White Paper

embarcadero®

The Delphi Language for Mobile Development

Marco Cantù, Delphi Product Manager

Embarcadero Technologies

April 2013

Americas 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
Level 2
100 Clarence Street
Sydney NSW 2000
Australia

INTRODUCTION

This document is an introduction to changes in the "Mobile" version of Delphi and the new Delphi ARM compiler. The focus of this document is to highlight the language changes and techniques that can be used to port existing code and to maintain backwards compatibility.

Author: Marco Cantu, Delphi Product Manager, Embarcadero Technologies
(suggest updates and integrations to marco.cantu@embarcadero.com). Written with very significant technical contributions by Allen Bauer and the help of many reviewers.

Document Revision: 1.0

1. A NEW COMPILER ARCHITECTURE

Moving Delphi to mobile ARM platforms is part of a larger evolution for the Delphi language. As such, the R&D team here at Embarcadero adopted a new architecture that will be common among all Embarcadero languages. Rather than building the compiler and all of the related tools (often indicated with the term "toolchain") in a completely proprietary and autonomous way, we decided to leverage an existing compiler and tool chain infrastructure that has broad industry support, making it faster for us to add new platforms and operating systems in the future as market demands change.

Specifically, the new generation of Delphi compilers (and also the C++Builder compilers) utilize the LLVM architecture. What is this LLVM and why does this matter? Let's take a quick look at LLVM, and return to our main topic later.

1.1: INTRODUCING LLVM

The LLVM project has its main web site with a detailed description at

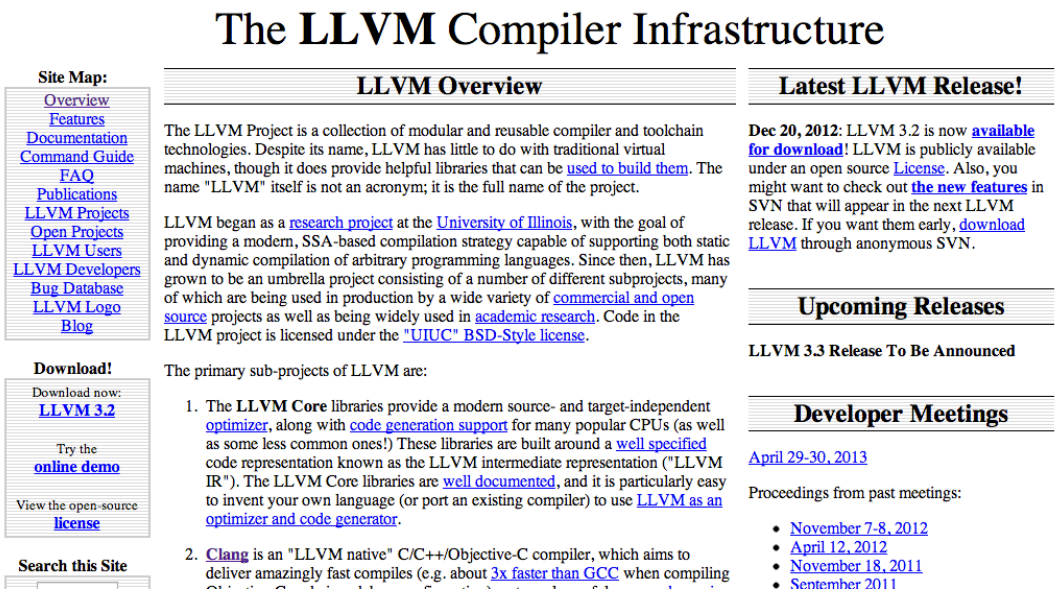
http://llvm.org

In short, LLVM is “a collection of modular and reusable compiler and tool-chain technologies”.

Despite the name (which was originally an acronym, but it is now considered as “*the full name of the project*”), LLVM has little to do with virtual machines.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of different subprojects, many of which are being used in production by a wide variety of commercial and open source projects, as well as being widely used in academic research.

Another way to appreciate LLVM is to look at the various subprojects, listed on the main web site. Specifically, the LLVM Core is built around an intermediate code representation, known as the LLVM intermediate representation (or LLVM IR). Tool builders, like Embarcadero, can create compilers to translate their own language to this intermediate representation, and they can create further tools to “compile” this representation into the native code for a CPU or into an executable intermediate representation.



The LLVM Compiler Infrastructure

Site Map:
Overview
Features
Documentation
Command Guide
FAQ
Publications
LLVM Projects
Open Projects
LLVM Users
LLVM Developers
Bug Database
LLVM Logo
Blog

Download!
Download now:
LLVM 3.2
Try the
online demo
View the open-source
license

Search this Site

LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of different subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-Style license.

The primary sub-projects of LLVM are:

1. The **LLVM Core** libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.
2. Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles (e.g. about 3x faster than GCC when compiling

Latest LLVM Release!

Dec 20, 2012: LLVM 3.2 is now **available for download**! LLVM is publicly available under an open source License. Also, you might want to check out **the new features** in SVN that will appear in the next LLVM release. If you want them early, download LLVM through anonymous SVN.

Upcoming Releases

LLVM 3.3 Release To Be Announced

Developer Meetings

April 29-30, 2013

Proceedings from past meetings:

- November 7-8, 2012
- April 12, 2012
- November 18, 2011
- September 2011

The difference between LLVM IR and other similar intermediate representations, though, is that the LLVM intention was to provide a clear separation between the front-end and the back-ends. So while allowing consumption by a virtual execution environment and JIT compiling, the LLVM IR can also be seen as an intermediate compiler representation, like a C/C++ OBJ file or a Delphi DCU (but a much more abstract one than those representations, as it does not include binary code for any target platform). The advantage of this scenario is that if you build a compiler from a

language to LLVM IR, you can use all of the available back ends, including the ARM ones. These back ends are well support by CPU vendors to deliver highly optimized executable files.

Finally, it is worth mentioning that the LLVM architecture is gaining a lot of traction both in the open source tools world and in the proprietary world. For example Apple is using LLVM (and the C/C++/ObjectiveC front-end known as Clang) in Xcode for building Mac OS X and iOS applications. Below is the LLVM home page:

1.2: DELPHI AND LLVM

Given the description above, it should be fairly obvious how Delphi's new generation compiler architecture fits in. The idea is to be able to compile Delphi source code to the LLVM IR and offer support for several CPU targets, starting with the ARM compilers for iOS and Android. This is an important consideration at a time when operating system platforms are more fragmented and are changing faster than in past years.

So when you build a Delphi iOS ARM application, you invoke a new compiler:

```
C:\Program Files\Embarcadero\RAD Studio\11.0\bin>dcciosarm
Embarcadero Delphi Next Generation for iPhone compiler
version 25.0
Copyright (c) 1983,2013 Embarcadero Technologies, Inc.
```

Of course, while the compiler is at the center of the technology, there is more to a development environment than just the compiler. For example, the Delphi IDE needs to integrate with debugging tools to let you debug ARM applications within the Delphi IDE. In addition to an IDE, Delphi comes with a compiler, a portable run-time library, a Windows-only and a platform-independent application framework… but the focus in this document is on the compiler.

An important element to consider is that the LLVM architecture and its compiler back-ends push tool builders towards specific architectures in terms of memory management and core run-time library (memory, threads, exceptions, and other low-level language elements). Notice that this push is not compulsory, as you can adopt different memory models within LLVM.

It is worth noting, though, that for the mobile platforms it has become fairly common to use LLVM (or virtual execution environments like Java and .NET) and to embrace either garbage collection or automatic reference counting (ARC). While many developers are familiar with garbage collection (for the good and the bad), ARC is much less known. You can learn more about ARC by referring to its use in ObjectiveC (within the Clang project) at:

```
http://clang.llvm.org/docs/AutomaticReferenceCounting.html
```

Let me rephrase this more clearly: While you can use LLVM with different memory management models, there is support for some advanced ones that are worth embracing, specifically for the resource-constrained mobile platforms. Also, making memory management more automatic has the effect of easing the adoption of the language by new developers.

Given that moving existing Delphi code to the mobile platform will require you to have a second look at your code, we felt it was the right time to evolve the language for this brave new world. While these changes initially apply only to the mobile platforms, Delphi XE3 already included specific instructions, library functions, and compiler directives that can get you started. This is the core topic of this paper.

1.3: Why Changes in the Delphi Language?

While we are evolving the language, our goal is to maintain a very large degree of backwards compatibility. So why are we making any changes to the existing Delphi language at all, including some that can cause issues when migrating existing code?

There are 3 main reasons, summarized here and detailed later throughout this document:

- If we just keep adding features to the language, specifically new ways of doing the same thing or new variations of existing data types, the language becomes too fat, too complicated, and harder to maintain and port to new platforms.
- Having multiple ways of doing the same basic things tends to confuse newcomers to the language quite a bit.
- Having inconsistencies in the language is a significant issue (and so are features that are slightly different than other languages, especially for new developers to the language). For example, the fact that in classic Delphi most of the data structures use 0-based indexes for data access, while strings use a 1-based index is very inconsistent. While it is true that Delphi developers can freely define array ranges (following classic Pascal use of sub-ranges), dynamic arrays are 0-based.

Given our goal of enticing new developers to the Delphi language, removing even minor unnecessary hurdles is relevant. With the business and technical benefits that Delphi for Mobile delivers, we are expecting significant new developer adoption.

1.4: What Stays the Same?

The answer is easy: almost everything! Classes and objects, methods, inheritance and polymorphism, interfaces, generic types, anonymous methods, reflection (or RTTI)…

There are also old-fashioned features that are still part of the language, like the ability to use global functions and variables, which date back to Turbo Pascal. Again, there is a very long list of features that do not change with the migration to the new compiler architecture, letting you move existing code over and allowing existing Delphi developers retain their existing knowledge.

While today if you are approaching mobile development, you would generally learn a new language, IDE, RTL, and user interface library, Delphi XE4 developers will have the luxury of using the same IDE, language, and libraries… with only limited changes. While the focus of this document is to document those changes, it is important to realize that most of the language (and most of your existing Delphi code) can be ported to the mobile platform and is already cross-platform enabled.

1.5 The Delphi Compilers in XE4

As you might have realized by reading so far, Delphi XE4 actually ships with several different compilers: one for each of the three supported desktop platforms, one for the iOS Simulator on the Mac platform, and one for the new ARM compiler. Given that this can cause a bit of confusion, here is a short summary:

- The Win32 compiler (DCC32)
- The Win64 compiler (DCC64)
- The Mac compiler (DCCOSX)
- The iOS Simulator compiler (DCCIOS32)
- The iOS ARM compiler (DCCIOSARM)

Only the last one of these five compilers is based on the LLVM tool chain, but the iOS Simulator compiler uses some of the settings of the LLVM compiler for strings and memory management, as I'll cover in detail in this paper.

2. The String Type

The management of strings of characters is the area of the language with some of the more significant changes going forward. There are several ideas behind these changes: the simplification of the traditional Delphi model (with a number of slightly different string types), some optimization requirements (partially due to the needs of the mobile platform and the model pushed by the LLVM platform), and the need to align the Delphi language with other commonly used programming languages.

Maintaining backwards compatibility all the way to Turbo Pascal and Delphi 1 causes a lot of burden and poses several challenges both to new and existing developers. Below are the actual changes, with some code samples highlighting them and some hints for both migration and cross-platform compatibility going forward.

2.1: Only One String Type

Recent versions of Delphi for Windows have seen an exponential growth of different string types. Delphi offers:

- Pascal short strings, limited to 255 one-bye characters
- Standard reference-counted copy-on-write AnsiString
- Further specific AnsiString derived types, based on the string type construction mechanism like UTF8String and RawByteString
- RTL functions to manage C-language strings (PChar)
- Unicode reference-counted copy-on-write strings, which are currently the default string type, implemented with UTF16
- COM-compatible wide strings, still UTF16 based, but not reference counted

In the new Delphi LLVM-based compiler, there is one string type, representing Unicode strings (UTF16), and mapped to the current string type in Delphi XE3 (an alias for the UnicodeString type on the Windows compiler). However, this new string type uses a different memory management model. The string type is still reference counted, but it is immutable, which means you cannot modify the string contents once it is constructed (as we'll see in more details in a specific section). In other words strings are now *Unicode-based, soon-to-become immutable, and reference-counted*.

If you need to handle one-byte strings (like ANSI or UTF8 strings), particularly when interacting with the file system or sockets, you can use dynamic arrays of bytes (something I'll focus on specifically in the section "Managing One-Byte Strings"). For now, let me show you some actual code showing the core features and offering alternative coding styles. Later I'll show some solutions for handling one-byte strings in the new compiler.

If you are still using AnsiString types, the ShortString type, the WideString type, or any other special purpose string, we strongly recommend moving all of your code to the predefined string type (well, here we are referring to code you want to migrate to the mobile platform of course, as the Delphi Windows compilers are not going to change any time soon).

Note: The need for these "special purpose" string types was mainly for interfacing with the outside world. A good example of this is the WideString type, introduced for COM support. These special types were certainly convenient to use, but they somehow "polluted" the language with types that carried slightly different semantics and behaviors for the sole purpose of interfacing with the outside world. Now that Delphi is moving to many other platforms, keeping these types around only complicates and confuses things. For instance, what does WideString mean on non-Windows platforms? It is far better to be more explicit in the types used to interface with the platform and construct them as custom types using records with methods and operators, generics, and other language features, rather than having specialized data types build into the language and the compiler.

2.2: Reference-Counted Strings

As was the case in the past, Delphi strings are reference counted. This means that if you assign a second variable to the same string or pass a string as a parameter, the reference count is increased. As soon as all references go out of scope, the string is deleted from memory.

For most developers, this implies you don't have to worry, and memory management for strings just works. If you want to understand more of the low-level implementation (*subject to change without notice*) you can read the following details; if not, skip to the next section.

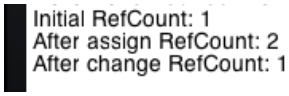
If you want to delve into the implementation details, you can query the reference count of a string with the *StringRefCount* function (added since Delphi 2009), as in this code snippet:

```
var
  str2: string;

procedure TTabbedForm.btnRefCountClick(Sender: TObject);
var
  str1: string;
begin
  str2 := 'Hello';
  Memo1.Lines.Add('Initial RefCount: ' +
    IntToStr (StringRefCount(str2)));
  str1 := str2;
  Memo1.Lines.Add('After assign RefCount: ' +
    IntToStr (StringRefCount(str2)));
  str2 [1] := '&';
  Memo1.Lines.Add('After change RefCount: ' +
    IntToStr (StringRefCount(str2)));
end;
```

If you run this, you'll see that the initial reference count for *str2* is 1, it increases to 2 after the assignment of *str1*, and gets back to 1 after the change of *str2*, given there is a copy operation of the shared string data to *str2* as the string is modified (using the copy-on-write mechanism

described in the next section). Running this code on Windows, on the simulator, or on the device produces the same output sequence (1-2-1):



Notice that when using a function or method, if you pass a string as a *const* parameter, its reference count is not modified, and the resulting code is faster (even if only by a few CPU cycles). This is done, for example, by the function returning the reference count, or it will always be increased by one by the function itself:

```
function StringRefCount(const S: UnicodeString): Longint;
```

In other words, reference counting works in a very similar way in the classic and in the new compilers, and it is quite an efficient implementation.

2.3: Copy-On-Write and Immutable Strings

Where things start to change, however, is when you modify an existing string, not by replacing it with a new value (in which case you get a brand new string) but when you modify one of its elements, as shown in this line of code (and also in the previous section, where I introduced the topic):

```
Str1 [3] := 'x';
```

All Delphi compilers use a copy-on-write semantics: If the string you modify has more than one reference, it is first copied (adjusting the reference counts of the various strings involved as required) and later modified.

Note: For those of you not familiar with the term, "copy-on-write" indicates that a string is not copied when you assign it to a new string variable (*copy*), but only when (and if) it is modified. In other word, rather than copying a string when you make the copy operation, it copies the string only at a later time, when this is effectively required. In case, there is no change, the copy operation is avoided altogether.

The new compiler does something very similar to the classic one. It implements a copy-on-write mechanism, unless there is a single reference to the string, in which case the string gets modified in place. As an example, consider the following code, which outputs the in-memory location of the actual string (using the custom *StrMemAddress* function you can find in the demo source code):

```
procedure TTabbedForm.btnCopyClick(Sender: TObject);
var
```

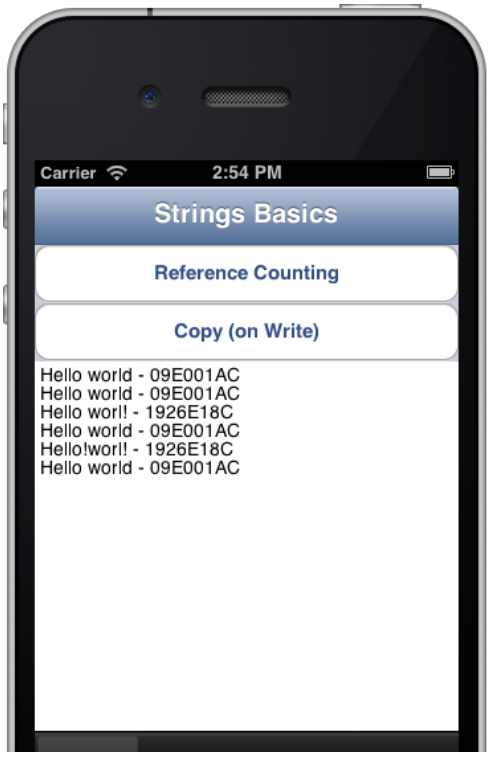
```
  str3, str4: string;
begin
  // define a string and create an alias
  str3 := Copy ('Hello world', 1);
  str4 := str3;

  // show memory location
  Memo1.Lines.Add (str3 + ' - ' + StrMemAddr (str3));
  Memo1.Lines.Add (str4 + ' - ' + StrMemAddr (str4));

  // change one (not the other)
  str3 [High(str3)] := '!';
  Memo1.Lines.Add (str3 + ' - ' + StrMemAddr (str3));
  Memo1.Lines.Add (str4 + ' - ' + StrMemAddr (str4));

  // change the first one, again
  str3 [5] := '!';
  Memo1.Lines.Add (str3 + ' - ' + StrMemAddr (str3));
  Memo1.Lines.Add (str4 + ' - ' + StrMemAddr (str4));
end;
```

The output of this method is displayed to the right (in the iOS Simulator). As you can see, one of the two strings (*str4*) is never affected, and the other is re-allocated only for the first write operation:



If you can modify a string, where is the difference compared to the classic compiler? The current implementation of strings management parallels the classic one, but it is subject to change in the future. Immutable strings offer a better memory management model. Using immutable strings you won't be able to modify one of the elements of a string directly or this operation might get slower, while string concatenation becomes faster.

So even if this operation is allowed in the current Delphi ARM compiler, the internal strings management implementation is subject to change in the future and this behavior might be disallowed.

If you want to get a warning indicating that your code might not be optimized or might even not work in future versions of the compiler, you can turn on a specific warning (already available in

Delphi XE3 for the classic compilers). The directive is *{$WARN IMMUTABLE_STRINGS ON}* and if you turn it on, and then write code like:

```
str1[3]:='w';
```

You'll see a warning with the text:

```
[dcc32 Warning]: W1068 Modifying strings in place may not be
supported in the future"
```

How does the change to immutable strings affect your code? If you are doing simple string concatenation, like the following, there is no significant drawback (nor will there be in the future, as improving performance for concatenation is actually one of goals of the potential future changes):

```
ShowMessage ('Dear ' + LastName +
  ' your total is ' + IntToStr (value));
```

Unlike other development environments, we don't expect string concatenation to become much slower or to be disallowed in any way. It is only changing individual characters of the string that might end up causing issues. The research the R&D team in currently doing in this direction is looking for optimizations to common operations like concatenating string and using the Format family of functions.

In any case, if you want to shield yourself from some of the platform specific string modification and concatenation implementation, you might want to use string construction code that is more implementation- and compiler-independent, for example using the *TStringBuilder* class.

2.4: USING THE *TSTRINGBUILDER* CLASS

A significant option going forward is to use a string constructions class, like *TStringBuilder*, for creating strings out of many sub-elements. If you are creating a string by concatenating individual characters or small strings, moving your code to *TStringBuilder* is the preferred option if you want to have good speed both when compiling for Windows and for mobile.

Here is a simple example of the same loop written using the standard string concatenation and the *TStringBuilder* class:

```
const
  MaxLoop = 2000000; // two millions

procedure TTabbedForm.btnConcatenateClick(Sender: TObject);
```

```
var
  str1, str2, strFinal: string;
  sBuilder: TStringBuilder;
  I: Integer;
  t1, t2: TStopwatch;
begin
  t1 := TStopwatch.StartNew;
  str1 := 'Hello ';
  str2 := 'World ';
  for I := 1 to MaxLoop do
    str1 := str1 + str2;
  strFinal := str1;
  t1.Stop;
  Memo2.Lines.Add('Length: ' + IntToStr (strFinal.Length));
  Memo2.Lines.Add('Concatenation: ' +
    IntToStr (t1.ElapsedMilliseconds));

  t2 := TStopwatch.StartNew;
  str1 := 'Hello ';
  str2 := 'World ';
  sBuilder := TStringBuilder.Create (str1,
    str1.Length + str2.Length * MaxLoop);
  try
    for I := 1 to MaxLoop do
      sBuilder.Append(str2);
    strFinal := sBuilder.ToString;
  finally
    sBuilder.Free;
  end;
  t2.Stop;
  Memo2.Lines.Add('Length: ' + IntToStr (strFinal.Length));
  Memo2.Lines.Add('StringBuilder: ' +
    IntToStr (t2.ElapsedMilliseconds));
end;
```

In classic versions of Delphi, the execution speed is very close (with a little advantage to the native concatenation, provided you pre-allocate the final size of the string builder as in the code above; native concatenation is between 10% and 20% faster with no pre-allocation):

```
Length: 12000006
Concatenation: 60 (msec)
Length: 12000006
StringBuilder: 61 (msec)
```

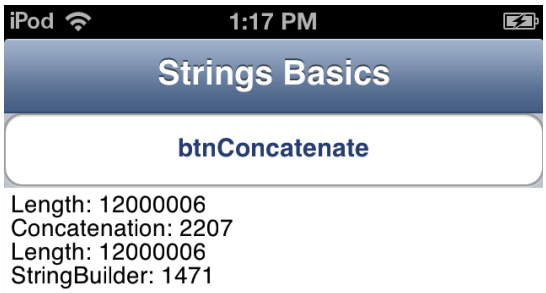
The iOS simulator is almost as fast as the Windows version of the code, given it is compiled for the Mac Intel processor, and the numbers are very similar to those above.

On the mobile platform (the physical device using the ARM compiler), one might expect plain concatenation code to slow down considerably, making *TStringBuilder* the only alternative for strings of a significant size. If this sounds familiar to some of you, it is probably because it is very similar to what happens on the Microsoft .NET platform (for example, when using C#) and other managed platforms.

While it is true that strings need to be reallocated, the memory manager is smart enough to impose a limited penalty on the execution speed (and a linear one, rather than an exponential one):

```
Length: 12000006
Concatenation: 2109 (msec)
Length: 12000006
StringBuilder: 1228 (msec)
```

Look carefully at the numbers above, *on the iOS ARM device using TStringBuilder is almost twice as fast*, and we are talking seconds here, not milliseconds! Here is the actual output of the program, captured from a physical device:



The positive thing is that the *TStringBuilder* class has been around since Delphi 2009, so you can start adding it even to applications you are maintaining in older versions of Delphi. Granted the native and mobile platforms have different processing speeds, the speed difference between my MacBook Pro and the iOS device I'm using is very large.

As another example, consider the following loop that scans all elements of a string and replaces some of them:

```
// loop on string, conditionally change some elements
for I := Low (str1) to High (str1) do
  if str1 [I] = 'a' then
    str1 [I] := 'A';
```

Considering the immutable string issue, you might expect this code to become very slow (in case of a large string, of course). As a replacement, you can use the following:

```
// loop on string, add result to string builder
sBuilder := TStringBuilder.Create;
for I := Low (str1) to High (str1) do
  if str1.Chars [I] = 'a' then
    sBuilder.Append ('A')
  else
    sBuilder.Append (str1.Chars [I]);
str1 := sBuilder.ToString;
```

As it turns out, the direct string element replacement code above is about ten times faster than the *TStringBuilder* code. Honestly, the algorithm could be significantly optimized looking for the next uppercase A and copying the string fragment before it. But if we look at the same brute-force algorithm (check each element of the string), the first, simpler code is faster in the current implementation. This might not be the same in the future, given that this is a direct change to the string structure, which an immutable strings implementation might prevent.

Note: Again, keep in mind for the future that the internal implementation of strings might change. There are languages that represent large strings in terms of a collection of string fragments, rather than a linear sequence of characters. There might be optimizations you write today that will result in slower implementations tomorrow.

2.5: Zero-Based Access

A second change is the support for 0-based strings. With this term, we refer to the ability to access elements (characters) of a string using the square brackets and an index that starts from 0, rather than from 1. Using 1-based strings is a very traditional convention of the Pascal language dating back to its early implementations. The reason for this decision, though, has little to do with readability or more natural ways of counting elements, but is the result of an implementation decision. In other words, it was the implementation (and the need to use the 0-th byte for storing the string length, as opposed to using a null terminator as in C) that surfaced as a specification.

We can argue at length what is more natural, but it is certain that most programming languages use 0-based strings and that practically all other Delphi data structures are 0-based: dynamic arrays, container classes, RTL classes like *TStringList*, VCL and FireMonkey classes with sub-elements (menu items, list box items, sub controls…).

Given this is a significant change to the language (not tied to the new compiler architecture or mobile support), which might affect a lot of code, this change is controlled by a new compiler directive already available in Delphi XE3, *$ZEROBASEDSTRINGS*. By default, this directive is off in

Delphi XE3 and it is on for the mobile compiler in XE4. However (given it is a directive), you can turn it on already in Delphi XE3 to start moving your code base to this model, or turn it off in the mobile compiler to keep your existing code working (until you migrate it to use 0-based indexing for strings).

Note that there is also a corresponding Extended Syntax compiler directive you can specify at the project options level, which will apply to all units (unless there is a different local setting). If you use it, make sure you are including in your project only units written with the corresponding string access model in mind, which is why a local setting at the unit level might help avoid a mismatch.

There are a few important related elements to notice:

- The internal structure of strings is not affected. You can mix, in a single project, units with different settings for this directive, or pass a string to a function compiled in a different way. A string is a string, no matter how the compiler interprets the *[]* expression at a given source code line.
- The classic string RTL functions will keep the existing semantics, returning and expecting 1-based positions for the string elements if *$ZEROBASEDSTRINGS* is off. However, we recommend moving away from the classic string RTL functions, which are available for backwards compatibility. The suggestion is to use the new *TStringHelper* functions (described next).
- There is a new set of functions, part of the *TStringHelper* intrinsic type helper, that we encourage Delphi developers to migrate their code to going forward. This helper class is covered in a following section and uses 0-based string indexing, regardless of the compiler (that is, it works the same on Windows, Mac, and mobile).

Consider the following trivial code snippet:

```
procedure TForm4.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'Hello';
  s := s + ' foo';
  s[2] := 'O';
  Button1.Caption := s;
end;
```

By default in Delphi XE3, the button caption becomes "*HOllo foo*". However, if you add, before the method, the directive:

```
{$ZEROBASEDSTRINGS ON}
```

The button caption will become "*HeOlo foo*". In this case, in fact, the element 2 of the string is the 3rd element, given the index becomes zero-based. What changes between the two compilers is how a line of code like the following one is evaluated:

```
aChar := aString[2];
```

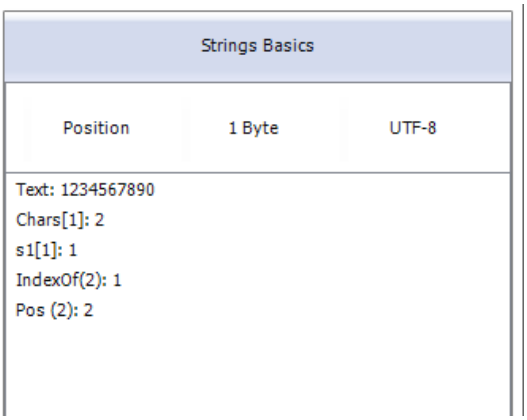
The actual behavior of the line above is controlled by the *$ZEROBASEDSTRINGS* compiler directive, which has a different default value in the two compilers. We strongly encourage you to move your code to the new model (0-based strings), possibly enabling this setting also in your Delphi XE3 Windows and Mac applications. Moving rapidly to a single string-indexing model (or more abstract coding practices as explained below) will certainly help code readability going forward.

Note: As this has caused some confusion (due to past changes in strings management in Delphi), notice that the way the compiler interprets the square brackets string elements access operator has nothing to do with the in-memory structure of a string. In other words, strings remain exactly the same internally and don't behave in any different way whether you are turning on or off the $ZEROBASEDSTRINGS directive.
It is only the way the compiler interprets the *[i]* code that changes, which means you can mix different units and functions compiled with different settings in your program. You don't pass a zero-based string to a function, you just pass a string from code or to code that uses a given compiler setting.

As this is important, let's look at the effect of the following code fragment, which highlights the potential issue:

```
var
  s1: string;
begin
  s1 := '1234567890';
  ListBox1.Items.Add('Text: ' + s1);
  ListBox1.Items.Add('Chars[1]: ' + s1.Chars[1]);
  ListBox1.Items.Add('s1[1]: ' + s1[1]);
  ListBox1.Items.Add('IndexOf(2): ' +
    IntToStr (s1.IndexOf('2')));
  ListBox1.Items.Add('Pos (2): ' +
    IntToStr (Pos ('2', s1)));
```

By default, we will get different results on the simulator and device, compared to the output on Windows and Mac. Here is a screen shot of the simulator and one from a bare-bone Windows version:



On iOS the output sequence is 2/2/1/2, while on Windows it is 2/1/1/2. Again, the only difference is in the second value, where the direct access with square brackets is used. However, by using the $ZEROBASEDSTRINGS directive, we can reverse the behavior on each platform. On Windows with *$ZEROBASEDSTRINGS ON* the output sequence is 2/2/1/2; on iOS with *$ZEROBASEDSTRINGS OFF* it is 2/1/1/2.

While in most cases this is avoidable, if you need to access a string element in a specific position, using the classic access specifier and making sure the code works regardless of the compiler and its settings, you can define a constant tied to the value of the *Low()* function, which returns the lower boundary of the string:

```
const
  thirdChar = Low(string) + 2;
```

The value of this constant will be 2 or 3, as appropriate to access the third string character. Once you have defined such a constant, you can refer to it like:

```
s1[thirdChar]
```

Code that works regardless of the compiler is an example of code that is base-string-index agnostic.

You generally face a similar task when looping the elements of a string. As an example, you can replace a classic loop with the following alternatives:

```
var
  S: string;
  I: Integer;
  ch1: Char;
begin
  // classic for
  for I := 1 to Length(S) do
    use(S[I]);

  // ZBS for
  for I := 0 to Length(S) - 1 do
    use(S[I]);

  // "agnostic" for-in loop (works since Delphi 2006)
  for ch1 in S do
    use (ch1);

  // classic for with Chars (any compiler setting, from XE3)
  for I := 0 to S.Length - 1 do
    use(S.Chars[I]);

  // flex boundaries with Low and High (works since XE3)
  for I := Low(S) to High(S) do
    use(S[I]);
```

Low and High. Low(s) returns 0 in 0-based string scenario, and 1 in 1-based. High(s) returns Length(s) - 1 in 0-based, and Length(s) in 1-based. In case of an empty string passed as a parameter, Low returns the same values, while High returns either -1 (if 0-based) or 0 (if 1-based). You can also pass the data type string to Low to determine the current settings, while passing the type to High has no meaning.

2.6: USING THE TSTRINGHELPER INTRINSIC TYPE HELPER

Another approach you can start using even in Delphi XE3 is the helper for the string data type. A new language feature in XE3, in fact, has been the ability to add custom methods not only to existing classes or records, but also to native types (which aren't records, of course). The resulting syntax is a bit unusual for Delphi developers. This is an example, based on a custom defined type:

```
type
  TIntHelper = record helper for Integer
    function ToString: string;
  end;

procedure TForm4.Button2Click(Sender: TObject);
var
  I: Integer;
```

```
begin
  I := 4;
  Button2.Caption := I.ToString;
  Caption := 400000.ToString;
end;
```

In Delphi XE3, there was this new language construct and a practical implementation, the *TStringHelper* record helper for the string type. *TStringHelper* is defined in the SysUtils unit, and offers methods like *Compare*, *Copy*, *IndexOf*, *Substring*, *Length*, *Insert*, *Join*, *Replace*, and *Split* (among many others). For example, you can write:

```
procedure TForm4.Button1Click(Sender: TObject);
var
  s1: string;
begin
  // with a variable
  s1 := 'Hello';
  if s1.Contains('ll') then
    ShowMessage (s1.Substring(1).Length.ToString);
  // with a constant
  Left := 'Hello'.Length;
  // chaining
  Caption := ClassName.Length.ToString;
end;
```

Notice that all of these methods (including the *Chars* indexed property) use the zero-based notation explained earlier, regardless of the value of the related compiler directive.

Notice in the code above the use of methods chaining, that is, the definition of methods that are applied to an object of a given type, and returning the object itself, or another object of the same type.

2.7: Managing One-Byte Strings

As I indicated in section 2.1, all 1-byte string types are not supported by the Delphi ARM compiler. This doesn't mean you cannot handle such a data structure, of course, only that you don't do it any more with a native data type. In practical terms, you cannot use data types like *AnsiString*, *AnsiChar*, or *PAnsiChar*.

As an example, consider the need to use the Unicode UTF8 format. If you have such a file, you can use a higher-level approach, based on the *TTextReader* interface and its support for encodings:

```
var
```

```
  filename: string;
  textReader: TStreamReader;
begin
  filename := TPath.GetHomePath + PathDelim
    + 'Documents' + PathDelim + 'Utf8Text.txt';

  textReader := TStreamReader.Create (
    filename, TEncoding.UTF8);
  while not textReader.EndOfStream do
    ListBox1.Items.Add (textReader.ReadLine);
```

This is rather easy to write, but hides the direct management of 1-byte UTF8 strings. The same effect can be achieved with this rather more complex low-level code, which highlights some of the internals:

```
var
  fileStream: TFileStream;
  byteArray: TArray<Byte>;
  strUni: string;
  strList: TStringList;
begin
  ...
  fileStream := TFileStream.Create (filename, fmOpenRead);
  SetLength (byteArray, fileStream.Size);
  fileStream.Read (byteArray[0], fileStream.Size);
  strUni := TEncoding.UTF8.GetString (byteArray);

  strList := TStringList.Create;
  strList.Text := strUni;
  ListBox1.Items.AddStrings (strList);
```

There might be situations (for example when making a low-level call to a function to do some direct data manipulation) where you might want to handle such a data structure in memory. In this case we strongly recommend using a dynamic array of bytes. You might even want to mimic the current behavior by wrapping such an array in a custom data structure (here is a very simple version):

```
type
  UTF8String = record
  private
    InternalData: TBytes;
  public
    class operator Implicit (s: string): UTF8String;
    class operator Implicit (us: UTF8String): string;
    class operator Add (us1, us2: UTF8String): UTF8String;
```

```
end;
```

The implementation of this record with operator overloading can be based on the *TUTF8Encoding* class, which offers ready-to-use methods to convert the array of UTF-8 bytes in a standard UTF-16 string and vice versa.

Given such a record, you can take code like the following and compile it with the new Delphi ARM compiler, which lacks the predefined *UTF8String* type:

```
var
  strU: UTF8String;
begin
  strU := 'Hello';
  strU := strU + string (' ăąāäâå');
  ShowMessage (strU);
```

3. Automatic Reference Counting

Delphi has had memory management based on reference counting since Delphi 2, when "long strings" were introduced. Strings, as detailed earlier in this paper, use reference counting and are removed from memory when all of the references to them go out of scope. Since Delphi 3, there is partial support for reference counting also for objects, as long as you use interface-type variables to refer to them. Finally, dynamic arrays also use reference counting.

So, while reference counting isn't new to the Delphi world, the new ARM compiler includes full support for the Automatic Reference Counting model for all classes and objects for the first time. Before we get to the details, let me start with an introduction to the topic.

What is Automatic Reference Counting (ARC)? As you can see in the page linked earlier in the section on LLVM, ARC is a way to manage an object's lifetime without the need to explicitly free objects you don't need any more. As the reference to the object (for example, a local variable) goes out of scope, the object will be automatically destroyed. Delphi already has reference counting support for strings and for objects referenced through interface-type variables. So, talking of objects, the closest thing to ARC in Delphi for Windows is the use of interfaces. (Notice

though, that ARC has some more flexibility in solving issues like circular references that are hard to tackle today in Delphi by using interface type variables, as I'll explain in a moment).

Differently from Garbage Collection (GC), ARC is deterministic, and objects are created and destroyed within the application flow, and not by a separate background thread. This has both advantages and disadvantages, but a discussion of GC vs. ARC is way beyond the scope of this paper.

Note: In which compilers is ARC enabled?
While the new LLVM-based compiler has ARC by default, it is important to notice that also the compiler used for the iOS Simulator (technically a compiler for the Intel CPU and the Mac OS X operating system) also enables ARC, even if it is based on the classic compiler architecture. This way the memory management in the simulator and on the device will match.

3.1: ARC Coding Style

Given that the new compiler supports reference counting, when you need to refer to a temporary object within a method, you can simplify the code significantly by ignoring memory management completely:

```
class procedure TMySimpleClass.CreateOnly;
var
  MyObj: TMySimpleClass;
begin
  MyObj := TMySimpleClass.Create;
  MyObj.DoSomething;
end;
```

In the specific test case, I've added a destructor to *TMySimpleClass*, logging to the form (in my demos, there is some more logging in the methods themselves, here omitted for simplicity). What happens here is that the destructor for the object is called as the program reaches the *end* statement, that is, when the *MyObj* variable goes out of scope.

If, for any reason, you want to stop using the object before the end of the method, you can set the variable to *nil*:

```
class procedure TMySimpleClass.SetNil;
var
  MyObj: TMySimpleClass;
begin
  MyObj := TMySimpleClass.Create;
  MyObj.DoSomething (False); // True => raise
  MyObj := nil;
```

```
    // do something else
end;
```

In this case, the object is destroyed before the end of the method, exactly as we set the variable to nil. But given there is no try-finally block, what happens in case the *DoSomething* procedure raises an exception? The *nil* assignment statement will be skipped, but as the method terminates, the object is still destroyed.

In summary, reference counting is triggered as you assign an object to a variable and when a variable goes out of scope, regardless of the fact it is a local stack-based variable, a temporary one added by the compiler, or a field of another object. The same holds for parameters: When you pass an object as parameter to a function, the object's reference count is incremented and when the function terminates and returns, it is decremented.

Note: Optimizing Parameters Passing:
Exactly as it happens for strings, you can optimize parameters passing by using the *const* modifier. An object passed as a constant, doesn't incur in the reference counting overhead. Given the same modifier can be used also on Windows, where it is useless, it is a good suggestion to update your code to use *const* object parameters, and *const* string parameters. Don't expect a very significant speed improvement, though, as the reference counting overhead is very limited.

Although you shouldn't generally use this next trick, but just allow the object's lifetime to follow the program flow, you can query the reference count of an object (only on platforms with reference counting) using the new property:

```
public
  property RefCount: Integer read FRefCount;
```

Note: The Speed of Interlocked Operations
To be thread safe, the increment and decrement operations of the reference count of an object are accomplished using "interlock" or thread-safe operations. There was a time where Intel CPUs would stall all pipelines/CPUs when executing a LOCK instruction, making them slow. With modern Intel CPUs, only the pertinent cache line will be locked. The situation is similar on ARM CPUs, used on mobile platforms. The fact that increment and decrement operations are thread safe *doesn't* mean that instances are now "thread-safe" in general. It merely means that the reference count instance variable is properly protected to ensure that all threads see the change immediately and cannot modify a "stale" value.

Note: ARC and compilers' compatibility.
If you want to write the best code in each scenario (ARC and non-ARC), for example in a library, you might want to consider using the {$IFDEF AUTOREFCOUNT} directive to discriminate among the two. This is an important directive, different from NEXTGEN, as it might as well happen in the future that ARC is implemented also on top of the classic Delphi compiler (something that already happens in the iOS Simulator).

3.2: THE FREE AND DISPOSEOF METHODS UNDER ARC

Delphi developers are used to a different coding pattern, based on the call of the *Free* method and generally protected by a try-finally block. Given most of you who used Delphi will have a lot of code based on this pattern, and might still need to write this code for Delphi for Windows compatibility, it is important to focus on the use of Free even under ARC. In short, your existing code will work, but you should keep reading to understand how.

For example, you'd generally write the code above as:

```
class procedure TMySimpleClass.TryFinally;
var
  MyObj: TMySimpleClass;
begin
  MyObj := TMySimpleClass.Create;
  try
    MyObj.DoSomething;
  finally
    MyObj.Free;
  end;
end;
```

In the classic Delphi compiler, *Free* is a method of *TObject* that checks if the current reference is not *nil* and if this is the case calls the *Destroy* destructor, which removes the objects from memory after executing the proper destructor code.

In the new generation compiler, instead, the call to *Free* is "replaced" with the assignment of the variable to *nil*. In case this was the last reference to the object, this is still removed from memory after calling its destructor. If there are other standing references, nothing happens (but a decrease in the reference count).

Similarly, a call like:

```
FreeAndNil (MyObj);
```

sets the object to *nil*, and again triggers the object destruction only if there are no other variables referring to it. In most cases, this is correct, as you don't want to destroy objects used in another part of a program. There are scenarios, though, when you really want to execute the destructor code (maybe closing a file or a database connection) right away, regardless of the fact that there might be other pending references.

In other words, while they are often useless, calls to *Free* or *FreeAndNil* are generally completely harmless and you can keep them in your Delphi programs for the mobile platform. There are some limited scenarios, though, in which you need a different approach.

To allow the developer to force the execution of the destructor (without releasing the actual object from memory), the new compiler introduces a *dispose* pattern. If you call:

```
MyObject.DisposeOf;
```

there is a forced execution of the destructor code, even if there are pending references. At this point the object is placed in a special state, so that the destructor won't be called again in case of further disposal operations or when the reference counting reaches zero and memory is actually released. This "disposed" state (or "*zombie*" state) is quite significant that you can query an object for it using the *Disposed* property.

Note: DisposeOf on Win32
The new method is available also on the classic compiler for Windows and Mac, but in this case a call to *DisposeOf* is remapped to a call to the *Free* method. In other words, the new method makes no difference, but has been introduced to improve source code compatibility among different platforms.

Why is this dispose pattern required? Consider the case of a collection of elements or the components owned by another component. A common usage pattern is to "destroy" a particular item in the collection in order to both clean up the item itself and remove it from the collection. Another common scenario is to destroy the collection or component owner and dispose of all the owned elements or components. In this case, if there are still pending references to the owned objects, it will be likely to force their destruction, or at least the execution of their destructor code.

Using these destroyed instances after they have been disposed, might result in an error, but that's not much different than under the classic Delphi compilers, where in freeing an instance and using other references, this will also result in an error. What is significantly different is that under the classic Delphi compilers, when you have two references to an object and free the object using one of them, there is no way to know if the other reference is still valid. Instead, using *DisposeOf*, you can query an object about its status:

```
myObj := TMyClass.Create; // an object reference
myObj.SomeData := 10;

myObj2 := myObj; // another reference to the same object

myobj.DisposeOf; // force cleanup

if myobj2.Disposed then // check statusof other reference
```

```
    Button1.Text := 'disposed';
```

I've mentioned earlier that the classic try-finally blocks used on the classic Delphi compilers still work fine under the new compiler, even if they are not required. In specific cases in which you want to force the execution of the destructor code as soon as possible and regardless of other references, you might want to use the dispose pattern instead (unless you want to recompile the code for earlier versions of Delphi, of course):

```
var
  MyObj: TMySimpleClass;
begin
  MyObj := TMySimpleClass.Create;
  try
    MyObj.DoSomething;
  finally
    MyObj.DisposeOf;
  end;
end;
```

In the classic compilers, the effect remains the same, as *DisposeOf* calls *Free*. On ARC, instead, the code executes the destructor when expected (that is, as the same time of the classic compiler) but the memory is managed by the ARC mechanism. This is nice, but you cannot use this same code for compatibility with older versions of Delphi. For that purpose, you might want to call *FreeAndNil*, redefining this procedure as a call to either *Free* or *DisposeOf*. Or just stick with a standard call to *Free*, which does the trick most of the time.

Note: Storage for the Disposed Flag
The actual storage for the Disposed "flag", rather than being an extra field, is a bit in the *FRefCount* field. Given that a second bit is used for related destruction-tracking purposes, the reference count has a theoretical limit of 2^30, which could hardly be seen as a real limit.

One way to look at the difference between *Free* and *DisposeOf* is to consider the intent of the two operations under ARC (as opposed to what happens under the classic Delphi compilers). When using *Free*, the intent is that the specific reference should simply "detach" from the instance. It does not imply any kind of disposal or memory de-allocation. It's merely that that block of code doesn't need that reference anymore. Usually this will simply happen upon exiting the scope, but you can *force* it by calling *Free* explicitly.

By contrast, *DisposeOf* is the programmer's way of explicitly telling the instance that it needs to "*clean itself up.*" *DisposeOf* never necessarily implies memory de-allocation; it merely does an explicit "*clean-up*" of the instance (executing any specific destructor code). The instance still relies on the normal reference count semantics to eventually de-allocate the memory it uses.

In other words, under ARC *Free* is an "instance reference centric" operation, whereas *DisposeOf* is an "instance centric" operation. *Free* means, "I don't care what happens to the instance, I just don't need it anymore." *DisposeOf* means, "I need this instance to internally clean itself up since it may be holding a non-memory resource that needs to be released" (like a file handle, database handle, a socket, and so on).

Another use for *DisposeOf* is to explicitly trigger proper cleanup and de-allocation for complicated reference cycles. While the use of *[Weak]* references (described in the next section) makes things more clear and explicit, there may be situations where an explicit trigger or notification is needed to "tell" other instances to drop their reference.

Note: Mixing Pointers and Objects
For example, if you assign an object to a pointer, reuse the object variable for a different object, and later assign the pointer back to the object variable, the object won't be there any more: as the object's reference counts gets to zero (given the pointer doesn't count as a reference and doesn't increase the reference count) this is destroyed. As an example see the changes done to the *TStringList.ExchangeItems* method of the RTL, which in the past used a pointer to keep a temporary "reference" to the object being moved to a new location.

3.3: Weak References

Another very important concept for ARC is the role of weak references. Suppose that two objects refer to each other using one of their fields, and an external variable refers to the first. The reference count of the first object will be 2 (the external variable, and the field of the second object): while the reference count of the second object is 1 (the field of the first object). Now, as the external variable goes out of scope, the two objects' reference count remains 1, and they'll remain in memory indefinitely.

To solve this type of situation, you should break the circular references, something far from simple, given that you don't know when to perform this operation (it should be performed when the last external reference goes out of scope, a fact of which the objects have no knowledge). The solution to this situation, and many similar scenarios, is to use a weak reference.

A weak reference is a reference to an object that doesn't increase its reference count. Given the previous scenario, if the reference from the second object back to the first one is weak, as the external variable goes out of scope, both objects will be destroyed.

Let's look at this simple situation in code:

```
type
  TMyComplexClass = class;

  TMySimpleClass = class
```

```
  private
    [Weak] FOwnedBy: TMyComplexClass;
  public
    constructor Create();
    destructor Destroy (); override;
    procedure DoSomething(bRaise: Boolean = False);
  end;

  TMyComplexClass = class
  private
    fSimple: TMySimpleClass;
  public
    constructor Create();
    destructor Destroy (); override;
    class procedure CreateOnly;
  end;
```

Here the constructor of the "complex" class creates an object of the other class:

```
constructor TMyComplexClass.Create;
begin
  inherited Create;
  FSimple := TMySimpleClass.Create;
  FSimple.FOwnedBy := self;
end;
```

Remember that the *FOwnedBy* field is a weak reference, so it doesn't increase the reference count of the object it refers to, in this case the current object (*self*). Given this class structure, we can write:

```
class procedure TMyComplexClass.CreateOnly;
var
  MyComplex: TMyComplexClass;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.fSimple.DoSomething;
end;
```

This will cause no memory leak, given the weak reference is properly used.

As a further example of the use of weak references, notice this code snippet in the Delphi RTL, part of the *TComponent* class declaration:

```
type
  TComponent = class(TPersistent, IInterface,
```

```
    IInterfaceComponentReference)
  private
    [Weak] FOwner: TComponent;
```

Notice that if you use the weak attribute in code compiled with the classic Delphi compiler, this will be ignored. However, you have to make sure that you add the proper code in the destructor of an "owner" object to also free the "owned" object. As we have seen, calling Free is allowed, although the effect is different in the Delphi classic and ARM compilers, the behavior will be correct in both in most circumstances.

When you are using a weak reference, as in the example above, you shouldn't test the weak reference itself to see if it *nil*. What you can do is to assign the weak reference to a strong reference first (which introduces some checks behind the scenes), and then check the strong reference value. As an example, given the *FOwner* weak reference above, you could write:

```
var
  TheOwner: TComponent; // strong reference alias
begin
  TheOwner := FOwner;
  if TheOwner <> nil then
    TheOwner.ClassName; // safe to use
end;
```

3.4 Checking for Memory Problems

Given how memory management works under the Delphi ARM compiler for iOS, it is worth considering some of the options you have in making sure everything is under control. Before we proceed, it is important to notice that on non-Windows platforms Delphi doesn't use the FastMM memory manager, so setting the *ReportMemoryLeaksOnShutdown* global flag to check for memory leaks when the program closes is useless. On the OS X and iOS platforms, in fact, Delphi called directly the *malloc* and *free* functions of the native *libc* library.

On the iOS platform a very nice solution is to use Apple's Instruments tool, which is a complete tracking system monitoring all aspects of your applications running on a physical device. You can find a very detailed video by Daniel Magin and Daniel Wolf covering this tool from a Delphi perspective at:

```
http://www.danielmagin.de/blog/index.php/2013/03/apple-instruments-
and-delphi-for-ios-movie/
```

Given that one of the potential issues causing memory leaks are circular references among objects, there is a little function that might help you figure out how your application behaves. This is part of the Classes unit and is called *CheckForCycles*:

```
procedure CheckForCycles(const Obj: TObject; const
  PostFoundCycle: TPostFoundCycleProc); overload;
procedure CheckForCycles(const Intf: IInterface; const
  PostFoundCycle: TPostFoundCycleProc); overload;
```

This is not a function you'd generally use in your final code, but only for testing purposes during development and debugging. The second parameter of the procedure is an anonymous method receiving as parameter the object's class, its memory address, and a stack with the objects in the cycle. This is a rather basic example of its usage, based on the class described earlier after removing the weak reference (with the weak reference, there is no cycle):

```
var
  MyComplex: TMyComplexClass;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.fSimple.DoSomething;
  CheckForCycles (myComplex,
    procedure (const ClassName: string; Reference: IntPtr;
      const Stack: TStack<IntPtr>)
    begin
      Log('Object ' + IntToHex (Reference, 8) +
        ' of class ' + ClassName + ' has a cycle');
    end)
```

3.5: THE UNSAFE ATTRIBUTE

There are some very specific circumstances (for example during the creation of an object) in which a function might return an object with a reference count set to zero. In this case, in order to avoid the compiler deleting the object right away (before it has a chance to be assigned to a variable, which would increase its reference count to 1), we have to mark the object as "unsafe" (because its reference count has to be temporarily ignored to make the code "safe").

This behavior is accomplished by using a new specific attribute, *[Unsafe]*, a feature you should need only in very specific circumstances:

```
var
  [Unsafe] Obj1:  TObject;
```

```
[Result: Unsafe]
function GetObject: TObject;
```

In the System unit, a corresponding directive, *unsafe*, replaces the attribute only because you cannot use the attribute before it is defined in the same unit. An example is the low-level *InitInstance* class function of the *TObject* class, used to allocate the memory for an object, which is declared as:

```
type
  TObject = class
public
  constructor Create;
  procedure Free;
  class function InitInstance(Instance: Pointer):
    TObject {$IFDEF AUTOREFCOUNT}unsafe{$ENDIF};
```

Usage of the *unsafe* directive (as shown in the above code) is limited to the System unit.

3.6 Low-Level Reference Counting Operations

Although in most scenarios you should just adapt your code to the use of reference counting, potentially adding weak references and the unsafe attribute as needed, there are circumstances in which you have direct memory allocation for the objects, managed in custom ways. In similar scenarios, the object won't be properly managed in terms of reference counting, and you might end up with an object you need to stay in memory while it has no actual references. In such a (seldom to rare) case, you can force a change in reference counting by calling two public virtual methods of the *TObject* class:

```
function __ObjAddRef: Integer; virtual;
function __ObjRelease: Integer; virtual;
```

Both methods return the reference count after the operation.

Note: Reference Counting Speed
The two functions above implement the core of the reference counting code in Delphi, triggered automatically by the compiler as needed. When you assign an object to a new variable, the overhead is a single call to a function in the virtual method table, which in turn increments a field of the object itself. This is much faster than alternative implementations, including Apple's current implementation of ARC for ObjectiveC.

A potential scenario for using these methods is when you have a block of memory (possibly allocated by some external API) that you want to treat as, or cast to, an object type. Another example, taken from the RTL, is when you copy the data of an object using a pointer:

```
class function TInterlocked.CompareExchange(
  var Target: TObject; Value, Comparand: TObject): TObject;
begin
{$IFDEF AUTOREFCOUNT}
  if Value <>nil then
    Value.__ObjAddRef;
{$ENDIF AUTOREFCOUNT}

  Result := TObject(CompareExchange(
  Pointer(Target), Pointer(Value), Pointer(Comparand)));

{$IFDEF AUTOREFCOUNT}
  if (Value <> nil) and
       (Pointer(Result) <> Pointer(Comparand)) then
    Value.__ObjRelease;
{$ENDIF AUTOREFCOUNT}
end;
```

3.7 Summary of TObject's Construction and Destruction Methods under ARC

This is a summary of the methods of the *TObject* class related with creation and destruction, both new and classic (I've omitted some conditionally defined directives):

```
type
  TObject = class
  public
    constructor Create;
    procedure Free;
    procedure DisposeOf;
    destructor Destroy; virtual; // protected on ARC
    property Disposed: Boolean read GetDisposed;
    property RefCount: Integer read FRefCount; // only if ARC
    // low level operations
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    classfunction NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    function __ObjAddRef: Integer; virtual;
    function __ObjRelease: Integer; virtual;
```

Now we could certainly delve into the details of the implementation of each method for the different platforms, with or without reference counting, but it would be too advanced a topic for this new language features introduction.

3.8: Bonus Feature: Operator Overloading for Classes

There is a very interesting side effect of using ARC for memory management: the compiler can handle the lifetime of temporary objects returned by functions. One specific case is that of temporary objects returned by operators. In fact, a brand new feature of the new Delphi compiler is the ability to define operators for classes, with the same syntax and model that has been available for records since Delphi 2006.

Note: Which Compilers Support Operators Overloading for Classes?
This language feature works on the iOS ARM compiler, but also on the iOS Simulator on Mac. Of course, you cannot compile this code for Windows or Mac with the classic compilers. While it won't be terribly difficult to add it, operators end up creating many temporary variables, and without an automatic memory management mechanism (like ARC or garbage collection) this approach really makes no sense.

As an example, consider the following simple class:

```
type
  TNumber = class
  private
    FValue: Integer;
    procedure SetValue(const Value: Integer);
  public
    property Value: Integer read FValue write SetValue;
    classoperator Add (a, b: TNumber): TNumber;
    class operator Implicit (n: TNumber): Integer;
  end;

class operator TNumber.Add(a, b: TNumber): TNumber;
begin
  Result := TNumber.Create;
  Result.Value := a.Value + b.Value;
end;

class operator TNumber.Implicit (n: TNumber): Integer;
begin
  Result := n.Value;
end;
```

Given this code, we can use the class as follows:

```
procedure TForm3.Button1Click(Sender: TObject);
var
  a, b, c: TNumber;
begin
  a := TNumber.Create;
```

```
  a.Value := 10;

  b := TNumber.Create;
  b.Value := 20;

  c := a + b;

  ShowMessage (IntToStr (c));
end;
```

3.9: Mixing Interfaces and Classes

In the past, given that interface variable and standard object variables used different memory management models, it was generally suggested to avoid mixing the two approaches (like using an interface and an object variable or parameter to refer to the same object in memory).

With the new ARM compilers with ARC, the reference counting between object and interface variables is unified, so you can mix the two easily. This makes using interfaces more powerful and flexible on Delphi ARC platforms than it is on Delphi non-ARC platforms.

4. Other Language Changes

Besides string type changes and objects memory management, there are other current or expected changes in the new Delphi ARM compiler that you can easily start to adopt:

- Sooner or later, the *with* statement is going to be deprecated and removed from the Delphi language. You can easily start removing it now from your code, and most Delphi developer will agree this is a good idea anyway, given some of the hidden pitfalls of this keyword.
- Reduce or remove pointer usage, given direct usage of pointer is being discouraged as we move towards some automatic memory management. Using generic container classes, instead of *TList* (internally based on pointers) is a good example of the migration that the Delphi RTL library is undergoing, and we suggest that Delphi developers perform similar conversions in their code as well.
- Remove assembly code, as this isn't portable to the new compiler and isn't applicable to ARM CPUs.
- The volatile attribute is used to mark fields subject to change by different threads, so that the code generation won't optimize copying the value in a register or another temporary memory location not shared by the separate thread.

5. RTL CONSIDERATIONS

The last part of this paper (after the introduction to LLVM and the description of compiler changes) is focused on the Run-Time Library (RTL) and its changes to support the mobile platform. This last section won't be as extensive and detailed as the previous ones, but will simply outline some of the issues you might encounter when migrating or writing new Delphi applications for the mobile platform, in terms of the Run-Time Library (not the user interface or the database or the native device sensors).

In other words, in this short section, there is a list of generic suggestions for supporting different operating systems, information related to file access, support for packages and libraries (of lack thereof).

5.1: RTL AND CROSS PLATFORM

Just a few suggestions on several partially related topics:

- Avoid direct API calls whenever possible, that is, when a higher-level component or class is available. This makes the code easier to port to different platforms (today or in the future).
- Prefer cross-platform units. For example, for files management it is suggested to use IOUtils unit records and their class methods, rather than old style file management Pascal routines. Some specific examples of the IOUtils unit are covered in the next section.
- Avoid any Windows-isms and Windows specific API if you want to make your code portable to either Mac or Mobile. A good example is not using MSXML but favor the Delphi based ADOM XML processing engine, which works also on mobile. Another example is avoiding specific socket and Web libraries, favoring Indy. Needless to say, code that uses COM, ActiveX, ADO, BDE and other Windows-specific Delphi technologies won't work on mobile (or on the Mac, for the record).
- Use the generic container classes defined in the unit *Generics.Collections*. The old-style Contnrs unit is not available on the mobile platform, because it is based on a pointer list (the non-generic *TList* class) and won't work properly with the ARC memory management model.
- As an aside, while using TStringList with objects attached to each element is supported, I strongly recommend using a generic dictionary with a string as key, like in the type definition *TDictionary<string,TMyClass>*. Not only is this much cleaner, but in most cases

it is also much faster, because the dictionary uses a hash table. This is detailed in one of the following sub-sections.

- Avoid any pointer-based structure and method, unless you are calling native APIs requiring it.

This list could certainly grow much longer. Here I want to offer only these few bullets and add to those a couple of specific details, on file access and libraries.

5.2: File Access

Since several versions of Delphi, the classic Pascal file access routines (like FindFirst and FindNext to search in a folder) have been replaced by a set of higher-level records, grouped in the *IOUtils* (Input/Output utilities) unit. Using IOUtils is the recommended approach to make your code cross-platform.

This unit has three records mostly defining class methods, which are compatible with corresponding .NET classes:

- *TDirectory* matches System.IO.Directory
- *TPath* matches System.IO.Path
- *TFile* matches System.IO.File

While it is quite obvious that *TDirectory* is for browsing a folder and finding its files and sub-folders, it might not be so clear what is the difference between a *TPath* and *TFile*. *TPath* is used for manipulating file names and directory names, with methods for extracting the drive, file name with no path, extension and the like, but also for manipulating UNC paths. The *TFile* record, instead, lets you check the file time stamps and attributes, but also manipulate a file, writing to it or copying it.

For example, when you deploy a file along with your application on the mobile platform, you can access the application "documents" folder in the same way you access the user's document on Windows:

```
var
  myfilename: string;
begin
  myfilename := TPath.GetHomePath + PathDelim
    + 'Documents' + PathDelim + 'thefile.txt';
  if TFile.Exists (myfilename) then
    ...
```

One of the available features is searching folders and files. The following code snippet reads the folders under a given initial folder (*BaseFolder*), going only one level down, and reads the files in the subfolders:

```
var
  pathList, filesList: TStringDynArray;
  strPath, strFile: string;
begin
  if TDirectory.Exists (BaseFolder) then
  begin
    ListBox1.Items.Clear;
    ListBox1.Items.Add ('Searching in ' + BaseFolder);
    pathList := TDirectory.GetDirectories(BaseFolder,
      TSearchOption.soTopDirectoryOnly, nil);
    for strPath in pathList do
    begin
      ListBox1.Items.Add (strPath);
      filesList := TDirectory.GetFiles (strPath, '*');
      for strFile in filesList do
         ListBox1.Items.Add ('- ' + strFile);
    end;
    ListBox1.Items.Add ('Searching done in ' + BaseFolder);
  end
  else
    ListBox1.Items.Add ('No folder in ' + BaseFolder);
end;
```

5.3 Replacing String Lists With a Generic Dictionary

Over the years many Delphi developers, myself included, have overused the *TStringList* class. Not only you can use it for a plain list of strings and for a list of name/value pairs, but you can also use it to have a list objects associated with strings and search these objects.

Given Delphi fully supports generics, the role of this Swiss-army knife kind of tools can be better replaced by specific and focused container classes. For example, a generic Dictionary with a string key and an object-value will generally be better that a string list on two counts: cleaner and safer code, as there will be fewer type casts involved, and faster execution, given that dictionaries use hash tables.

To demonstrate these differences consider the following example, which has two identical lists, declared as:

```
  private
```

```
sList: TStringList;
sDict: TDictionary<string,TMyObject>;
```

The lists a filled with random but identical entries in a cycle, which repeats this code:

```
sList.AddObject (aName, anObject);
sDict.Add (aName, anObject);
```

The speed test is done using two methods that retrieve each element of the list doing a search by name on each of them. Both methods scan the string list for the values, but the first locates the objects in the string list, while the second uses the dictionary. Notice that in the first case you need an *as* type cast to get back the given type, while the dictionary is tied to that class already. Here are the main loops of the two methods:

```
theTotal := 0;
for I := 0 to sList.Count - 1  do
begin
  aName := sList[I];
  // now search for it
  anIndex := sList.IndexOf (aName);
  // get the object
  anObject := sList.Objects [anIndex] as TMyObject;
  Inc (theTotal, anObject.Value);
end;

theTotal := 0;
for I := 0 to sList.Count - 1  do
begin
  aName := sList[I];
  // get the object
  anObject := sDict.Items [aName];
  Inc (theTotal, anObject.Value);
end;
```

How much time does it take to search in the sorted string list (which does a binary search) compared to the hashed keys of the dictionary? Not surprisingly the dictionary is faster, here are numbers in milliseconds for a test on the Windows platform (but similar differences show up on all platforms):

```
StringList: 2839
Dictionary: 686
```

The result is the same, given the initial values where identical, but the time is quite different, with the dictionary taking about ***one fourth of the time*** (with a test done using a million entries).

Of course, this example and this short section just scratch the tip of the iceberg in illustrating the power of generic dictionaries and some of the more recent data structures added to the Delphi RTL, after the inclusion of generics at the compiler level. Again, these new structure are a good choice on all platforms, but become even more relevant on iOS, due to the changes in the memory management model.

5.4: Libraries and Packages

A powerful Delphi feature is the use of run-time packages to deploy applications in a more modular way. Packages are "special purpose" dynamic link libraries, DLLs on Windows or *dylib* on Apple platforms. On Windows and on OS X (with some differences) you can deploy run-time packages in common folders, so that they are shared among multiple applications, or in the specific application folder to avoid potential conflicts among those different programs.

On the iOS platform, instead, neither scenario is allowed. You cannot deploy shared libraries to a physical iOS device (while you can use them in the iOS simulator), as this is something only Apple can do at the operating system level. You cannot add dynamic libraries to an application, as the only executable file can be the main program.

This limitation is not tied to run-time packages, but it is more generic in nature. For example, standard Delphi libraries such as the midas.dll or dbExpress drivers are statically linked to the application executable file. The same happens even for the InterBase client library. In fact, the compiler has a way to recognize references to static libraries and link those in the final executable, a technical issue I don't really want to broach in this paper.

6. Conclusion

With the release of the first Delphi compiler for ARM, based on the LLVM architecture, the Delphi language is undergoing a major transition, as you have seen in this paper. While effort was made to maintain backwards compatibility, we expect developers to fully embrace the new features moving forward.

The language changes described in this paper, and particularly ARC support, are going to shape Delphi going forward. These changes are partially driven by the new platform, and partially meant to clean up some rough corners of the language and add new, and in many cases user-friendly, features to Delphi.

About The Author

Marco Cantù recently joined Embarcadero Technologies as Delphi Product Manager. He was the author of the best-selling Mastering Delphi series and in the recent years he has self-published *Handbooks* on the several versions of Delphi (from 2007 to XE).

Marco is a frequent conference speaker, author of countless articles on Delphi, and used to teach advanced Delphi classes (including Web development with Delphi) at companies worldwide. You can read Marco's blog at *http://blog.marcocantu.com*, follow him on Twitter as *@marcocantu*, and contact him on *marco.cantu@embarcadero.com*.

About Embarcadero Technologies

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. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.