White Paper

embarcadero®

モバイル開発のための Delphi 言語

Marco Cantu（Delphi プロダクト マネージャ）

エンバカデロ・テクノロジーズ

2013 年 4 月

エンバカデロ・テクノロジーズ

〒102 – 0072 東京都千代田区飯田橋 4-7-1 ロックビレイビル 8F

TEL 03-4577- 4530 FAX 03-6843-0961

はじめに

このドキュメントでは、Delphi の "モバイル" 版と新しい Delphi ARM コンパイラにおける変更点を紹介します。既存コードの移植や下位互換性の維持に利用できる言語上の変更点と手法を明らかにすることに重点を置いています。

著者： Embarcadero Technologies 社 Delphi プロダクト マネージャ Marco Cantu（更新や統合の提案は marco.cantu@embarcadero.com まで）。執筆にあたっては、Allen Bauer 氏の技術面での多大なご助力と多くの査読者諸氏のお力添えをいただきました。

ドキュメント改訂番号： 1.0

1. 新しいコンパイラ アーキテクチャ

Delphi は、言語の大規模な進化の一環として、モバイル ARM プラットフォームに進出しつつあります。そのため、エンバカデロの研究開発チームでは、エンバカデロのすべての言語で共通に使用されることになる新しいアーキテクチャを採用しました。コンパイラとそのすべての関連ツール（"ツール チェーン" という用語で示されることが多い）を完全に独自に開発するのではなく、業界で広くサポートされている既存のコンパイラおよびツール チェーン インフラストラクチャを利用することにし、将来、市場需要の変化に合わせて新しいプラットフォームやオペレーティング システムをより迅速に追加できるようにしました。

特に、新世代の Delphi コンパイラ（また C++Builder コンパイラ）では LLVM アーキテクチャを利用しています。この LLVM とは何なのでしょうか、そしてなぜこれが重要なのでしょうか。それでは、まず LLVM を概観し、その後で本題に戻りましょう。

1.1 LLVM の紹介

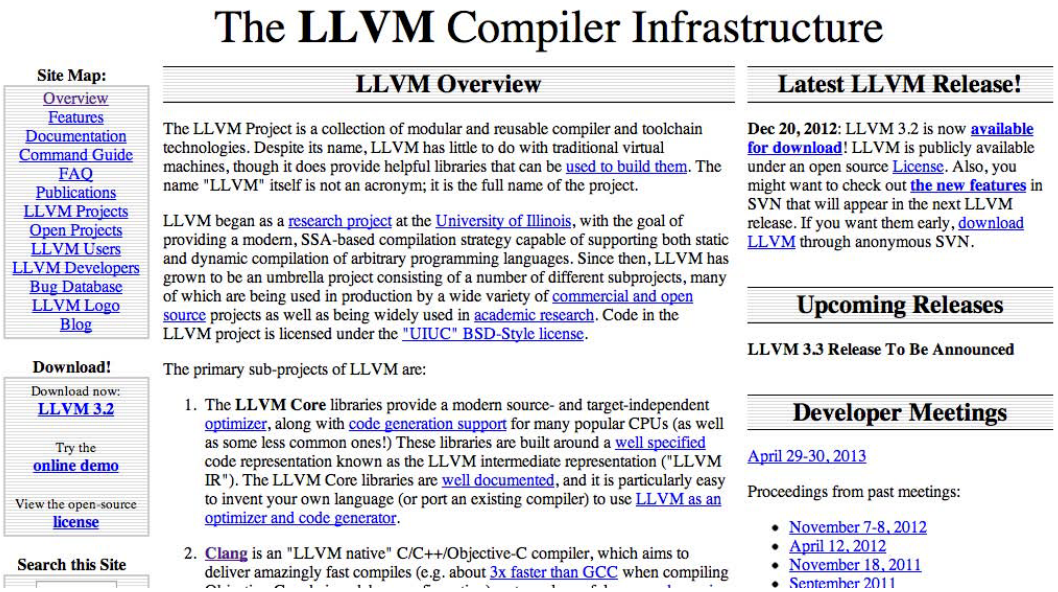
LLVM プロジェクトには、詳しい説明が掲載されている主要な Web サイト（http://llvm.org）があります。

手短に言えば、LLVM は "モジュール性の高い再利用可能なコンパイラおよびツール チェーン技術の集合体" です。

名前とは裏腹に、LLVM は仮想マシンとはほとんど関係がありません（この名前はもともと頭字語でしたが、今では **"プロジェクトの正式名称"** と見なされています）。

LLVM は、任意のプログラミング言語の静的コンパイルと動的コンパイルの両方をサポートできる SSA ベースの新しいコンパイル戦略を提供することを目的に、イリノイ大学で研究プロジェクトとして始まりました。それ以後、LLVM は発展し、多数の異なるサブプロジェクトから成る包括的なプロジェクトになりました。これらのサブプロジェクトの多くは、さまざまな商用プロジェクトおよびオープン ソース プロジェクトで本格使用されつつあるほか、学術研究でも広く使用されています。

LLVM をよく理解するには、その主要な Web サイトに掲載されているさまざまなサブプロジェクトを調べてみる方法もあります。特に、LLVM Core は LLVM 中間表現（略して LLVM IR）と呼ばれる中間コード表現に基づいて構築されています。エンバカデロのようなツール開発ベンダでは、独自の言語をこの中間表現に翻訳するコンパイラを作成でき、この表現を CPU のネイティブ コードや実行可能な中間表現に "コンパイル" するツールをさらに作成することができます。



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

同じような中間表現は他にもありますが、それらと LLVM IR の違いは、LLVM には、フロントエンドとバックエンドを明確に分離しようとする意図があったことです。そのため、LLVM IR は、仮想実行環境や JIT コンパイルで使用できる一方、C/C++ の OBJ ファイルや Delphi の DCU のようなコンパイラ中間表現と見ることもできます（ただし、どのようなターゲット プラットフォーム向けのバイナリ コードも含んでいないので、それらの表現よりはるかに抽象的な表現になります）。このシナリオの利点は、ある言語から LLVM IR へのコンパイラを作成すれば、ARM バックエンドを含む使用可能なすべてのバックエンドを使用できることです。これらのバックエンドは CPU ベンダで完全にサポートされており、非常に最適化された実行可能ファイルを生成します。

最後に、LLVM アーキテクチャがオープン ソース ツールの分野でも独自開発の分野でも非常に勢いを増しつつあることに触れておきましょう。たとえば、Apple 社では Mac OS X アプリケーションや iOS アプリケーションを構築するための Xcode で LLVM（および Clang と呼ばれる C/C++/Objective-C フロントエンド）を使用しています。以下は LLVM のホーム ページです。

1.2 DELPHI と LLVM

以上のことを考えると、Delphi の次世代コンパイラ アーキテクチャがいかに合っているかはほとんど明らかでしょう。Delphi ソース コードを LLVM IR にコンパイルして、iOS や Android 向けの ARM コンパイラを始めとして、いくつかの CPU ターゲットに対応することができるという考え方です。これは、オペレーティング システム プラットフォームがこれまでよりも細分化され急速に変化しつつある状況では重要な考えです。

ということで、Delphi iOS ARM アプリケーションを構築する際は、以下のように、新しいコンパイラを起動します。

```
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.
```

もちろん、コンパイラが技術の中心にはあるものの、開発環境としてはコンパイラだけでは不十分です。たとえば、Delphi IDE では、デバッグ ツールと統合して ARM アプリケーションを Delphi IDE 内でデバッグできるようにする必要があります。IDE に加えて、Delphi には、コンパイラ、移植性の高いランタイム ライブラリ、Windows 専用のアプリケーション フレームワーク、プラットフォームに依存しないアプリケーション フレームワークなども付属していますが、このドキュメントではコンパイラを中心に取り上げます。

考慮すべき重要な要因の 1 つは、LLVM アーキテクチャとそのコンパイラ バックエンドのために、ツール開発ベンダがメモリ管理とコア ランタイム ライブラリ（メモリ、スレッド、例外、その他の低レベル言語要素）に関して特定のアーキテクチャのみを念頭に置いている点です。これは、必ずそうしなければならないわけではなく、LLVM 内でさまざまなメモリ モデルを採用することができます。

とはいえ、モバイル プラットフォームの場合は、LLVM（または Java や .NET などの仮想実行環境）の使用とガベージ コレクションか自動参照カウント（ARC）のどちらかの採用がかなり一般的になっていることは注目に値します。ガベージ コレクションは（よかれあしかれ）多くの開発者にとってなじみがあるのに対して、ARC の方ははるかに知られていません。ARC の詳細については、（Clang プロジェクトの）以下のドキュメントで Objective-C での ARC の使用法を参照してください。

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

このことをもっとはっきりと言うならば、LLVM はさまざまなメモリ管理モデルで使用できますが、特に、リソース上の制約があるモバイル プラットフォームの場合には、採用に値する高度なメモリ管理モデルがいくつかサポートされています。さらに、メモリ管理をもっと自動化すると、新しい開発者に言語が採用されやすくなる効果もあります。

既存の Delphi コードをモバイル プラットフォームに移行するにはコードを見直す必要があることから、この素晴らしい新世界に向けて Delphi 言語を進化させるのにちょうどよい時期ではないかと私たちは思いました。これらの変更は最初はモバイル プラットフォームにのみ適用されますが、Delphi XE3 には、着手するのに使用できる特定の命令、ライブラリ関数、コンパイラ指令が既に含まれていました。これが、このホワイト ペーパーの中心となるトピックです。

1.3 DELPHI 言語の変更理由

当社では言語を発展させつつも、きわめて高い下位互換性を維持することを目標としています。それでは、現行の Delphi 言語に変更を加えているのは、いったいなぜなのでしょうか。それらの変更の中には、既存のコードを移行する際に問題が生じるおそれがあるものも含まれているのですから。

主な理由は以下のように 3 つあります。ここではそれらを簡単に述べ、詳細についてはこのドキュメントで後ほど説明します。

- 言語に機能をただ追加し続ける場合、特に、同じことを新しいやり方でできるようにしたり、既存のデータ型の新しいバージョンを追加するだけでは、言語は機能過多になり、複雑になりすぎ、保守も新しいプラットフォームへの移植も困難になります。

- 同じ基本的な事を行う方法が複数あると、言語の初心者にとっては、かなり紛らわしくなりがちです。

- 言語に矛盾点があると、重大な問題です（また、他の言語と少し異なる機能も、特に、言語に不慣れな開発者にとっては、やはり深刻な問題です）。たとえば、従来の Delphi では、ほとんどのデータ構造でデータ アクセスに 0 から始まるインデックスが使用されているのに、文字列では 1 から始まるインデックスが使用されていますが、これはまったく一貫性がありません。Delphi 開発者が（従来の Pascal における部分範囲型の使用法に従って）配列範囲を自由に定義できるのは事実ではありますが、動的配列のインデックスは 0 から始まります。

Delphi 言語に新規の開発者を引き込むという私たちの目標からすれば、不要な障害は、たとえささいなものでも取り除くのが適切です。モバイル開発用の Delphi がもたらすビジネス上および技術上のメリットにより、新規開発者によるこの言語の採用が大幅に伸びると予想しています。

1.4 現行のまま変わらない機能

答えは簡単、ほとんどすべてです。クラスとオブジェクト、メソッド、継承と多態性、インターフェイス、ジェネリック型、無名メソッド、リフレクション（つまり RTTI）など。

また、グローバル関数やグローバル変数を使用できるなど、従来からある一部の機能もまだ言語に含まれています。ちなみに、これらの起源は Turbo Pascal にさかのぼります。繰り返しになりますが、新しいコンパイラ アーキテクチャに移行しても変わらない機能がきわめてたくさんあるので、既存のコードを移行したり、既存の Delphi 開発者が現在の知識をそのまま役立てることができます。

現在、モバイル開発に取り組んでいる場合は、新しい言語、IDE、RTL、ユーザー インターフェイス ライブラリを習得するのが普通ですが、Delphi XE4 を使用する開発者は同じ IDE、言語、ライブラリなどをわずかな変更だけで使用できるというぜいたくが許されます。このドキュメントの焦点はそれらの変更点を説明することではありますが、言語の大部分（および既存の Delphi コードの大半）が既にクロスプラットフォーム対応になっており、モバイル プラットフォームに移行可能であることを明確に理解することが重要です。

1.5 XE4 の DELPHI コンパイラ

ここまでお読みいただいてお気づきのように、Delphi XE4 には実はいくつかの異なるコンパイラが付属しています。つまり、サポートしている 3 つのデスクトップ プラットフォームのそれぞれに 1 つ、Mac プラットフォーム上の iOS シミュレータ向けに 1 つ、ARM プラットフォーム向けの新しいコンパイラが 1 つです。このことで少し混乱が生じるおそれがあることを考慮して、以下にそれらを簡単にまとめておきましょう。

- Win32 用コンパイラ（DCC32）
- Win64 用コンパイラ（DCC64）
- Mac 用コンパイラ（DCCOSX）
- iOS シミュレータ用コンパイラ（DCCIOS32）

- iOS ARM 用コンパイラ（DCCIOSARM）

これら 5 つのコンパイラのうち、LLVM ツール チェーンに基づいているのは最後の 1 つだけですが、このホワイト ペーパーで詳しく述べるように、iOS シミュレータ用コンパイラでは文字列とメモリ管理に LLVM コンパイラの設定を一部使用しています。

2. 文字列型

文字列の管理は、Delphi 言語の中でも、大幅な変更がいくつか進んでいる分野です。これらの変更の裏には、いくつかの考えがあります。つまり、（少し異なる文字列型がいくつもある）従来の Delphi モデルの簡素化、（モバイル プラットフォームからの要求や LLVM プラットフォームで強いられるモデルなどに起因する）いくつかの最適化要件、Delphi 言語を一般に使用される他のプログラミング言語と整合させる必要性です。

Turbo Pascal や Delphi 1 に至るまで下位互換性を保つと、負荷が大きくなり、新規の開発者にとっても既存の開発者にとってもいくつか難しい問題が発生します。以下では、実際の変更点と、それらを強調したいくつかのサンプル コード、および将来の移行とクロスプラットフォーム互換性に向けたいくつかのヒントを示します。

2.1 唯一の文字列型

Delphi for Windows の最近のバージョンでは、さまざまな文字列型が急に増えました。Delphi には以下のものが用意されています。

- Pascal の短い文字列型（255 個の 1 バイト文字に制限）
- 参照カウントされるコピーオンライト対応の標準 AnsiString

- さらに明確に限定された AnsiString 派生型（文字列型構成メカニズムに基づく。UTF8String や RawByteString など）
- C 言語文字列（PChar）を管理するための RTL 関数群
- 参照カウントされるコピーオンライト対応の Unicode 文字列（現在、デフォルトの文字列型。UTF16 で実装）
- COM 互換のワイド文字列（やはり UTF16 に基づいて実装、参照カウントには対応せず）

LLVM ベースの新しい Delphi コンパイラには、文字列型が 1 つあります。これは Unicode 文字列（UTF16）を表し、Delphi XE3 の現行の文字列型（Windows 用のコンパイラでは UnicodeString 型のエイリアス）にマップされます。ただし、この新しい文字列型では、異なるメモリ管理モデルが使用されています。この文字列型はやはり参照カウントされますが、不変です。つまり、文字列をいったん作成すると、その内容を変更できません（詳細については、それに関するセクションで説明します）。言い換えれば、文字列は現在 Unicode ベースで、間もなく不変になります。また、参照カウントされます。

1 バイト文字列（ANSI 文字列や UTF8 文字列）を処理する必要がある場合、特に、ファイル システムやソケットとやり取りしているときは、動的バイト配列を使用できます（動的バイト配列については特に「1 バイト文字列の管理」のセクションで取り上げます）。とりあえず今は、コア機能と代わりのコーディング スタイルを示す実際のコードをいくつか紹介しましょう。新しいコンパイラにおける 1 バイト文字列の処理法をいくつか後で示します。

AnsiString 系列の型、ShortString 型、WideString 型、その他のあらゆる専用文字列型をまだ使用している場合は、すべてのコードを定義済みの文字列型に移行することを強くお勧めします（なお、Windows 用の Delphi コンパイラが今すぐにでも変わるわけで

はないので、これはもちろん、モバイル プラットフォームに移行するコードについての話です）。

メモ： これら "専用" の文字列型は主に、外界とインターフェイスを取るためのものでした。COM サポートのために導入された WideString 型は、この良い例です。これらの特殊な型は確かに便利でしたが、外界とインターフェイスを取ることだけを目的とした少し異なるセマンティクスと動作を持つ型によって、言語が何となく "汚染" されてしまいました。今や Delphi は他の多くのプラットフォームに進出しつつあるので、これらの型をいつまでも使い続けていると、物事が複雑になり、ややこしくなるだけです。たとえば、Windows 以外のプラットフォームでは WideString はどういう意味を持つのでしょうか。特化したデータ型を言語やコンパイラに組み込むよりも、プラットフォームとのインターフェイスに使用する型を細部まで明確にし、メソッドと演算子を持つレコード、ジェネリックスなどの言語機能を使ってそれらをカスタム型として作成する方がはるかによいでしょう。

2.2 参照カウントされる文字列

これまでもそうでしたが、Delphi 文字列は参照カウントされます。つまり、同じ文字列を 2 つ目の変数に代入するか文字列をパラメータとして渡すと、その参照カウントが増えます。すべての参照がスコープ外に出るとすぐに、その文字列はメモリから削除されます。

大半の開発者にとって、これはつまり、文字列のメモリ管理がきちんと機能しているので心配いらないということです。低レベルの実装（予告なく変更されることがあります）についてもっと理解したい場合は、以下の詳細に目を通してください。そうでなければ、次のセクションに進んでかまいません。

実装の詳細を調べる場合は、以下のコード断片に示すように、（Delphi 2009 以降追加されている）*StringRefCount* 関数で文字列の参照カウントを問い合わせることができます。

```
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;
```

これを実行すると、str2 の参照カウントは最初は 1 ですが、str1 への代入後は 2 に増え、str2 の変更後はまた 1 に戻ることがわかります。ここでは、（次のセクションで説明しているコピーオンライト メカニズムを使って）共有文字列の変更時に str2 への共有文字列データのコピー操作が行われます。このコードを Windows、シミュレータ、モバイル デバイスのどちらで実行しても、同じ一連の出力（1、2、1）が得られます（以下を参照）。

```
Initial RefCount: 1
After assign RefCount: 2
After change RefCount: 1
```

関数やメソッドの使用時に文字列を定数パラメータとして渡した場合、その文字列の参照カウントは変更されず、その結果、（たとえ数 CPU サイクルだけだとしても）コードの実行が速くなることに注意してください。これは、たとえば、参照カウントを返す関数で行われます。そうでない場合は、関数自身により参照カウントが常に 1 だけ増えます。

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

つまり、参照カウントは従来のコンパイラでも新しいコンパイラでもまったく同じように機能し、なかなか効率的な実装です。

2.3 コピーオンライトと不変文字列

ただし、事情が変わり始めるのは、既存の文字列を変更するときです。と言っても、新しい値に置き換えるのではなく（置き換える場合は、まったく新しい文字列が得られます）、以下のコード行に示すように（また、このトピックの導入部となった前のセクションでも示したように）、文字列の要素の 1 つを変更するときです。

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

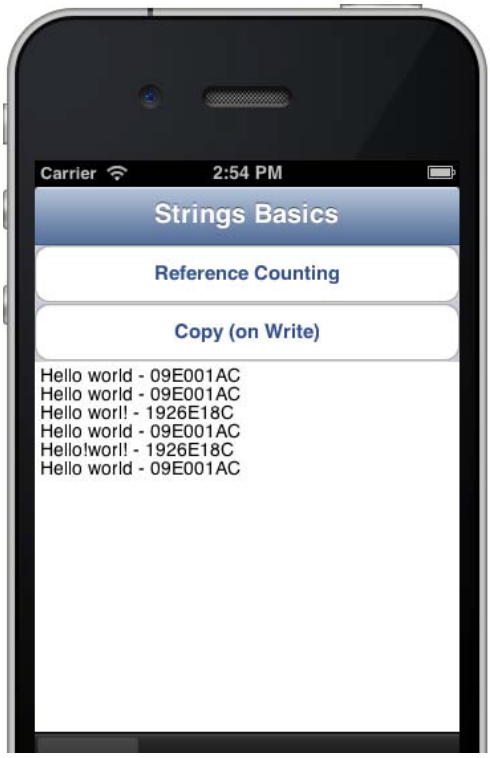
すべての Delphi コンパイラでは、コピーオンライト セマンティクスを使用しています。つまり、変更する文字列に複数の参照がある場合、その文字列はまずコピーされ（関与するさまざまな文字列の参照カウントを必要に応じて調整します）、後ほど変更されます。

メモ： "コピーオンライト" という用語になじみのない方々のために補足させていただくと、コピーオンライトとは、文字列が実際にコピーされるのは、それが新しい文字列変数に代入（コピー）されるときではなく、その文字列が変更されるときだけである、ということを示しています。つまり、プログラムでコピー操作を行ったときに文字列がコピーされるのではなく、後で事実上必要になったときにのみ、文字列がコピーされます。変更されない場合は、コピー操作はまったく行われません。

新しいコンパイラでは、従来のコンパイラとよく似たことを行います。文字列への参照が 1 つしかない場合は、文字列が直接変更されますが、それ以外の場合は、コピーオンライト メカニズムが実行されます。例として、以下のコードを考えてみましょう。ここでは、（デモ ソース コードにあるカスタム *StrMemAddress* 関数を使って）実際の文字列のメモリ位置を出力しています。

```
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;
```



このメソッドの出力は右の図に示すとおりです（iOS シミュレータでの表示です）。この図でわかるように、2 つの文字列の一方（str4）はまったく影響を受けず、もう一方は最初の書き込み操作のときにのみ再割り当てされます。

文字列を変更できるなら、従来のコンパイラと比較してどこが違うのでしょうか。文字列管理の現在の実装は従来のものと同等ですが、今後変更されることがあります。不変文字列により、メモリ管理モデルが改善されます。不変文字列を使用した場合、文字列の要素の 1 つを直接変更することはできなくなります。あるいは、その操作の実行が遅くなるおそれがあります。一方、文字列連結は高速になります。

そのようなわけで、このような操作がたとえ現在の Delphi ARM コンパイラで可能だとしても、文字列管理の内部実装は今後変更されることがあり、このような動作は無効になる可能性があります。

コードが最適化されていない可能性がある、あるいはコンパイラの将来のバージョンで動作しなくなるおそれさえあることを示す警告を出力させたい場合は、（既に Delphi XE3 で従来のコンパイラの場合に使用できるようになっている）特定の警告を有効にすることができます。*{$WARN IMMUTABLE_STRINGS ON}* がその指令で、これを有効にしたうえで、以下のようなコードを書くと、

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

以下のような警告が表示されます。

> [dcc32 警告]: W1068 直接編集での文字列の変更は将来的にサポートされない可能性があります

不変文字列に対する変更はコードにどのような影響があるのでしょうか。以下のような単純な文字列連結を行っている場合は、重大な問題はありません（連結のパフォーマンス向上は実は今後行われる可能性がある変更の目標の 1 つになっているので、将来も問題はありません）。

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

他の開発環境とは異なり、Delphi では、文字列連結がはるかに遅くなったり、どのような形であれ無効になるようなことはないと思っています。最終的に問題が発生するおそれがあるのは、文字列の個々の文字を変更する場合だけです。当社の研究開発チームがこの方向で現在行っている研究は、文字列の連結や Format 系関数の使用といった一般的な操作に対する最適化を模索することです。

いずれにせよ、文字列の変更や連結のプラットフォーム固有の実装による影響を受けたくなければ、実装やコンパイラへの依存度が低い文字列作成コード（たとえば *TStringBuilder* クラスなど）を使用するとよいでしょう。

2.4 TSTRINGBUILDER クラスの使用

今後は、多数のサブ要素から文字列を作成するための文字列作成クラス（*TStringBuilder* など）を使用することが重要な選択肢になります。個々の文字や短い文字列を連結して文字列を作成している場合は、Windows 向けのコンパイルでもモバイル向けのコンパイルでもコードの実行を速くする必要があれば、*TStringBuilder* を使用するようにコードを変更することをお勧めします。

同じループを標準的な文字列連結と *TStringBuilder* クラスをそれぞれ使って書いた簡単な例を以下に示します。

```
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;
```

Delphi の従来のバージョンでは、双方の実行速度は非常に近いです（上記コードにあるように文字列作成オブジェクトの最終サイズを事前に割り当てる場合は、ネイティブ連結の方が少し有利です。事前割り当てがない場合は、ネイティブ連結の方が 10 ～ 20% 高速です）。

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

Mac Intel プロセッサ向けにコンパイルされると仮定すると、iOS シミュレータで実行する場合も Windows 版のコードとほぼ同じ速度になり、数値は上記とよく似ています。

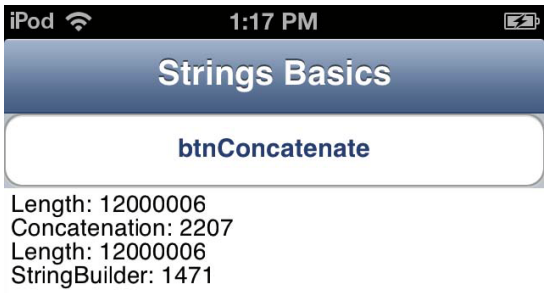
モバイル プラットフォーム（ARM コンパイラを使用する物理デバイス）では、普通の連結コードではかなり速度が遅くなると予想されるので、*TStringBuilder* は大きいサイズの文字列の場合に選ぶべき唯一の選択肢となります。このことに聞き覚えがある方がいれば、それはおそらく、Microsoft .NET プラットフォーム（たとえば、C# の使用時）などのマネージ プラットフォームで起こる現象とよく似ているからです。

文字列の再割り当てが必要なことは確かですが、メモリ マネージャは十分に高性能なので、それによる実行速度の低下は限られています（しかも、それは指数関数的ではなく線形です）。

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

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

上記の数値をよく見てください。iOS ARM デバイスでは、*TStringBuilder* を使用した場合はほぼ 2 倍速くなっています。ミリ秒ではなく秒数についての話です。プログラムの実際の出力は以下のとおりです。これは物理デバイスからキャプチャしたものです。



好ましいことに、*TStringBuilder* クラスは Delphi 2009 以来一般に使用されているので、古いバージョンの Delphi で保守しているアプリケーションにさえ追加できます。ネイティブ プラットフォームとモバイル プラットフォームでは処理速度が異なるのはわかりますが、それにしても、私が使っている MacBook Pro と iOS デバイスの速度差は非常に大きいです。

もう 1 つの例として、以下のループを考えてみましょう。ここでは、文字列のすべての要素をスキャンして、そのいくつかを置換しています。

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

不変文字列の問題を考慮すると、このコードの実行は非常に遅くなるのではないかと思われるかもしれません。上記の代わりに、以下のコードを使用することもできます。

```
// 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;
```

ふたを開ければ、文字列要素を直接置換するコードの方が、*TStringBuilder* を使用するコードより約 10 倍速いという結果になりました。正直なところ、次の大文字の A を探し、そこまでの部分文字列をコピーするようにして、アルゴリズムを大幅に最適化することはできます。しかし、（文字列の各要素をチェックする）前述の総当たりアルゴリズムを調べると、現在の実装では、より単純な最初のコードの方が速いのです。これは文字列構造の直接変更であり、不変文字列の実装によってそれが防止される可能性があることから、今後、この状況は変わるかもしれません。

メモ： 繰り返しますが、今後、文字列の内部実装は変わる可能性があることを覚えておいてください。長い文字列を文字の直線的な並びではなく部分文字列のコレクションで表す言語があります。今日、最適化コードを作成しても、明日には、それが遅い実装になるかもしれません。

2.5 0 から始まるインデックス方式によるアクセス

2 番目の変更点は、0 から始まるインデックス方式の文字列をサポートしたことです。つまり、1 ではなく 0 から始まるインデックスと角かっこを使って文字列の要素（文字）にアクセスできるということです。1 から始まるインデックス方式の文字列を使用するのは、Pascal 言語で初期の実装以来、綿々と行われてきた慣習の 1 つです。もっとも、このように決めた理由は、読みやすさや要素のより自然な数え方とはほとんど関係がなく、実装上の判断の結果として、こうなっただけです。言い換えれば、実装（および、C 言語にあるようなヌル終端文字を使用しない代わりに、文字列の長さを格納するために 0 番目のバイトを使用する必要があったこと）が仕様として表れたものです。

どうするのがより自然かを詳細に論じることはできますが、確かなことは、大半のプログラミング言語で 0 から始まるインデックス方式の文字列が使用されており、その他の事実上すべての Delphi データ構造でも 0 から始まるインデックス方式が採用されて

いることです。つまり、動的配列、コンテナ クラス、*TStringList* のような RTL クラス、サブ要素（メニュー項目、リスト ボックス項目、サブコントロールなど）を持つ VCL クラスおよび FireMonkey クラスなどのデータ構造です。

これが言語の大幅な変更であり（新しいコンパイラ アーキテクチャやモバイル サポートには関係していない）、大量のコードに影響を及ぼすおそれがあることを考えて、この変更は、既に Delphi XE3 で使用可能になっている新しいコンパイラ指令 *$ZEROBASEDSTRINGS* で制御されます。Delphi XE3 では、この指令はデフォルトで OFF になっており、XE4 では、モバイル コンパイラの場合にはデフォルトで ON になっています。ただし、（これが指令であることから）既に Delphi XE3 でこれを ON にして、このモデルへのコード ベースの移行に着手することもできますし、あるいは、モバイル コンパイラでこれを OFF にして、（既存のコードを移行して、0 から始まる文字列インデックス方式を使用するようになるまで）既存のコードを引き続き動作させることもできます。

なお、プロジェクト オプション レベルで指定できる、これに対応する拡張構文コンパイラ指令もあり、それは（異なるローカル設定がない限り）すべてのユニットに適用されます。これを使用する場合は、必ず、対応する文字列アクセス モデルを念頭に置いて作成したユニットだけをプロジェクトに組み込むようにしてください。ユニット レベルでのローカル設定が不一致を避けるのに役立つ可能性があるのは、このためです。

以下のとおり、注意すべき重要な関連事項がいくつかあります。

- 文字列の内部構造は影響を受けません。この指令の設定が異なるユニットを単一のプロジェクトで混在させることや、文字列をそれとは異なる設定でコンパイルされた関数に渡すことができます。特定のソース コード行の [] 式をコンパイラがどう解釈しようと、文字列はあくまで文字列です。

- *$ZEROBASEDSTRINGS* が OFF であれば、従来の文字列処理用 RTL 関数は既存のセマンティクスを保ち、文字列要素の位置が戻り値や引数になる場合は、1 から始まるインデックスが使用されます。ただし、従来の文字列処理用 RTL 関数はもう使用しないようにすることをお勧めします。これらは、下位互換性を保つために使用できるようになっているものです。代わりに、新しい *TStringHelper* の関数（後述）を使用することをお勧めします。

- 組み込み型ヘルパ *TStringHelper* には、一連の新しい関数が含まれており、今後、Delphi 開発者が自分のコードを移行してこれらを使用するようにすることをお勧めします。このヘルパ クラスについては次のセクションで説明しますが、このクラスでは、コンパイラによらず、0 から始まる文字列インデックス方式を使用します（つまり、Windows、Mac、モバイル デバイスのどれでも同じように動作します）。

次の簡単なコード断片について考えてみましょう。

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

Delphi XE3 の場合、デフォルトでは、ボタンのキャプションは "*HOllo foo*" になります。しかし、メソッドの前に以下の指令を追加した場合は、

```
{$ZEROBASEDSTRINGS ON}
```

ボタンのキャプションは "*HeOlo foo*" になります。この場合、インデックスが 0 から始まる方式になることから、文字列の [2] 要素は実際には第 3 要素になるのです。2 つのコンパイラの違いは、次のようなコード行がどう評価されるかに現れます。

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

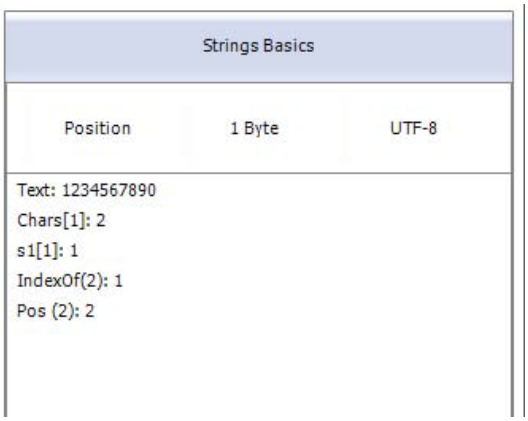
上記コードの実際の動作は *$ZEROBASEDSTRINGS* コンパイラ指令で制御されますが、そのデフォルト値が 2 つのコンパイラで異なるのです。コードを新しいモデル（0 から始まるインデックス方式の文字列）に移行することを強くお勧めします。Delphi XE3 で作成する Windows アプリケーションや Mac アプリケーションでも、この設定をできる限り有効にします。単一の文字列インデックス モデル（あるいは、後述のような、より抽象的なコーディング慣習）に迅速に移行すると、今後、コードが確実に読みやすくなります。

メモ： この結果、（Delphi でこれまで文字列管理に対して行われてきた変更が原因となって）少し混乱が生じているので、文字列要素アクセス演算子 [] をコンパイラがどう解釈するかは、文字列のメモリ内構造とはまったく関係がないことに注意してください。つまり、文字列は内部的にはまったく変わらず、*$ZEROBASEDSTRINGS* 指令を ON にしようと OFF にしようと、文字列の動作が異なることはありません。[i] のコードに対するコンパイラの解釈方法が変わるだけです。そのため、異なる設定でコンパイルしたさまざまなユニットや関数をプログラムに混在させることができます。0 から始まるインデックス方式の文字列を関数に渡すわけではなく、指定されたコンパイラ設定を使用するコードとの間で文字列を受け渡すだけです。

これは重要なことなので、以下のコード断片の結果を見てみましょう。ここでは、起こり得る問題を浮き彫りにしています。

```
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)));
```

Windows と Mac での出力に比べると、デフォルトでは、シミュレータとデバイスで結果が異なります。シミュレータのスクリーンショットと、必要最小限の機能だけを備えた Windows 版のスクリーンショットを以下に示します。



出力は、iOS では 2、2、1、2 の順ですが、Windows では 2、1、1、2 の順になります。やはり、2 番目の値だけが異なりますが、ここは、[] を使った直接アクセスが使用されている所です。ただし、*$ZEROBASEDSTRINGS* 指令を使用することで、各プラットフォームでの動作を逆にすることができます。Windows で *$ZEROBASEDSTRINGS* ON の場合、出力は 2、2、1、2 の順ですが、iOS で *$ZEROBASEDSTRINGS OFF* の場合、出力は 2、1、1、2 の順になります。

ほとんどの場合、こうした事態は避けることができますが、もし、従来のアクセス指定子を使って特定の位置の文字列要素にアクセスする必要があり、コンパイラとその設定にかかわらずコードが必ず動作するようにしなければならない場合は、以下のように、

Low() 関数の値に関係する定数を定義できます。この関数は、文字列のインデックスの下限を返すものです。

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

この定数の値は 2 か 3 で、文字列の 3 番目の文字にアクセスするのに適した方になります。このような定数をいったん定義したら、それを以下のように参照できます。

```
s1[thirdChar]
```

コンパイラの種類にかかわらず動作するコードは、文字列の基底インデックスを意識しないコードの例です。

一般に、文字列の要素をループで処理する際にも同様の取り扱いが必要になります。一例として、従来のループの代わりに以下のようなコードを使用することができます。

```
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 関数と High 関数。Low(s) は、0 から始まるインデックス方式の文字列の場合には 0 を返し、1 から始まるインデックス方式の場合には 1 を返します。High(s) は、0 から始まるインデックス方式の場合には Length(s) - 1 を返し、1 から始まるインデックス方式の場合には Length(s) を返します。空の文字列をパラメータとして渡した場合、Low は上記と同じ値を返すのに対して、High は -1（0 から始まるインデックス方式の場合）か 0（1 から始まるインデックス方式の場合）のどちらかを返します。また、データ型 string を Low に渡して現在の設定を判断することもできます。一方、その型を High に渡しても意味はありません。

2.6 組み込み型ヘルパ TSTRINGHELPER の利用

Delphi XE3 からでも使用できるもう 1 つのアプローチは、データ型 string のヘルパです。XE3 で導入された新しい言語機能の 1 つとして、実は、既存のクラスやレコードだけでなくネイティブ型（もちろんレコードではない）にもカスタム メソッドを追加できるようになりました。その結果登場したのは、Delphi 開発者にとっては少し変な構文です。以下は、カスタム定義型に基づいた例です。

```
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;
```

Delphi XE3 では、この新しい言語構文要素と、その実用的な実装の 1 つである string 型のレコード ヘルパ *TStringHelper* が導入されました。*TStringHelper* は SysUtils ユニットに定義されており、*Compare*、*Copy*、*IndexOf*、*Substring*、*Length*、*Insert*、*Join*、

Replace、*Split* など、多数のメソッドを提供します。たとえば、次のようなコードを作成できます。

```
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;
```

なお、（インデックス付けされたプロパティ *Chars* のほか）これらのすべてのメソッドでは、既に説明した 0 から始まるインデックス方式の表記を使用します。関連するコンパイラ指令の値には左右されません。

上記のコードでは、メソッド連鎖、つまり、特定の型のオブジェクトに適用され、そのオブジェクトそのものか同じ型の別のオブジェクトを返すメソッドの定義が使用されていることに注意してください。

2.7 1 バイト文字列の管理

セクション 2.1 で示したように、1 バイト文字列型はすべて Delphi ARM コンパイラではサポートされていません。と言っても、もちろんそのようなデータ構造を処理できないということではなく、単にもうネイティブ データ型では扱わないということです。現実的には、*AnsiString*、*AnsiChar*、*PAnsiChar* のようなデータ型を使用することはできません。

一例として、Unicode UTF8 形式を使用する必要性を考えてみましょう。そのようなファイルがある場合は、*TTextReader* インターフェイスとそれによるエンコーディングサポートに基づいて、より高いレベルのアプローチを用いることができます。

```
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);
```

これはかなり書きやすいコードですが、1 バイト UTF8 文字列の直接管理を隠蔽しています。これと同じ効果を以下のかなり複雑な低レベル コードで実現することができます。このコードでは内部機能のいくつかを強調しています。

```
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);
```

このようなメモリ内のデータ構造を処理した方がよい状況があるかもしれません（たとえば、低レベルの関数呼び出しを行って何らかの直接データ操作を行う場合など）。このような場合は、動的バイト配列を使用することを強くお勧めします。そのような配列をカスタム データ構造にラップすることで、現在の動作をまねることさえできるでしょう（以下に非常に簡単なバージョンを示します）。

```
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;
```

演算子オーバーロードを伴うこのレコードは、*TUTF8Encoding* クラスを基に実装することができます。このクラスには、UTF-8 バイト配列を標準の UTF-16 文字列に（またはその逆に）変換するメソッドが、すぐに使用できる状態で用意されています。

このようなレコードがあると、以下のようなコードを新しい Delphi ARM コンパイラでコンパイルすることができます。このコンパイラでは *UTF8String* 型はあらかじめ定義されていません。

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

3. 自動参照カウント

"長い文字列" が導入された Delphi 2 以降、Delphi では、メモリ管理は参照カウントに基づいています。このホワイト ペーパーで既に詳しく述べたように、文字列では参照

カウントを使用しており、文字列へのすべての参照がスコープ外に出ると、その文字列はメモリから削除されます。Delphi 3 以降は、オブジェクトに対しても参照カウントが一部サポートされています。ただし、インターフェイス型変数を使用して、それらのオブジェクトを参照する場合に限ります。最後に、動的配列でも参照カウントを使用しています。

そのため、参照カウントは Delphi の世界では目新しくはありませんが、新しい ARM コンパイラでは、このたび初めて、すべてのクラスとオブジェクトに対して自動参照カウント モデルが完全にサポートされました。詳細に入る前に、まず、このトピックの概論から始めさせてください。

自動参照カウント（ARC）とは何でしょうか。LLVM に関する先述のセクションでリンクされているページをご覧になるとおわかりのように、ARC は、もう必要ないオブジェクトを明示的に解放しなくてもオブジェクトの存続時間を管理できる手段です。オブジェクトへの参照（たとえばローカル変数）がスコープ外に出ると、そのオブジェクトは自動的に破棄されます。Delphi では、文字列と、インターフェイス型変数を通じて参照されるオブジェクトに対して既に参照カウントをサポートしています。そのため、オブジェクトについて言えば、Delphi for Windows で ARC に最も近いのはインターフェイスの使用です。ただし、この後すぐに説明するように、ARC では、今日 Delphi においてインターフェイス型変数では対処しにくい循環参照のような問題をもう少し柔軟に解決できます。

ガベージ コレクション（GC）とは異なり、ARC は決定論的です。オブジェクトの生成と破棄は、別個のバックグラウンド スレッドではなくアプリケーション フロー内で行われるのです。これには長所も短所もありますが、GC 対 ARC の議論はこのホワイトペーパーの範囲をはるかに超えています。

メモ： ARC は、どのコンパイラで有効になっているのでしょうか。

新しい LLVM ベースのコンパイラにはデフォルトで ARC 機能が備わっていますが、iOS シミュレータ用のコンパイラ（厳密には、Intel CPU および Mac OS X オペレーティング システム用のコンパイラ) でも ARC は有効であることに注意してください。たとえ、そのコンパイラが従来のコンパイラ アーキテクチャに基づいているとしてもです。このように、シミュレータとモバイル デバイスでのメモリ管理は一致することになります。

3.1 ARC コーディング スタイル

新しいコンパイラで参照カウントがサポートされていることから、メソッド内で一時オブジェクトを参照する必要がある場合は、以下のように、メモリ管理を完全に無視することでコードを大幅に簡略化することができます。

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

この特定のテスト ケースでは、*TMySimpleClass* にデストラクタを追加し、フォームにログを記録しました（デモでは、メソッド自体にもう少しログ記録がありますが、ここでは話を簡単にするために省略しています）。ここで何が起こるかというと、プログラムの実行が end 文に達したとき、つまり *MyObj* 変数がスコープ外に出ると、オブジェクトのデストラクタが呼び出されます。

何らかの理由で、メソッドが終了する前にオブジェクトの使用をやめる場合は、以下のように、この変数を nil に設定できます。

```
class procedure TMySimpleClass.SetNil;
var
        MyObj: TMySimpleClass;
begin
        MyObj := TMySimpleClass.Create;
```

```
        MyObj.DoSomething (False); // True => raise
        MyObj := nil;
        // do something else
end;
```

このような場合は、メソッドが終了する前でも、変数を nil に設定した時点でオブジェクトは破棄されます。しかし、try-finally ブロックがないことを考えると、*DoSomething* 手続きで例外が発生したら、どうなるのでしょうか。nil の代入文はスキップされますが、それでも、このメソッドが終了した時点でオブジェクトは破棄されます。

まとめると、オブジェクトを変数に代入したときと変数がスコープ外に出たときに参照カウントがトリガされるということです。この変数は、スタックベースのローカル変数、コンパイラにより追加された一時変数、別のオブジェクトのフィールドのいずれの場合もあります。同じことがパラメータにも当てはまります。つまり、オブジェクトをパラメータとして関数に渡すと、そのオブジェクトの参照カウントがインクリメントされ、その関数が終了して戻ったときに参照カウントがデクリメントされます。

メモ： パラメータ受け渡しの最適化

文字列の場合とちょうど同じように、*const* 修飾子を使ってパラメータの受け渡しを最適化できます。オブジェクトを定数として渡すと、参照カウントのオーバーヘッドは発生しません。この修飾子は Windows では役に立たないながらも使用できることを考えると、*const* のオブジェクト パラメータや *const* の文字列パラメータを使用するようにコードを更新するのが得策です。もっとも、参照カウントのオーバーヘッドは非常に限られているので、あまり大幅な速度向上は期待できませんが。

今度の裏技を広く使用するのではなく、オブジェクトの存続時間がプログラム フローに従うようにすべきではありますが、以下のような新しいプロパティを使って、（参照カウントに対応しているプラットフォームでのみ）オブジェクトの参照カウントを問い合わせることができます。

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

メモ： インターロックされた操作の速度

スレッドセーフになるように、オブジェクトの参照カウントのインクリメント操作とデクリメント操作は、"インターロックされた" 操作つまりスレッドセーフな操作を使って実行されます。以前には、*LOCK* 命令の実行時に Intel CPU がすべてのパイプライン /CPU を停止するため処理が遅くなることもありました。現代の Intel CPU では、関係のあるキャッシュ ラインのみロックされます。モバイル プラットフォームで使用される ARM CPU でも状況は似ています。インクリメント操作とデクリメント操作がスレッドセーフだからと言って、今ではインスタンスが一般に "スレッドセーフ" であるというわけではありません。すべてのスレッドで変更が直ちに認識され、"古い" 値が変更されることが絶対にないように、参照カウントを表すインスタンス変数が適切に保護されているだけです。

メモ： ARC とコンパイラの互換性

たとえばライブラリに各シナリオ（ARC 対応と ARC 非対応）ごとに最良のコードを作成する場合は、*{$IFDEF AUTOREFCOUNT}* 指令を使用して両者を判別することを検討するとよいでしょう。ARC が従来の Delphi コンパイラにも実装されるようなことに将来なるかもしれないので（iOS シミュレータでは既にそれが起こっています）、これは NEXTGEN とは異なる重要な指令です。

3.2 ARC 下での FREE メソッドと DISPOSEOF メソッド

Delphi 開発者は、*Free* メソッドの呼び出しをベースにした別のコーディング パターンに慣れています。また、このパターンは通常、try-finally ブロックで保護されています。Delphi を使用してきた開発者の大半は、このパターンに基づいた大量のコードを所有しているでしょうし、Delphi for Windows の互換性を保つためにこのようなコードを書く必要がまだあるかもしれないことを考えると、ARC の下で *Free* を使用することに着

目することが大切です。手短に言えば、既存のコードは動作しますが、今後もずっと、コードを読んでその動作を理解しなければなりません。

たとえば、上記のコードなら一般に次のように書くでしょう。

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

従来の Delphi コンパイラの場合、*Free* は *TObject* のメソッドで、現在の参照が nil でないかどうかを確かめ、該当する場合は *Destroy* デストラクタを呼び出します。その結果、適切なデストラクタ コードの実行後、オブジェクトがメモリから削除されます。

新世代の Delphi コンパイラの場合は、そうではなく、*Free* の呼び出しの "代わり" に、変数への *nil* の代入が使用されます。これがオブジェクトへの最後の参照である場合は、オブジェクトのデストラクタの呼び出し後、やはりオブジェクトがメモリから削除されます。他に参照が残っている場合は、（参照カウントの減少以外は）何も起こりません。

同様に、次のような呼び出しでは、

```
FreeAndNil (MyObj);
```

オブジェクトを nil に設定し、そのオブジェクトを参照している変数が他にない場合に限り、やはりオブジェクトの破棄をトリガします。ほとんどの場合はこれで間違いはありません。プログラムの別の部分で使用されているオブジェクトを破棄しようとはしないからです。とは言え、保留中の参照が他に残っている可能性があっても、（たとえば、

ファイルやデータベース接続を閉じるために）デストラクタ コードを本当にすぐに実行しなければならない場合もあります。

つまり、*Free* や *FreeAndNil* の呼び出しは役に立たないことが多いのですが、通常は完全に無害なので、モバイル プラットフォーム向けの Delphi プログラムにそのまま残しておいてもかまいません。もっとも、別のアプローチが必要な場合も、限られてはいますが、いくつかあります。

開発者が（実際のオブジェクトをメモリから解放せずに）デストラクタを強制的に実行できるようにするため、新しいコンパイラには破棄パターンが導入されています。以下を呼び出す場合、

```
MyObject.DisposeOf;
```

たとえ保留中の参照があっても、デストラクタ コードが強制的に実行されます。破棄操作がさらに実行される場合や参照カウントがゼロに達しメモリが実際に解放される場合にデストラクタが再度呼び出されないように、この時点でオブジェクトは特別な状態に置かれます。この **"破棄"** 状態（または **"ゾンビ"** 状態）は非常に重要なので、オブジェクトがその状態にあるかどうかを、*Disposed* プロパティを使って問い合わせることができます。

メモ： Win32 での DisposeOf
この新しいメソッドは Windows や Mac 用の従来のコンパイラでも使用できますが、その場合、*DisposeOf* の呼び出しは *Free* メソッドの呼び出しに再マッピングされます。つまり、新しいメソッドはほとんど違いがなく、異なるプラットフォーム間のソースコードの互換性を高めるために導入されています。

この破棄パターンは、なぜ必要なのでしょうか。要素のコレクションや、別のコンポーネントに所有されているコンポーネント群の場合を考えてみましょう。よくある使用パターンは、コレクション内の特定の項目を、その項目自体のクリーンアップとコレクシ

ョンからの削除を目的に "破棄" することです。よくあるシナリオとしては、もう 1 つ、コレクションやコンポーネント オーナーを破棄し、そこに含まれている要素や所有されているコンポーネントをすべて処分する場合があります。このシナリオでは、所有されているオブジェクトに対する保留中の参照がまだある場合でも、それらのオブジェクトが強制的に破棄されるか、少なくとも、それらのデストラクタ コードが強制的に実行される可能性が高いです。

破棄されたこれらのインスタンスを処分後に使用すると、エラーになるおそれがありますが、それは、従来の Delphi コンパイラの場合と大差ありません。従来のコンパイラでも、インスタンスを解放してからそれに対する他の参照を使用すると、やはりエラーになります。大きく異なるところは、従来の Delphi コンパイラでは、あるオブジェクトへの参照が 2 つあり、そのうちの一方を使ってそのオブジェクトを解放すると、もう一方の参照がまだ有効かどうかがわからないことです。代わりに *DisposeOf* を使用した場合は、以下のように、オブジェクトにその状態について問い合わせることができます。

```
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';
```

先に述べたように、従来の Delphi コンパイラの場合に使用される従来の try-finally ブロックは、たとえ必要なくても、新しいコンパイラでまだ問題なく動作します。他の参照を考慮せずに、デストラクタ コードをできるだけ早く強制的に実行する必要があるような場合には、（もちろん、Delphi の以前のバージョン向けにコードを再コンパイルするのでない限り）代わりに以下のような破棄パターンを使用するとよいでしょう。

```
var
        MyObj: TMySimpleClass;
```

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

従来のコンパイラでは、*DisposeOf* が *Free* を呼び出すので、結果は変わりません。ARC が有効な場合は、そうではなく、上記のコードで予想どおりに（つまり、従来のコンパイラの場合と同じタイミングで）デストラクタが実行されますが、メモリは ARC メカニズムで管理されます。これはすばらしいですが、Delphi の以前のバージョンとの互換性を保つには、これと同じコードを使用することはできません。その目的には、*FreeAndNil* を *Free* か *DisposeOf* のどちらかの呼び出しとして再定義したうえで呼び出すとよいでしょう。あるいは、単に *Free* の標準的な呼び出しを使用し続けます。ほとんどの場合は、これでうまくいきます。

メモ： Disposed フラグの格納場所

Disposed "フラグ" の実際の格納場所は、特別なフィールドではなく、*FRefCount* フィールド内の 1 ビットです。関係する破棄の追跡に第 2 ビットが使用されることから、参照カウントの理論的上限は 2^30 で、これは実質的に上限があるとは考えられません。

Free と *DisposeOf* の違いの見方の 1 つは、（従来の Delphi コンパイラではどうなるかではなく）ARC 下での 2 つの操作の意図を考えることです。*Free* を使用する場合、その意図は、特定の参照をインスタンスから単に "切り離す" 必要があるということです。いかなる種類の破棄やメモリの割り当て解除も意味していません。そのコード ブロックではその参照がもう必要ないということにすぎません。通常、これはスコープから出るときに行われるだけですが、*Free* を明示的に呼び出すことで強制的に実行することもできます。

これに対して、*DisposeOf* は、プログラマがインスタンスに **"自分自身をクリーンアップする"** 必要があることを明示的に伝える手段です。*DisposeOf* は、必ずしもメモリの割り当て解除を意味しておらず、（特定のデストラクタ コードを実行して）ただインスタンスの明示的な **"クリーンアップ"** を行うだけです。インスタンスは依然として通常の参照カウント セマンティクスに基づいて、使用したメモリを最終的に割り当て解除します。

言い換えれば、ARC の下では、*Free* は "インスタンスへの参照を中心とした" 操作であるのに対して、*DisposeOf* は "インスタンスを中心とした" 操作です。Free が意味しているのは、言わば、"インスタンスがどうなろうとかまわない。もうそれが必要ないだけだ" ということです。一方、*DisposeOf* の方は、"このインスタンスには、解放しなければならない非メモリ リソースが保持されている可能性があるので、インスタンスに自分自身を内部的にクリーンアップしてもらいたい" ということを意味しています（なお、このような非メモリ リソースとしては、ファイル ハンドル、データベース ハンドル、ソケットなどがあります）。

DisposeOf のもう 1 つの用途は、参照の複雑な循環の適切なクリーンアップと割り当て解除を明示的にトリガすることです。[Weak] 参照（次のセクションで説明）を使用すると、物事がもっと明確になる一方、他のインスタンスにそれらの参照を削除するように "命じる" には明示的なトリガまたは通知が必要になるような状況が出てくる可能性があります。

メモ： ポインタとオブジェクトの混在

たとえば、オブジェクトをポインタに代入し、そのオブジェクト変数を別のオブジェクトに再利用し、後でそのポインタを元のオブジェクト変数に代入した場合、そのオブジェクトはもうそこには存在しません。（ポインタは参照としてカウントされず、参照カウントを増やさないことから）オブジェクトの参照カウントがゼロになると、オブジェクトは破棄されます。一例として、RTL の *TStringList.ExchangeItems* メソッドの変更

点を見てください。このメソッドでは、これまでポインタを使用して、新しい場所に移動されるオブジェクトの一時 "参照" を保持していました。

3.3 弱い参照

ARC のもう 1 つの非常に重要な概念は、弱い参照の役割です。2 つのオブジェクトがそれぞれのフィールドの 1 つを使って互いを参照し合っており、ある外部変数が第 1 のオブジェクトを参照しているとしましょう。第 1 オブジェクトの参照カウントは 2 になります（外部変数と第 2 オブジェクトのフィールド）。これに対して、第 2 オブジェクトの参照カウントは 1 です（第 1 オブジェクトのフィールド）。さてここで、外部変数がスコープ外に出ると、2 つのオブジェクトの参照カウントは 1 のまま変わらず、それらはいつまでもメモリに残ります。

この種の状況を解決するには、循環参照を解除しなければなりませんが、その操作をいつ実行すべきかがわからないことを考えると、それは決して単純なことではありません（最後の外部参照がスコープ外に出たときに実行すべきですが、オブジェクトにはその事実がわかりません）。このような状況やそれに似た多くのシナリオを解決する方法は、弱い参照を使用することです。

弱い参照とは、参照カウントを増やさないオブジェクト参照のことです。前のシナリオを考えると、第 2 オブジェクトから第 1 オブジェクトへの参照が弱い参照であれば、外部変数がスコープ外に出たとき、両方のオブジェクトは破棄されます。

以下のような単純な状況を記述したコードを見てみましょう。

```
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;
```

以下のように、"複合" クラスのコンストラクタにより、もう一方のクラスのオブジェクトが生成されます。

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

FOwnedBy フィールドは弱い参照であることを思い出してください。そのため、参照先のオブジェクト（この例では現在のオブジェクト *self*）の参照カウントは増えません。このクラス構造を前提として、以下のコードを作成できます。

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

弱い参照が適切に使用されていれば、この結果、メモリ リークが発生することはありません。

弱い参照のさらなる使用例としては、Delphi RTL に含まれている以下のコード断片に注目してください。これは *TComponent* クラス宣言の一部です。

```
type
```

```
TComponent = class(TPersistent, IInterface,
        IInterfaceComponentReference)
private
        [Weak] FOwner: TComponent;
```

従来の Delphi コンパイラでコンパイルされるコードで [Weak] 属性を使用しても無視されることに注意してください。ただし、必ず、"所有する側の" オブジェクトのデストラクタに適切なコードを追加して "所有される側の" オブジェクトも解放する必要があります。これまで見たように、Free の呼び出しが可能です。その結果は、従来の Delphi コンパイラと Delphi ARM コンパイラで異なりますが、どちらの場合もほとんどの状況で正しく動作します。

上記の例にあるような弱い参照を使用する場合は、その弱い参照そのものを検査して nil かどうか確かめないでください。ではどうするかと言うと、まず弱い参照を強い参照に代入し（その結果、いくつかの確認が裏で組み込まれます）、その後で、その強い参照値を確認するのです。一例として、上記の弱い参照 *FOwner* を前提として、以下のようなコードを作成することもできます。

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

3.4 メモリの問題のチェック

iOS 向けの Delphi ARM コンパイラの下でメモリ管理がどう機能するかを考えると、すべてを確実に管理できるようにするための選択肢をいくつか検討することも価値があります。先に進む前に、Windows 以外のプラットフォームでは Delphi はメモリ マネージャ FastMM を使用しないので、グローバル フラグ *ReportMemoryLeaksOnShutdown* を設定して、プログラム終了時のメモリ リークの有無を調べても無駄であることに注意

してください。実際に OS X と iOS の両プラットフォームでは、Delphi はネイティブ *libc* ライブラリの *malloc* 関数と *free* 関数を直接呼び出します。

iOS プラットフォームの場合、とてもすばらしい解決法は Apple 社の Instruments ツールを使用することです。このツールは、物理デバイスで動作するアプリケーションをあらゆる角度から監視する完全な追跡システムです。このツールについては、Delphi の観点からこのツールを詳しく解説しているビデオが参考になります。このビデオは Daniel Magin 氏と Daniel Wolf 氏によるもので、以下の URL でアクセスできます。

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

メモリ リークを発生させるおそれのある問題の 1 つがオブジェクト間の循環参照であることから、アプリケーションの動作を把握するのに役立つ可能性のあるちょっとした手続きが用意されています。それは Classes ユニットに含まれているもので、以下のように CheckForCycles という名前です。

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

これは最終コードで一般に使用する手続きではなく、開発時やデバッグ時のテスト専用のものです。この手続きの第 2 パラメータは、オブジェクトのクラス、そのメモリ アドレス、循環に関与するオブジェクトが格納されているスタックをパラメータとして受け取る無名メソッドです。以下はそのかなり基本的な使用例で、先述のクラス（弱い参照を持ち、循環がない）から弱い参照を取り除いたものをベースにしています。

```
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 UNSAFE 属性

参照カウントがゼロに設定されたオブジェクトを関数が返す可能性があるような非常に限定的な状況がいくつかあります（たとえば、オブジェクトの生成時など）。このような場合、（変数に代入されると参照カウントが 1 に増えるにもかかわらず、その機会を得る前に）そのオブジェクトがコンパイラにすぐに削除されないようにするため、そのオブジェクトに "Unsafe"（危険）のマークを付ける必要があります（コードを "安全" なものにするため、そのオブジェクトの参照カウントを一時的に無視しなければならないため）。

この動作は新しい固有の属性 [Unsafe] を使用することで実現されます。これは、非常に限定的な状況でのみ必要になる機能です。

```
var
        [Unsafe] Obj1: TObject;
[Result: Unsafe]
function GetObject: TObject;
```

System ユニットでは、対応する指令 *unsafe* が属性の代わりに使用されます（属性が同じユニットで定義される前に、その属性を使用することはできないからです）。その一例が、*TObject* クラスの低レベル クラス関数 *InitInstance* です。これは、オブジェクトのメモリを割り当てるのに使用されるもので、以下のように宣言されています。

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

（上記のコードに示すような）unsafe 指令の使用は System ユニットに限られます。

3.6 参照カウントを扱う低レベル操作

ほとんどのシナリオでは、参照カウントの使用に合わせてコードを修正するだけで済み、場合によっては弱い参照と [Unsafe] 属性を必要に応じて追加することになりますが、オブジェクトの直接メモリ割り当てをカスタム的に管理するような状況もあります。それと似たシナリオでは、オブジェクトが参照カウントの点で適切に管理されず、最終的には、オブジェクトをメモリに残す必要があるにもかかわらずその実際の参照はないという状況になるおそれがあります。このような（めったにない）ケースでは、*TObject* クラスの 2 つの public 仮想メソッド（以下を参照）を呼び出すことで、参照カウントを強制的に変更できます。

```
function__ObjAddRef: Integer; virtual;
function __ObjRelease: Integer; virtual;
```

どちらのメソッドも、操作後の参照カウントを返します。

メモ： 参照カウントの速度

上記の 2 つの関数は Delphi における参照カウント コードの中核部分を実装しており、その部分は必要に応じてコンパイラにより自動的にトリガされます。オブジェクトを新しい変数に代入するとき、オーバーヘッドは仮想メソッド テーブル内の関数への 1 回の呼び出しで、今度はそれがオブジェクト自身のフィールドをインクリメントします。これは、Apple 社による ARC for Objective-C の現在の実装を始めとする代替実装よりもずっと高速です。

これらのメソッドを使用する可能性のあるシナリオは、オブジェクト型として扱うまたはオブジェクト型にキャストするメモリ ブロック（おそらく何らかの外部 API で割り当てられたもの）がある場合です。もう 1 つの例は RTL から引用したもので、以下のように、ポインタを使ってオブジェクトのデータをコピーする場合です。

```
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 ARC 下での TOBJECT の生成メソッドと破棄メソッドのまとめ

TObject クラスのメソッドのうち、生成と破棄に関係するものを以下にまとめます（ここには新しいものも従来のものも含まれていますが、条件付きで定義されたいくつかの指令は省略されています）。

```
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;
```

ここで、参照カウントに対応するものも対応しないものも含め、さまざまなプラットフォームに対する各メソッドの実装の詳細に立ち入ることも確かにできるのですが、それは、このような新しい言語機能の入門的解説のトピックとしては高度すぎます。

3.8 ボーナス機能：クラスの演算子オーバーロード

メモリ管理に ARC を使用する場合は、関数から返される一時オブジェクトの存続時間をコンパイラで処理できるという非常に興味深い副作用があります。演算子から返される一時オブジェクトのケースは、その具体例の 1 つです。実のところ、新しい Delphi コンパイラの最新機能は、Delphi 2006 以降レコードに対して使用可能になっているのと同じ構文およびモデルでクラスの演算子を定義できることです。

メモ： どのコンパイラでクラスの演算子オーバーロードがサポートされているか
この言語機能は iOS ARM コンパイラの場合に働きますが、Mac 上の iOS シミュレータでも動作します。もちろん、このコードを従来のコンパイラで Windows や Mac 向けにコンパイルすることはできません。この機能を追加することは非常に難しいわけではありませんが、演算子によって結局は多数の一時変数が生じることになり、（ARC やガベージ コレクションのような）自動メモリ管理メカニズムがなければ、このアプローチはまったく意味をなしません。

一例として、以下の簡単なクラスについて考えてみましょう。

```
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;
```

このコードを前提として、このクラスを以下のように使用することができます。

```
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 インターフェイスとクラスの混在

これまでは、インターフェイス変数と標準のオブジェクト変数で異なるメモリ管理モデルが使用されていたことから、この 2 つのアプローチの混在（たとえば、インターフェイスおよびオブジェクト変数/パラメータを使用して同じメモリ内オブジェクトを参照するなど）を避けるように一般に言われていました。

ARC 対応の新しい ARM コンパイラでは、オブジェクト変数とインターフェイス変数の間で参照カウントが統一されているので、両者を容易に混在させることができます。これにより、ARC 非対応の Delphi プラットフォームよりも ARC 対応の Delphi プラットフォームの方がインターフェイスをより強力かつ柔軟に使用できるようになります。

4. 言語のその他の変更点

文字列型の変更とオブジェクト メモリ管理の他にも、新しい Delphi ARM コンパイラには、以下のように、たやすく導入に着手できる現行の変更点および予定されている変更点があります。

- 遅かれ早かれ、with 文は非推奨となり、Delphi 言語から削除されます。これを今すぐにコードから削除するのは簡単であり、このキーワードの隠れた落とし穴を考えると、大半の Delphi 開発者がこの構文の削除に賛同するでしょう。
- 何らかの自動メモリ管理に移行する際にはポインタを直接使用しないように勧められることから、ポインタの使用は制限または禁止されます。（内部的にはポインタに基づいている）TList ではなくジェネリック コンテナ クラスを使用するのは、Delphi RTL ライブラリに対して行われている移行の良い例であり、同じような変換を Delphi 開発者が自らのコードでも行うことを提案します。
- アセンブリ コードは新しいコンパイラに移植できず、ARM CPU に適用できないので、削除されます。
- レジスタまたは別個のスレッドで共有されない別の一時メモリ領域への値のコピーがコード生成時に最適化されないように、別のスレッドによって変更される可能性があるフィールドであることを示すために [volatile] 属性が使用されます。

5. RTL の考慮事項

このホワイト ペーパーでは、LLVM の紹介、コンパイラの変更点の説明と話を進めてきましたが、最後に、ランタイム ライブラリ（RTL）と、モバイル プラットフォームをサポートするための RTL の変更点に目を向けます。この最後のセクションはこれまで

のセクションほど詳細ではなく、モバイル プラットフォーム向けの Delphi アプリケーションへの移行や新規作成の際に（ユーザー インターフェイスでもデータベースでもネイティブ デバイス センサでもなく）ランタイム ライブラリに関して生じるおそれがある問題をいくつか取り上げ、その概要を説明するだけです。

つまり、この短いセクションでは、さまざまなオペレーティング システムをサポートするための一般的な提案、ファイル アクセスに関連する情報、パッケージおよびライブラリのサポート（あるいはその欠如）を一覧します。

5.1 RTL とクロスプラットフォーム

部分的に関連するいくつかのトピックに関する提案を以下に少し示します。

- 可能なら、つまりより高レベルのコンポーネントやクラスが使用可能なときは常に、API の直接呼び出しは避けます。これにより、コードが（現在または将来）別のプラットフォームに移植しやすくなります。

- クロスプラットフォーム ユニットを優先的に使用します。たとえば、ファイル管理の場合は、ファイル管理用の古い形式の Pascal ルーチンではなく、IOUtils ユニットのレコードおよびそれらのクラス メソッドを使用することを推奨します。IOUtils ユニットの具体例については、次のセッションでいくつか取り上げます。

- コードを Mac やモバイル デバイスに移植できるようにする場合は、Windows 流の考え方や Windows 固有の API はすべて避けます。たとえば、MSXML を使用せずに（モバイル プラットフォームでも動作する）Delphi ベースの ADOM XML 処理エンジンを使用するのは、この良い例です。もう 1 つ例を挙げれば、特定のソケット ライブラリや Web ライブラリを避け、Indy を使用します。言うまで

もなく、COM、ActiveX、ADO、BDE、その他の Windows 固有の Delphi 技術を使用するコードはモバイル プラットフォーム（または Mac）では動作しません。

- *Generics.Collections* ユニットに定義されているジェネリック コンテナ クラスを使用します。古い形式の Contnrs ユニットはモバイル プラットフォームでは使用できません。なぜなら、この古いユニットはポインタ リスト（ジェネリックでない *TList* クラス）に基づいており、ARC メモリ管理モデルでは正しく動作しないからです。

- 余談ですが、オブジェクトが各要素に関連付けられている TStringList の使用はサポートされてはいますが、型定義 *TDictionary<string,TMyClass>* にあるように文字列をキーとするジェネリック ディクショナリを使用することを強くお勧めします。これではるかにすっきりするだけでなく、ほとんどの場合、はるかに高速でもあります。ディレクトリではハッシュ テーブルを使用するからです。この点については、以下のサブセクションの 1 つで詳しく説明します。

- ポインタ ベースの構造体やメソッドが必要なネイティブ API を呼び出しているのでない限り、それらはすべて避けます。

この一覧がもっと長くなることは確実です。以下では、ファイル アクセスとライブラリに関して詳しい説明を加えます。

5.2 ファイル アクセス

Delphi のいくつかのバージョン以降、（フォルダ内を検索するための FindFirst や FindNext のような）従来の Pascal ファイル アクセス ルーチンの代わりに、より高レベルの一連のレコードが使用されるようになってきました。これらのレコードは IOUtils ユニット（入出力ユーティリティ）にまとめられています。IOUtils の使用は、コードをクロスプラットフォーム対応にするための推奨アプローチです。

このユニットには、主にクラス メソッドを定義している 3 つのレコードがあり、それらは対応する以下の .NET クラスと互換性があります。

- *TDirectory* は System.IO.Directory に対応します。
- *TPath* は System.IO.Path に対応します。
- *TFile* は System.IO.File に対応します。

TDirectory がフォルダの閲覧とその中のファイルおよびサブフォルダの検索のためのものであることは一目瞭然ですが、*TPath* と *TFile* の違いはそれほど明白でないかもしれません。*TPath* はファイル名とディレクトリ名の操作に使用されるもので、ドライブ、ファイル名（パスを含まない）、拡張子などを抽出するメソッドのほか、UNC パスを操作するメソッドも用意されています。*TFile* レコードはそれとは異なり、ファイルのタイム スタンプと属性の確認のほか、ファイルの操作、ファイルへの書き込み、ファイルのコピーにも使用できます。

たとえば、アプリケーションと一緒にファイルをモバイル プラットフォームに配置する際は、Windows でユーザーのドキュメントにアクセスするのと同じように、アプリケーションの "Documents" フォルダにアクセスできます（以下を参照）。

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

使用可能な機能の 1 つがフォルダとファイルの検索です。以下のコード断片では、指定された初期フォルダ（BaseFolder）内のフォルダを読み取り、そこから 1 レベルだけ下がって、サブフォルダ内のファイルを読み取っています。

```
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 文字列リストのジェネリック ディクショナリへの切り替え

長年にわたり、私自身も含めて多数の Delphi 開発者が *TStringList* クラスを多用してきました。このクラスは、普通の文字列リストや名前/値ペアのリストに使用できるだけでなく、オブジェクトのリストを文字列に関連付け、それらのオブジェクトを検索するのに使用することもできます。

Delphi ではジェネリックスを完全にサポートしていることから、この万能ナイフ的なツールの役割は、的を絞った特定のコンテナ クラス群にうまく置き換えることができます。たとえば、文字列キーとオブジェクト値を持つジェネリック ディクショナリは、一般に、コードの簡潔さと安全性の 2 点で文字列リストより優れています。ディレクトリがハッシュ テーブルを使用していることで、必要になる型キャストが少なく実行も速いからです。

これらの違いをはっきり示すために、以下の例を考えてみましょう。ここには、以下のように定義された同一内容のリストが 2 つあります。

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

これらのリストには、ランダムながら同一のエントリが、以下のコードの繰り返しで追加されます。

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

それぞれのリストに対して名前で検索を行ってリストの各要素を取得する 2 つのメソッドを使って、速度テストを行います。どちらのメソッドも文字列リストをスキャンして値を求めますが、第 1 のメソッドは文字列リスト内のオブジェクトを検索するのに対して、第 2 のメソッドではディクショナリを使用します。第 1 の場合は、指定された型に戻すのに型キャストが必要になりますが、第 2 の場合はディレクトリが既に指定の型に結び付いています。これら 2 つのメソッドのメイン ループは以下のとおりです。

```
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;
```

ディクショナリのハッシュ キーの場合と比べて、ソート済みの文字列リスト内で検索（二分探索を実行）するのに、どれくらいの時間がかかるのでしょうか。予想どおり、ディクショナリの方が速く、Windows プラットフォームで行ったテストの数値（ミリ秒単位）は以下のようになります（すべてのプラットフォームで似たような違いが現れます）。

```
TStringList: 2839
TDictionary: 686
```

初期値が同じなので結果は同じですが、時間はかなり異なります。ディクショナリの方は約 1/4 の時間しかかかっていません（テストは 100 万個のエントリを使って行いました）。

もちろん、この例とこの短いセクションでは、コンパイラ レベルでジェネリックスを組み込んだ後のジェネリック ディクショナリおよび Delphi RTL に追加された最近のデータ構造の威力を示したと言っても、氷山の一角を引っかいた程度にすぎません。繰り返しますが、これらの新しい構造体はどのプラットフォームでも良い選択肢になりますが、メモリ管理モデルの変更のために、iOS の場合にはなおさら適切な選択肢となります。

5.4 ライブラリとパッケージ

Delphi の強力な機能の 1 つは、ランタイム パッケージを使用して、アプリケーションをよりモジュール的に配置できることです。パッケージは "専用の" ダイナミック リンク ライブラリで、Windows では DLL、Apple プラットフォームでは dylib になります。Windows と OS X では（両者には少し違いはありますが）、ランタイム パッケージを共通フォルダに配置して複数のアプリケーションで共有することもできますし、特定のアプリケーション フォルダに配置して異なるプログラム間で発生するおそれのある競合を避けることもできます。

一方、iOS プラットフォームでは、このどちらのシナリオも選択できません。共有ライブラリを iOS の物理デバイスに配置することはできません（ただし、iOS シミュレータで使用することはできます）。それが、オペレーティング システム レベルで Apple 社だけができることだからです。また、ダイナミック ライブラリをアプリケーションに追加することもできません。メイン プログラムになることができるのは実行可能ファイルだけだからです。

このような制限はランタイム パッケージに関係したものではなく、事実上、もっと一般的なことです。たとえば、midas.dll や dbExpress ドライバなどの 標準 Delphi ライブラリは、アプリケーションの実行可能ファイルに静的にリンクされます。同じことは InterBase クライアント ライブラリの場合にさえ起こります。事実、コンパイラには、静的ライブラリへの参照を認識し、それらを最終的な実行可能ファイルにリンクする手段が用意されていますが、この技術的な問題については、このホワイト ペーパーでは取り上げません。

6. まとめ

LLVM アーキテクチャに基づいた初めての ARM 用 Delphi コンパイラのリリースで、このホワイト ペーパーで見てきたように、今 Delphi 言語は重大な転換期にあります。下位互換性を保つための努力は行われましたが、開発者の皆さんには、今後、新しい機能を十分に活用していただけるだろうと思っています。

このホワイト ペーパーで説明した言語上の変更点、特に、ARC のサポートは Delphi の将来を決定するものです。これらの変更点の一部は、新しいプラットフォームが原動力となり、また一部は、言語をより洗練されたものにし、新しく多くの場合に使い勝手の良い機能を Delphi に追加するためのものです。

執筆者について

Marco Cantu 氏は最近、Delphi プロダクト マネージャとして Embarcadero Technologies 社に入社しました。ベストセラーになっている『Mastering Delphi』シリーズ本の著者で、近年は Delphi のいくつかのバージョン（2007 ～ XE）に関するハンドブックを自費出版しています。

Marco 氏は会議などでたびたび講演し、Delphi に関する数え切れないほどの記事を執筆しています。また、以前は、世界中の企業で Delphi に関する上級クラス（Delphi による Web 開発など）の講師をよく務めていました。同氏のブログは http://blog.marcocantu.com でご覧いただけます。また、Twitter アカウントは @marcocantu、連絡先は marco.cantu@embarcadero.com です。

エンバカデロ・テクノロジーズについて

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