.InternalData(Node: PVirtualNode):
  Pointer;
begin
  if (Node = FRoot) or (Node = nil) then
    Result := nil
  else
    Result := PByte(Node) + FInternalDataOffset;
end;

```

In the above snippet, Node is not actually character data. It is being cast to a PChar merely for the purpose of using pointer arithmetic to access data that is a certain number of bytes after Node. This worked previously because `SizeOf(Char) = SizeOf(Byte)`. This is no longer true, and to ensure the code remains correct, it needs to be change to use PByte rather than PChar. Without the change, Result will end up pointing to the incorrect data.

## VARIANT OPEN ARRAY PARAMETERS

If you have code that uses TVarRec to handle variant open array parameters, you may need to adjust it to handle UnicodeString. A new type vtUnicodeString is defined for use with UnicodeStrings. The UnicodeString data is held in vUnicodeString. See the following snippet from DesignIntf.pas, showing a case where new code needed to be added to handle the UnicodeString type.

```

procedure RegisterPropertiesInCategory(const CategoryName: string;
  const Filters: array of const); overload;
var
  I: Integer;
begin
  if Assigned(RegisterPropertyInCategoryProc) then
    for I := Low(Filters) to High(Filters) do
      with Filters[I] do
        case vType of
          vtPointer:
            RegisterPropertyInCategoryProc(CategoryName, nil,
              PTypeInfo(vPointer), );
          vtClass:
            RegisterPropertyInCategoryProc(CategoryName, vClass,
              nil, );
          vtAnsiString:
            RegisterPropertyInCategoryProc(CategoryName, nil, nil,
              string(vAnsiString));
          vtUnicodeString:
            RegisterPropertyInCategoryProc(CategoryName, nil, nil,
              string(vUnicodeString));
        else
          raise Exception.CreateResFmt(@sInvalidFilter, [I,
            vType]);
        end;
      end;
end;

```

## CREATEPROCESSW

The Unicode version of `CreateProcess` (`CreateProcessW`) behaves slightly differently than the ANSI version. To quote MSDN in reference to the `lpCommandLine` parameter:

"The Unicode version of this function, `CreateProcessW`, can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a `const` variable or a literal string). If this parameter is a constant string, the function may cause an access violation."

Because of this, some existing code that calls `CreateProcess` may start giving Access Violations when compiled in Delphi 2009.

Examples of problematic code:

### PASSING IN A STRING CONSTANT

```

CreateProcess(nil, 'foo.exe', nil, nil, False, 0, nil, nil,
  StartupInfo, ProcessInfo);

```

### PASSING IN A CONSTANT EXPRESSION

```

const
  cMyExe = 'foo.exe'
begin
  CreateProcess(nil, cMyExe, nil, nil, False, 0, nil, nil,
    StartupInfo, ProcessInfo);
end;

```

## PASSING IN A STRING WITH A REFERENCE COUNT OF -1:

```
const
  cMyExe = 'foo.exe'
var
  sMyExe: string;
begin
  sMyExe := cMyExe;
  CreateProcess(nil, PChar(sMyExe), nil, nil, False, 0, nil, nil,
    StartupInfo, ProcessInfo);
end;
```

## CODE TO SEARCH FOR

The following is a list of code patterns that you might want to search for to ensure that your code is properly Unicode-enabled.

- Search for any uses of `of Char` or of `AnsiChar` to ensure that the buffers are used correctly for Unicode
- Search for instances `string[` to ensure that the characters reference are placed into `Chars` (i.e. `WideChar`).
- Check for the explicit use of `AnsiString`, `AnsiChar`, and `PAnsiChar` to see if it is still necessary and correct.
- Search for explicit use of `ShortString` to see if it is still necessary and correct
- Search for `Length` to ensure that it isn't assuming that `Length` is the same as `SizeOf`
- Search for `Copy`, `Seek`, `Pointer`, `AllocMem`, and `GetMem` to ensure that they are correctly operating on strings or array of `Chars`.

They represent code constructs that could potentially need to be changed to support the new `UnicodeString` type.

## CONCLUSION

So that sums up the types of code idioms you need to review for correctness in the Unicode world. In general, most of your code should work. Most of the warnings your code will receive can be easily fixed up. Most of the code patterns you'll need to review are generally uncommon, so it is likely that much if not all of your existing code will work just fine.

# APPENDICES

## EMBARCADERO AND PARTNER BLOG ENTRIES ABOUT UNICODE

The Unicode Shift

<http://blogs.embarcadero.com/nickhodges/2008/03/24/39041>

Unicode Character Categorization

<http://blogs.embarcadero.com/abauer/2008/01/11/38848>

Meanwhile, back at the (Unicode) ranch

<http://blogs.embarcadero.com/abauer/2008/01/28/38853>

DPL & Unicode – a toss up

<http://blogs.embarcadero.com/abauer/2008/01/09/38845>

Tiburon's LoadFromFile and SaveToFile for Unicode characters

<http://blogs.embarcadero.com/davidi/2008/07/15/38898>

Delphi 2009 - Unicode in Type Libraries

<http://chrisbensen.blogspot.com/2008/09/delphi-2009-unicode-in-type-libraries.html>

Unicode database support in Tiburon for Delphi and C++Builder 2009 -

<http://blogs.embarcadero.com/davidi/2008/07/15/38895>

Delphi 2009 and Unicode

<http://www.jacobthurman.com/?p=30>

Delphi 2009 Unicode videos from Marco Cantu

<http://blog.digivendo.com/2008/09/delphi-d2009-unicode-videos-from-marco-cantu/>

Why you need Delphi 2009

<http://compaspascal.blogspot.com/2008/09/why-you-need-delphi-2009.html>

Delphi 2009 TStringBuilder (Recap and Benchmark) –

<http://www.monien.net/blog/index.php/2008/10/delphi-2009-tstringbuilder/>

## EMBARCADERO DEVELOPER NETWORK VIDEOS ABOUT UNICODE

### GET OFF YOUR ASCII AND EXPAND YOUR BUSINESS TO GLOBAL MARKETS

Watch

<http://windemo1.codegear.com/Tiburon/LaunchReplays/ASCIInew/ASCIInew.html>

Download

<http://windemo1.codegear.com/Tiburon/LaunchReplays/ASCIInew.zip>

## **MIGRATING YOUR PROJECTS TO DELPHI 2009 – IT'S EASY!**

Watch

<http://windemo1.codegear.com/Tiburon/LaunchReplays/MigrateToDelphi2009/MigrateToDelphi2009.html>

Download – <http://windemo1.codegear.com/Tiburon/LaunchReplays/MigrateToDelphi2009.zip>

## **ADDITIONAL SOURCES OF DELPHI 2009 INFORMATION**

Delphi 2009 product page

<http://www.embarcadero.com/products/delphi/>

Data Sheet

[http://www.embarcadero.com/products/delphi/Delphi\\_Datasheet.pdf](http://www.embarcadero.com/products/delphi/Delphi_Datasheet.pdf)

Feature Matrix

[http://www.embarcadero.com/products/delphi/Delphi\\_Feature-Matrix.pdf](http://www.embarcadero.com/products/delphi/Delphi_Feature-Matrix.pdf)

Detailed Feature Guide

<http://www.embarcadero.com/products/delphi/Delphi2009-Feature-Guide.pdf>

Delphi 2009 Trial Download

<http://downloads.embarcadero.com/free/delphi>

## **ADDITIONAL SOURCES OF UNICODE INFORMATION**

The Unicode Consortium home page - <http://unicode.org/>

Unicode FAQ - <http://www.unicode.org/faq/>

The Unicode Standard - <http://www.unicode.org/standard/standard.html>



Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, CodeGear™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at [www.embarcadero.com](http://www.embarcadero.com).