White Paper

embarcadero

모바일 개발을 위한 델파이 언어

마르코 칸투, 델파이 프로덕트 매니저

Embarcadero Technologies

박지훈.임프 번역

2013년 4월

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
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

서론

이 문서에서는 델파이의 "모바일" 버전과 새로운 델파이 ARM 컴파일러의 변경 사항들에 대해 소개합니다. 이 문서는 언어의 변경 사항들과 기존의 코드를 포팅하고 하위 호환성을 유지하는 데 사용될 수 있는 테크닉들에 초점을 둡니다.

필자: 마르코 칸투, 델파이 프로덕트 매니저, 엠바카데로 테크놀로지
(수정 및 추가 제안 사항들은 marco.cantu@embarcadero.com으로 보내주십시오)
이 문서는 앨런 바우어를 비롯한 많은 리뷰어들의 도움으로 작성되었습니다.

문서 리비젼: 1.0

1. 새로운 컴파일러 아키텍처

델파이를 모바일 ARM 플랫폼으로 옮기는 것은 델파이 언어에 있어 큰 진화입니다. 따라서, 엠바카데로의 R&D 팀은 엠바카데로의 언어들에 공통적으로 적용될 새로운 아키텍처를 적용하였습니다. "툴체인"이라고 하는 모든 관련된 툴들을 독자적인 방식으로 개발하는 대신, 업계에서 광범위하게 지원하고 있는 기존의 컴파일러와 툴체인 기반 기술을 활용하기로 결정했습니다. 이렇게 하면 미래에 시장이 요구하게 될 새로운 플랫폼 및 운영체제를 더 빠르게 지원할 수 있게 됩니다.

구체적으로 말하면, 새로운 세대의 델파이 컴파일러(C++빌더 컴파일러도 포함)는 LLVM 아키텍처를 이용합니다. LLVM이란 무엇이고 왜 이것이 중요할까요? LLVM에 대해 간단히 알아보고 우리의 주 주제로 다시 돌아오겠습니다.

1.1: LLVM 소개

LLVM 프로젝트에 대해서는 LLVM 메인 웹 사이트에서 자세히 알아볼 수 있습니다.

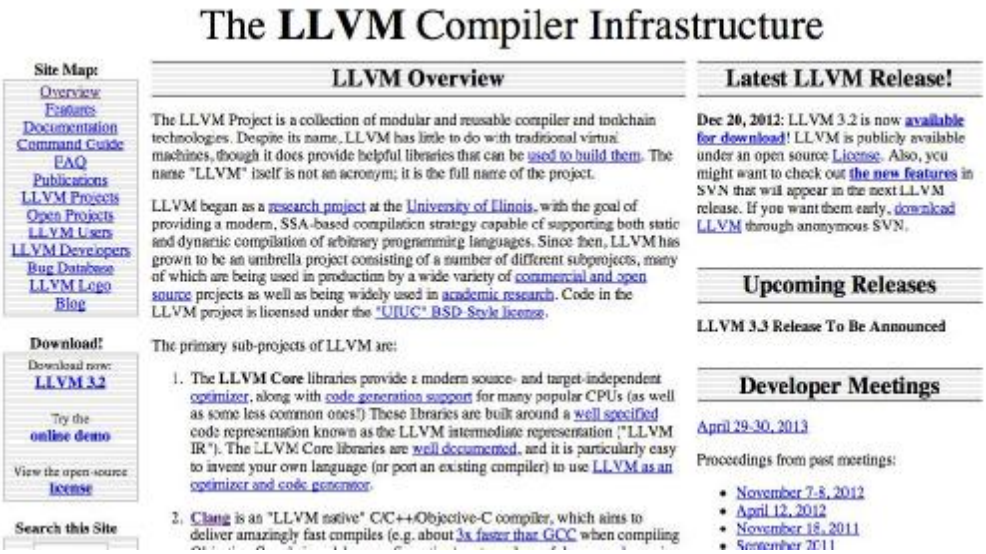
http://llvm.org

짧게 설명하자면, LLVM은 "모듈화되고 재사용 가능한 컴파일러 및 툴체인 기술들의 집합"입니다.

그 이름과 달리(LLVM이라는 이름은 처음에는 알파벳 이니셜 이름이었지만 이제는 프로젝트의 이름일 뿐입니다), LLVM은 가상 머신과는 거의 관련이 없습니다.

LLVM은 일리노이 대학교에서 연구 프로젝트로 시작되었으며, 그 목적은 어떤 프로그래밍 언어든 정적 혹은 동적 컴파일을 할 수 있는 현대적이며 SSA 기반의 컴파일 방식을 제공하는 데 있었습니다. 이후로 LLVM은 여러 서로 다른 서브프로젝트로 구성된 우산 프로젝트로 성장해왔고, 그 서브프로젝트들 중 많은 것들은 대단히 다양한 상용 및 오픈소스 프로젝트들의 개발 과정, 그리고 학술적 연구들에서 이용되어왔습니다.

LLVM의 메인 웹사이트에는 다양한 서브프로젝트들이 나열되어 있는 것을 볼 수 있습니다. LLVM 코어는 중간 코드 표현(representation)을 중심으로 개발되어있으며, 이것을 LLVM 중간 표현 (LLVM IR)이라고 부릅니다. 엠바카데로 같은 툴빌더들은 자사의 언어를 중간 표현으로 번역하는 컴파일러를 개발하고 이 중간 표현을 다시 CPU에 네이티브한 코드나 실행 중간 표현으로 "컴파일"하는 추가 툴들을 개발할 수 있습니다.



LLVM IR과 다른 비슷한 중간 표현들의 차이점은, LLVM은 프론트엔드와 백엔드를 분명하게 분리하는 것을 목적으로 한다는 것입니다. 따라서, 가상 실행 환경과 JIT 컴파일도 허용하기는 하지만, LLVM IR은 C/C++의 OBJ나 델파이의 DCU와 같은 중간 컴파일러 표현처럼 여길 수 있습니다(하지만 각 타겟 플랫폼을 위한 바이너리 코드를 생성하므로 OBJ나 DCU보다 추상적입니다). 이런 시나리오의 장점은, 일단 한 언어로부터 LLVM IR으로 컴파일하는 컴파일러를 개발하면 기존의 모든 백엔드들을 이용할 수 있다는 것으로, 여기에는 ARM도 포함됩니다. 이런 백엔드들은 CPU 벤더들로부터 고도로 최적화된 실행 파일을 생성할 수 있도록 잘 지원되고 있습니다.

마지막으로, LLVM 아키텍처가 오픈소스 툴들과 상용 툴들 업계에서 큰 영향력을 갖고

있다는 점도 언급할 필요가 있습니다. 일례로 애플도 Xcode에서 Mac OS X와 iOS 애플리케이션을 위해 LLVM과 Clang(C/C++/ObjectiveC를 위한 프론트엔드)를 이용하고 있습니다.

1.2: 델파이와 LLVM

위에서 설명한 것들을 고려하면, 델파이의 차세대 컴파일러 아키텍처가 아주 적합하다는 것은 분명할 것입니다. 기본적인 아이디어는, 델파이 소스 코드를 LLVM IR로 컴파일하여 iOS 및 안드로이드 ARM 컴파일러를 비롯한 여러 CPU 타겟들을 지원하는 것입니다. 이것은 운영체제 플랫폼들이 더욱 분화되고 과거보다 더 빠르게 바뀌어가고 있는 지금과 같은 상황에서는 더욱 중요해집니다.

따라서 델파이 iOS ARM 애플리케이션을 개발할 경우 새로운 컴파일러가 실행됩니다.

```
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.
```

물론 컴파일러가 기술의 중심이긴 하지만, 개발 환경에는 컴파일러 외에도 많은 것들이 있습니다. 예를 들어 델파이 IDE에는 ARM 애플리케이션들을 델파이 IDE 내에서 디버그할 수 있게 해주는 디버깅 툴들도 통합해야 합니다. IDE 외에도 델파이에는 컴파일러, 포팅 가능한 런타임 라이브러리, 윈도우 전용 및 플랫폼 독립적인 애플리케이션 프레임워크 등이 포함되지만, 이 문서에서는 컴파일러에만 초점을 둡니다.

LLVM 아키텍처와 그 컴파일러 백엔드들은 툴빌더들에게 메모리 관리와 코어 런타임 라이브러리(메모리, 쓰레드, 예외, 기타 저수준 언어 요소들)에 있어 특정 아키텍처를 요구합니다. 이런 요구는 강제적인 것은 아니므로 여러분은 LLVM과 다른 메모리 모델을 적용할 수도 있습니다.

모바일 플랫폼들에서는 LLVM이나 Java, .NET 같은 가상 실행 환경을 이용하고 가비지 컬렉션이나 자동 참조 카운팅(Automatic Reference Counting; ARC)을 적용하는 것이 일반적입니다. 많은 개발자들이 가비지 컬렉션에는 친숙하지만(좋은 쪽으로든 나쁜 쪽으로든), ARC는 훨씬 덜 알려져 있죠. Clang 프로젝트의 ObjectiveC에서의 ARC 사용에 대한 페이지를 참고하면 ARC에 대해 더 자세히 알아볼 수 있습니다.

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

LLVM에서 다른 메모리 관리 모델을 사용할 수 있지만, 적용할 만한 더 발전된 것들을 지원하고 있으며, 특히 리소스가 제한된 모바일 플랫폼들에서는 더욱 그렇습니다. 또한 메모리 관리를 더 자동화시키는 것은 새로운 개발자들이 델파이 언어를 받아들이기 쉽게 해주는 효과도 있습니다.

기존의 델파이 코드를 모바일 플랫폼으로 옮기려면 기존 코드를 다시 살펴봐야 하므로, 우리는 지금이 새로운 모바일 세상을 위해 델파이 언어를 더 진화시킬 좋은 시점이라고 생각했습니다. 이런 변화는 기본적으로 모바일 플랫폼들에만 적용되지만, 델파이 XE3에는 모바일용 명령들, 라이브러리 함수들, 컴파일러 디렉티브들이 이미 포함되어 있습니다. 이것이 이 문서의 핵심 주제입니다.

임프 역주: 이 문서에서 '클래식 델파이' 혹은 '클래식 컴파일러'라고 쓰는 것은 XE3 이전 버전들을 말하는 것이 아니라, 새로 추가된 모바일 컴파일러에 대비하여 윈도우 및 Mac용 델파이 컴파일러를 말합니다. 이 문서에서 설명하는 델파이 언어의 변경이 새로 추가된 모바일용 ARM 컴파일러에만 적용되고 윈도우, Mac 컴파일러에는 아직 적용되지 않았기 때문입니다. 하지만 이 문서에 따르면, 미래에는 윈도우와 Mac 컴파일러에도 새로운 컴파일러 변경 사항들이 적용될 가능성이 높습니다. 이런 이유로 필자는 '모바일 컴파일러'라고 부르기보다는 '새로운 컴파일러'라고 부르고 있습니다. (원문의 classic Delphi를 '전통적 델파이' 혹은 '기존의 델파이'라고 번역할 경우 이전 버전의 델파이를 가리키는 것으로 오인될 가능성이 높아 그냥 '클래식 델파이'라고 표기했습니다.

1.3: 왜 델파이 언어를 변경하는가?

우리의 목적은 델파이 언어를 진화시키면서도 하위 호환성을 최대한 유지하는 것이었습니다. 그러면, 왜 우리는 기존의 델파이 언어를 변경해야 했을까요? 기존의 코드를 이전할 때 이슈를 일으킬 수 있는 변경까지 포함해서 말이죠.

여기에는 아래와 같은 세 가지 주된 이유들이 있습니다. 자세한 내용은 이 문서 전체에 걸쳐 자세히 설명할 것입니다.

- 동일한 일을 하는 새로운 방법이나 기존 데이터 타입에 대한 새 변형을 위한 기능들을 계속 언어에 추가해나간다면, 델파이 언어는 지나치게 무겁고 복잡해질 것이며, 유지하고 새 플랫폼으로 포팅하는 일도 더 어려워질 것입니다.
- 동일한 기본 작업에 대해 여러 가지 방법을 제공하는 것은 언어를 처음 접하는 사용자들을 혼란시키기 쉽습니다.
- 델파이 언어 내에서 불일치한 부분이 있다는 것은 중요한 이슈입니다(또한 다른 언어들과 다르기 때문에 델파이 언어를 배우는 개발자들에게는 더욱 문제가 됩니다).

예를 들어, 클래식 델파이에서는 대부분의 데이터 구조들은 데이터 액세스에서 0 기반의 인덱스를 사용하는 데 반해, 문자열에서는 1 기반의 인덱스를 사용하여 서로 일치하지 않습니다. 델파이 개발자들은 자유롭게 배열 범위를 지정할 수 있지만 (전통적인 파스칼에서의 부분 범위 사용을 따른 것), 동적 배열의 경우엔 0 기반입니다.

델파이 언어에 새로운 개발자들을 끌어들이는 것도 우리의 목표들 중 하나이므로, 불필요한 소소한 장애물들을 없애는 것이 적절할 것입니다. 델파이 모바일이 제공하는 비즈니스 면과 기술적인 면의 이점들 덕분에, 의미 있는 수준의 새로운 개발자들이 유입될 것으로 기대하고 있습니다.

1.4: 어떤 것이 바뀌지 않는가?

그 답은 간단합니다. 거의 모든 것입니다! 클래스, 객체, 메소드, 상속, 다형성, 인터페이스, 제네릭 타입, 어노니머스 메소드, 리플렉션(혹은 RTTI) 등등.

그중에는 아주 오래된 기능들도 여전히 델파이 언어에 포함되어 있는 것들도 있습니다. 터보 파스칼에서부터 내려온 전역 함수 및 변수들의 사용 같은 것들입니다. 다시 말하지만, 새로운 컴파일러 아키텍처로 바뀌면서도 변경되지 않은 기능들이 압도적으로 많으므로, 여러분이 기존의 코드를 이용하고 기존의 델파이 개발자들이 기존의 지식을 그대로 유지할 수 있도록 해줍니다.

현재까지 여러분이 모바일 개발에 접근하려 할 때 일반적으로 여러분은 새로운 언어, IDE, RTL, 유저 인터페이스 라이브러리를 배워야 했지만, 델파이 XE4 개발자들은 최소한의 변경만으로 동일한 IDE, 언어, 라이브러리를 그대로 사용하는 사치를 누릴 수 있습니다. 이 문서의 초점은 이런 변경 사항들을 기술하는 데 있으며, 델파이 언어 기능들 대부분(그리고 여러분의 기존의 델파이 코드 대부분)이 새로운 모바일 플랫폼으로 포팅될 수 있으며 이미 크로스플랫폼이 지원되고 있다는 것을 인식하는 것이 중요합니다.

1.5 XE4의 델파이 컴파일러들

여기까지 읽으면서 눈치챘겠지만, 델파이 XE4에는 여러 종류의 컴파일러가 포함되어 있습니다. 데스크톱 플랫폼들을 지원하기 위한 세 가지의 컴파일러들, 맥 플랫폼에서의 iOS 시뮬레이터를 위한 컴파일러, 새로운 ARM 컴파일러 등입니다. 약간 혼란스러울 수도 있으므로 각각의 컴파일러들을 간단히 나열해봅니다.

- Win32 컴파일러 (DCC32)
- Win64 컴파일러 (DCC64)
- Mac 컴파일러 (DCCOSX)
- iOS 시뮬레이터 컴파일러 (DCCIOS32)
- iOS ARM 컴파일러 (DCCIOSARM)

이 다섯 가지 컴파일러들 중 마지막 컴파일러만이 LLVM 툴체인 기반이지만, iOS 시뮬레이터 컴파일러도 문자열 및 메모리 관리 설정 일부에서 LLVM 컴파일러의 설정을 이용합니다. 이에 대해서는 이후에 더 자세히 살펴보겠습니다.

2. 문자열 타입

문자열의 관리는 델파이 언어에서 더 중요한 변경들이 뒤따르는 영역입니다. 이 변경을 위한 몇가지 아이디어들이 있었습니다. 전통적인 델파이 모델(약간씩 다른 여러 문자열 타입들)의 단순화, 최적화 요구(일부는 모바일 플랫폼과 LLVM 플랫폼에서 요구되는 모델 때문), 그리고 델파이 언어를 다른 일반적인 프로그래밍 언어들과 맞출 필요 등입니다.

터보 파스칼 및 델파이 1 버전까지의 하위 호환성을 유지하는 일에는 많은 부담이 있었고, 새로운 개발자들은 물론 기존의 개발자들에게도 많은 도전을 제기하는 것이었습니다. 아래는 실제로 변경된 사항들로서, 각각에 대해 이해를 도와주는 코드 예제들과 마이그레이션과 향후의 크로스플랫폼 호환성 양쪽 모두를 위한 힌트들을 함께 제공합니다.

2.1: 단 하나의 문자열 타입

윈도우용 델파이의 최근 버전들에서는 기하급수적으로 문자열 타입들이 늘어났습니다. 델파이에서는 다음과 같은 문자열 타입들을 제공하고 있습니다.

- 파스칼 짧은 문자열(short string), 255 길이의 1바이트 문자들
- 참조 카운팅되는 copy-on-write AnsiString
- AnsiString에서 상속된 전용 타입들. UTF8String과 RawByteString처럼 문자열 타입 생성 메커니즘 기반
- C 언어 문자열(PChar)을 관리하기 위한 RTL 함수들

- 참조 카운트되는 copy-on-write 유니코드 문자열(UnicodeString). 현재 기본 문자열 타입이며 UTF16으로 구현
- COM 호환 와이드 문자열(WideString). UTF16 기반이지만 참조 카운트되지 않음

LLVM 기반의 새 델파이 컴파일러에는 단 하나의 문자열 타입만이 존재하며, 유니코드 문자열(UTF16)로서 델파이 XE3의 string 타입에 해당합니다(윈도우 컴파일러에서 UnicodeString 타입에 대한 별칭이기도 합니다). 하지만 이 새로운 문자열 타입은 이전과 다른 메모리 관리 모델을 사용합니다. 이 string 타입은 여전히 참조 카운트를 사용하지만, 향후 변경 불가(immutable)가 될 예정입니다. 변경 불가란 문자열이 생성된 후에는 그 내용을 변경할 수 없다는 의미입니다(해당 섹션에서 자세히 설명합니다). 요약하자면, 새 컴파일러에서 문자열은 *유니코드 기반*이며, *곧 변경불가(immutable)하게 될 것*이며, *참조 카운트*됩니다.

1바이트 문자열(ANSI 혹은 UTF8 문자열 등)을 다루어야 할 경우, 예를 들면 파일 시스템이나 소켓을 다루는 경우, 바이트의 동적 배열을 이용할 수 있습니다("1바이트 문자열 다루기" 섹션에서 자세히 다룹니다). 먼저 핵심 기능들을 보여주는 실제 코드를 보여주고 다른 코딩 스타일을 제시할 것입니다. 이후에 새 컴파일러에서 1바이트 문자열을 다루는 방법들을 보여드릴 것입니다.

여러분이 아직 AnsiString 타입, ShortString 타입, WideString 타입, 혹은 다른 특수 목적의 문자열 타입을 사용하고 있다면, 여러분의 전체 코드를 기본 string 타입으로 변경할 것을 강력하게 권합니다. (여기서 우리는 여러분이 모바일 플랫폼으로 마이그레이션 하려고 하는 코드에 대해 말하고 있는 것입니다. 윈도우 델파이 컴파일러는 당분간은 변경되지 않습니다)

노트: "특수 목적" 문자열 타입들은 주로 바깥 세계와의 인터페이스를 위한 것이었습니다. 단적인 예가 COM 지원을 위해 도입된 WideString 타입입니다. 이런 특수 타입들은 분명히 사용하기 편리하기는 하지만, 어쨌든 바깥 세상과의 인터페이스라는 단 하나의 목적만으로 약간씩 다른 의미와 동작을 가져와 델파이 언어를 "오염"시킨 것이 사실입니다. 델파이가 다른 많은 플랫폼들로 전진하고 있는 지금, 델파이에서 이런 이런 타입들을 그대로 유지하는 것은 일을 복잡하게 만들고 혼란스럽게 할 뿐입니다. 예를 들어 윈도우가 아닌 플랫폼에서 WideString이 어떤 의미가 있을까요? 각각의 플랫폼과 인터페이스하기 위한 이런 타입들에 대해서는 언어와 컴파일러에 전용 데이터 타입들로서 가지고 있기보다는, 메소드와 연산자를 가진 레코드나 제네릭, 기타 다른 언어 기능들을 이용하여 사용자 정의 타입으로 만드는 것이 훨씬 더 명쾌한 해결책입니다.

2.2: 참조 카운트되는 문자열

과거와 마찬가지로, 델파이 문자열은 참조 카운트됩니다. 이것은 동일 문자열에 두 번째 변수를 지정하거나 문자열을 파라미터로 전달할 경우 참조 카운트가 증가된다는 뜻입니다. 모든 참조가 스코프를 벗어난 즉시 해당 문자열은 메모리에서 삭제됩니다.

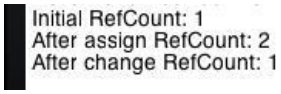
대부분의 개발자들에게 이것은 개발자가 문자열의 메모리 관리에 신경 쓰지 않아도 제대로 동작한다는 뜻입니다. 저수준 구현(공지 없이 변경될 수 있습니다만)에 대해 더 자세히 이해하고 싶다면, 다음의 자세한 내용을 읽어보면 되고, 관심이 없다면 다음 섹션은 그냥 넘어가면 됩니다.

여러분이 문자열 구현의 자세한 부분까지 파고들어보고 싶다면 StringRefCount 함수로 문자열의 참조 카운트를 알아낼 수 있습니다(델파이 2009에서 추가). 아래 코드에서처럼 사용하면 됩니다.

```
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;
```

이 코드를 실행해보면, str2의 참조 카운트는 처음에는 1이었다가 str1에 대입한 후에는 2로 증가하는 것이 보일 것입니다. 그리고 str2의 값을 바꾼 후에는 다시 1이 되는데, 이것은 str2가 변경되었기 때문에 공유되었던 문자열 데이터를 str2로 복사 동작이 일어났기 때문입니다(다음 섹션에서 이 copy-on-write 메커니즘에 대해 설명). 이 코드는 윈도우, 시뮬레이터, 모바일 디바이스 어디에서 실행하든 동일한 결과(1, 2, 1)가 나옵니다.



한가지 알아둘 것은, 함수나 메소드를 사용할 때 문자열을 const 파라미터로 전달하면 그 참조 카운트는 바뀌지 않으며 해당 코드가 더 빠르게 동작하게 됩니다(겨우 몇 CPU 사이클 정도입니다만). 참조 카운트를 리턴하는 함수 StringRefCount 내부에서도 이렇게 하고 있으며, 그렇게 하지 않았다면 호출할 때마다 이 함수 자체로 인해 참조 카운트가 1만큼 증가해버릴 것입니다.

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

요약하자면, 참조 카운팅은 클래식 컴파일러와 새 컴파일러가 매우 비슷한 방식으로 동작하며, 이것은 아주 효율적인 구현입니다.

2.3: COPY-ON-WRITE와 변경불가 문자열

임프 역주: 델파이 XE4 기준, 현시점에서 델파이 문자열은 변경불가(immutable)가 아닙니다. 마르코 칸투의 이 페이퍼와 엠바카데로의 공식 헬프 문서 양쪽 모두 XE4에서 문자열이 이미 변경불가라고 하기도 하고 향후 변경불가가 될 예정이라고 하기도 해서 혼란스럽게 하는데, 종합해보면 XE4 버전은 아직 변경불가 문자열이 적용되지 않은 것으로 보입니다.

기존의 문자열의 내용을 변경할 때는 달라지기 시작하는데, 이것은 문자열을 새로운 값으로 바꿔치울 때가 아니라(이런 경우엔 완전히 새로운 문자열을 갖게 됩니다) 아래 코드처럼 문자열의 요소를 변경할 때입니다(이전 섹션의 예에서도 마찬가지).

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

모든 델파이 컴파일러들은 copy-on-write 방식을 이용합니다. 여러분이 변경하는 문자열이 하나보다 많은 참조를 가지고 있을 경우, 일단 먼저 복사되고(관련된 문자열들의 참조 카운트가 조정됨) 이후에 수정됩니다.

노트: 이 용어에 익숙하지 않은 분들을 위해 설명하자면, "copy-on-write"는 문자열에 새 문자열 변수를 대입할 때는 복사가 일어나지 않으며, 그 내용이 변경될 때만 복사가 일어나는 것을 의미합니다. 다른 말로 하자면, 여러분이 복사 동작을 할 때 복사가 일어나는 것이 아니라 차후에 실제로 복사가 필요한 때에만 일어난다는 의미입니다. 내용 변경이 일어나지 않으면 복사 동작도 일어나지 않습니다.

새 컴파일러는 클래식 컴파일러와 매우 비슷하게 동작합니다. 문자열의 참조가 하나뿐인 경우가 아니라면 copy-on-write 메커니즘으로 구현되며, 참조가 하나뿐인 경우에는 기존의 문자열이 변경됩니다. 예를 들어 아래 코드에서는 데모 소스 코드에 있는 *StrMemAddr* 함수를 이용하여 실제 문자열의 메모리 위치를 보여줍니다.

```
procedure TTabbedForm.btnCopyClick(Sender: TObject);
var
  str3, str4: string;
begin
  // 문자열을 정의하고 별명을 만듭니다
  str3 := Copy('Hello world', 1);
  str4 := str3;

  // 메모리 위치를 보여줍니다
  Memo1.Lines.Add(str3 + ' - ' + StrMemAddr(str3));
  Memo1.Lines.Add(str4 + ' - ' + StrMemAddr(str4));

  // 하나를 변경함 (다른 쪽은 변경하지 않음)
  str3 [High(str3)] := '!';
  Memo1.Lines.Add(str3 + ' - ' + StrMemAddr(str3));
  Memo1.Lines.Add(str4 + ' - ' +
StrMemAddr(str4));

  // 첫번째를 다시 변경함
  str3 [5] := '!';
  Memo1.Lines.Add(str3 + ' - ' +
StrMemAddr(str3));
  Memo1.Lines.Add(str4 + ' - ' +
StrMemAddr(str4));
end;
```

오른쪽 그림은 위 코드의 실행 결과입니다 (iOS 시뮬레이터). 두 문자열 중 하나(*str4*)는 전혀 영향을 받지 않았으며, 다른쪽은 첫번째 쓰기 작업에서만 재할당된 것을 확인할 수 있습니다.

문자열을 변경할 수 있다면, 클래식 컴파일러와 다른 점은 무엇일까요? 현재의 문자열 조작 구현은 클래식 구현과 병행하고 있지만, 미래에는 바뀔 예정입니다. 변경불가 문자열(immutable string)은 더 나은 메모리 관리 모델을 제공합니다. 변경불가 문자열을 이용하면 문자열의 요소를 직접 변경할 수 없게 되며 더 느려지게 되지만, 반면 문자열 이어붙이기는 더 빨라집니다.

따라서 현재의 델파이 ARM 컴파일러에서는 이런 동작이 허용되고 있지만, 내부 문자열 관리 구현이 향후 바뀔 예정이며 따라서 이 동작이 허용되지 않게 될 수 있습니다.

코드가 최적화되지 않을 수도 있다거나 향후 버전에서 동작하지 않을 수도 있다는

경고(warning)를 발생시키기를 원한다면, 그 전용 경고를 켤 수 있습니다(델파이 XE3에서도 이미 클래식 컴파일러에 추가되어 있습니다). 해당 지시어는 *{$WARN IMMUTABLE_STRINGS ON}* 이며, 이 지시어를 지정한 상태에서 아래와 같은 코드를 사용하면,

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

다음과 같은 경고를 받게 될 것입니다.

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

변경불가 문자열로의 변화가 여러분의 코드에 어떤 영향을 미치게 될까요? 다음의 코드와 같은 단순한 문자열 이어붙이기의 경우 별다른 문제점이 없습니다. (이어붙이기 성능의 향상은 여러 목표들 중 하나이므로 향후로도 그럴 것입니다)

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

다른 개발환경과 달리, 우리는 문자열 이어붙이기가 훨씬 느려지거나 어떤 식으로든 금지되는 것을 기대하지 않습니다. 이슈가 될 수 있는 것은 오직 문자열의 각각의 문자를 변경하는 경우뿐입니다. 현재 R&D 팀에서 이 방향으로 진행중인 연구에서는 문자열 이어붙이기나 Format 류의 함수들 같은 일반적인 작업들을 최적화하는 방법을 찾고 있는 중입니다.

어떤 경우든, 플랫폼마다 다른 문자열 수정 및 이어붙이기의 구현 방식으로부터 자신을 보호하려면, 구현 방식과 컴파일러로부터 자유로운 문자열 생성 코드를 사용하는 편이 좋을 수 있습니다. 예를 들면 TStringBuilder 클래스가 그렇죠.

2.4: *TStringBuilder* 클래스의 사용

미래를 위한 주요한 옵션들 중 하나는, 여러 조각들로부터 문자열을 만들어낼 때 TStringBuilder와 같은 문자열 이어붙이기 클래스를 이용하는 것입니다. 각각의 문자나 작은 문자열들을 이어붙여 문자열을 생성한다면, 윈도우와 모바일 컴파일에서 높은 속도를 내려고 한다면 여러분의 코드를 *TStringBuilder*를 이용하도록 바꾸는 것이 좋은 방법입니다.

아래는 각각 일반적인 문자열 이어붙이기와 *TStringBuilder* 클래스를 이용하여 동일한 순환문을 작성한 예제 코드입니다.

```
const
  MaxLoop = 2000000; // 2백만

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;
```

클래식 델파이에서는 두 방식의 실행 속도가 매우 비슷합니다(네이티브 이어붙이기가 약간 빠르게 나왔는데, 위의 코드에서 스트링빌더에는 최종 크기를 미리 할당했기 때문입니다. 미리 할당해두지 않을 경우에는 네이티브 이어붙이기가 10%에서 20% 정도 더 빠릅니다).

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

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

iOS 시뮬레이터에서도 윈도우의 경우와 거의 비슷하며, 이것은 Mac 인텔 프로세서에 맞게 컴파일되기 때문입니다. 결과는 위의 결과와 아주 유사하게 나옵니다.

모바일 플랫폼(ARM 컴파일러와 물리적 모바일 디바이스)에서는 단순 이어붙이기는 상당히 느려질 것이 예상되며, 따라서 상당히 큰 문자열에 대해서는 *TStringBuilder* 만이 유일한 방법이 됩니다. 혹시 이런 말을 들어본 적이 있는 것 같다면, 아마도 마이크로소프트의 .NET 플랫폼 등의 매니지드 플랫폼들의 경우와 아주 비슷하기 때문일 것입니다.

문자열들이 재할당되어야 하는 것은 사실이지만, 메모리 매니저는 실행 속도에 대한 영향을 최소화할 수 있을 만큼 영리합니다. (지수적인 경우보다는 선형적인 경우에 더욱 그렇습니다)

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

위의 숫자들을 눈여겨보면, *iOS ARM* 디바이스에서는 *TStringBuilder를 이용하는 것이 거의 두배나 빠르며*, 그것도 밀리초 차이가 아닌 초 단위입니다! 아래는 물리적 디바이스에서 캡쳐한 실제 결과입니다.

TStringBuilder 클래스는 델파이 2009에서부터 지원되어왔기 때문에 델파이 구버전으로 개발해 유지보수하고 있는 애플리케이션에도 바로 적용할 수 있습니다. 네이티브와 모바일 플랫폼의 처리 속도의 차이 때문에 제 맥북 프로와 iOS 디바이스에서 속도 차이가 매우 크게 나타났습니다.

다른 예제로, 문자열의 모든 요소를 스캔하여 그중 일부를 치환하는 아래의 루프를 살펴봅시다.

```
// 문자열을 순환하면서 조건에 따라 일부를 바꿉니다
for I := Low (str1) to High (str1) do
  if str1 [I] = 'a' then
    str1 [I] := 'A'
```

변경불가 문자열이 적용된다면 위 코드는 매우 느려질 것으로 예상할 수 있을 것입니다(물론 큰 문자열의 경우). 대신 다음과 같은 코드를 사용할 수 있습니다.

```
// 문자열을 순환하면서 결과를 스트링빌더에 추가합니다
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;
```

결과를 보면, 위의 직접 문자열 요소 치환 코드가 *TStringBuilder* 코드보다 10배 가량 더 빠릅니다. 사실 다음번 대문자 A를 찾고 그 앞의 문자열 부분을 복사하는 방식으로 알고리즘을 훨씬 더 최적화할 수도 있습니다. 하지만 단순무식 알고리즘(문자열의 각 요소를 일일이 검사하는)과 비교하자면, 현재의 구현에서는 앞의 단순한 코드가 더 빠릅니다. 이것은 향후로는 바뀔 수도 있는데, 변경불가 문자열의 구현이 문자열 구조에 대한 직접 변경을 금지하게 될 수도 있습니다.

노트: 다시 한번 말하지만 문자열의 내부 구현이 향후에 바뀔 수도 있다는 점을 염두해두시길 바랍니다. 다른 언어들 중에는 큰 문자열을 문자들의 하나의 연속이 아닌 문자열 조각들의 집합으로 나타내는 언어들도 있습니다. 오늘 작성한 최적화 방식이 향후에는 더 느린 구현이 될 수도 있습니다.

2.5: 0 기반 액세스

두번째 변화는, 0 기반(0-based)의 문자열에 대한 지원입니다. 이 말은, 문자열의 요소(문자)들을 대괄호와 인덱스로 액세스할 때 1이 아닌 0부터 시작한다는 것입니다. 1 기반의 문자열을 사용해온 것은 파스칼 언어의 오랜 전통으로 내려온 관례죠. 하지만 1 기반의 문자열을 사용하도록 결정했었던 이유는, 가독성(readability)의 문제이거나 요소를 세기에 더 자연스럽다거나 하기보다는, 구현상의 결정의 결과입니다. 다르게

말하면, 구현상의 필요 때문에(NULL 터미네이터를 사용하는 C와 달리 문자열의 길이를 저장하기 위해 0번째 바이트를 사용할 필요가 있었음) 규정으로 정했던 것입니다.

물론 어떤 것이 더 자연스러운지에 대해 장시간 논쟁을 벌일 수도 있지만, 대부분의 언어들은 0 기반의 문자열을 사용하고 있고, 현실적으로 델파이에서도 다른 모든 데이터 구조들도 0 기반입니다. 동적 배열, 컨테이너 클래스들, *TStringList* 같은 RTL 클래스들, VCL 및 FireMonkey에서 하위 요소를 가진 클래스들(메뉴 아이템, 리스트박스 아이템, 서브컨트롤...)등이 그렇죠.

이것은 델파이 언어에 대한 중요한 변경이며(새 컴파일러의 아키텍처나 모바일 지원과는 무관), 많은 코드들에 영향을 미칠 수 있기 때문에, 새로운 컴파일러 지시어 *$ZEROBASEDSTRINGS*으로 바꿀 수 있도록 했습니다(델파이 XE3에 이미 추가되어 있음). 델파이 XE3에서 이 지시어의 디폴트는 off이며 XE4의 모바일 컴파일러에서는 on입니다. 하지만 이것은 컴파일러 지시어이기 때문에 델파이 XE3에서 이 지시어를 켤 수도 있고(on) 반대로 모바일 컴파일러에서 (0 기반 문자열로 코드 마이그레이션을 완료할 때까지) 기존 코드가 동작하도록 하기 위해 이 지시어를 끌 수도 있습니다(off).

노트: 이에 해당하는 프로젝트 옵션 수준에서 지정할 수 있는 Extended Syntax 컴파일러 지시어도 있으며, 모든 유닛에 적용되게 됩니다(다른 로컬 세팅이 없는 경우). 이 옵션을 사용하는 경우 프로젝트에 해당하는 문자열 액세스 모델로 작성된 유닛들만 포함하도록 주의해야 합니다. 이런 경우 유닛 레벨의 로컬 세팅이 불일치를 피할 수 있도록 도와줄 수 있습니다.

이와 관련된 알아두어야 할 몇가지 중요한 사항들이 있습니다.

- 문자열의 내부 구조는 영향을 받지 않습니다. 하나의 프로젝트에서 이 지시어로 서로 다른 설정을 가진 유닛들을 섞어 쓰거나 다른 방식으로 컴파일된 함수에 문자열을 넘겨줄 수도 있습니다. 특정 소스코드 라인에서 컴파일러가 [] 표현을 어떻게 해석하든 무관하게 문자열은 그대로 문자열일 뿐입니다.
- 클래식 문자열 RTL 함수들은 기존의 방식대로 동작할 것이며, *$ZEROBASEDSTRINGS*가 off라면 문자열 요소에 대해 1 기반의 위치를 이용합니다. 하지만 클래식 문자열 RTL 함수들은 하위호환성을 위해 유지될 뿐이므로 그런 함수들을 멀리할 것을 권장합니다. 새로운 *TStringHelper* 함수들(다음에 설명합니다)을 사용할 것을 권합니다.
- 새로운 고유 타입 헬퍼 *TStringHelper*의 함수 셋을 사용하여 기존 코드를 마이그레이션할 것을 권장합니다. 이 헬퍼 클래스는 다음 섹션에서 자세히

설명하며 모든 컴파일러에서 0 기반의 문자열 인덱스를 사용합니다(즉 윈도우, Mac, 모바일에서 동일하게 동작합니다).

다음의 간단한 코드를 살펴봅시다.

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

델파이 XE3의 디폴트 상태에서 버튼 캡션은 *"HOllo foo"*가 됩니다. 하지만 이 코드의 앞에 다음의 지시어를 추가하면,

```
{$ZEROBASEDSTRINGS ON}
```

버튼 캡션은 *"HeOlo foo"*가 됩니다. 이 경우 인덱스가 0 기반이기 때문에 문자열의 요소 2는 세번째 요소입니다. 두 컴파일러들 사이의 차이점은 다음과 같은 코드가 어떻게 해석되느냐입니다.

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

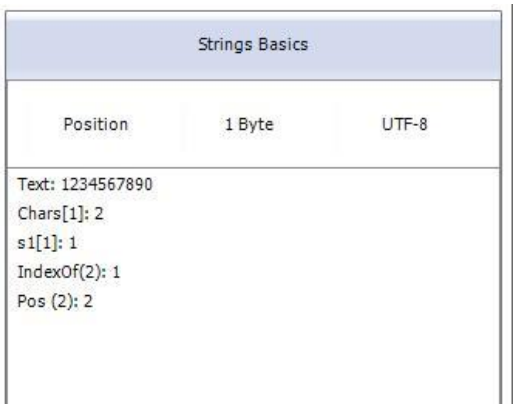
위 라인의 실제 동작은 *$ZEROBASEDSTRINGS* 컴파일러 지시어에 따라 결정되며, 두 컴파일러에서 디폴트 값이 다릅니다. 우리는 여러분의 코드를 새로운 모델(0 기반 문자열)로 바꿀 것을 강력히 권하며, 델파이 XE3 윈도우 및 맥 애플리케이션에 대해서도 가급적 그랬으면 합니다. 단일 문자열 인덱스 모델로 옮기면 향후 코드 가독성에 확실히 도움이 될 것입니다.

노트: 혼란의 여지가 많으므로 다시 설명합니다만, 컴파일러가 대괄호 문자 요소 액세스 연산자를 해석하는 방법은 문자열의 내부 구조와는 전혀 관련이 없다는 것을 알아두십시오. 다른 말로 하자면, 문자열은 내부적으로 이전과 완벽하게 동일하며 $ZEROBASEDSTRINGS 지시어를 켜든 끄든 전혀 다르지 않게 동작합니다.
컴파일러가 [i] 코드를 해석하는 방법만 달라진 것이며, 따라서 프로그램 내에서 서로 다른 설정으로 컴파일된 유닛 및 함수들을 섞어 쓸 수 있습니다. 함수에 0 기반의 문자열을 넘겨주는 것이 아니라 단지 특정 컴파일러 설정을 사용하는 코드로 혹은 코드로부터 문자열을 넘겨주는 것일 뿐입니다.

이것이 중요한 만큼, 아래 코드의 결과를 통해 잠재적인 이슈들을 살펴봅시다.

```
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)));
```

디폴트 상태에서, 시뮬레이터와 실제 디바이스에서는 윈도우 및 맥과 다른 결과가 나옵니다. 아래는 시뮬레이터의 스크린샷과 윈도우에서의 스크린샷입니다.



iOS에서 실행한 결과는 2/2/1/2이지만, 윈도우에서는 2/1/1/2로 나왔습니다. 유일한 차이는 두번째 값이며, 대괄호를 이용하여 직접 액세스를 한 경우입니다. 하지만 $ZEROBASEDSTRINGS 지시어를 이용하면 각 플랫폼에서의 동작을 뒤집을 수도 있습니다. 윈도우에서 *$ZEROBASEDSTRINGS ON* 으로 지정하면 결과는 2/2/1/2으로 나오고, iOS에서 *$ZEROBASEDSTRINGS OFF* 으로 지정하면 결과가 2/1/1/2으로 나옵니다.

대부분의 경우 이 문제를 피할 수 있지만, 특정 위치의 문자열 요소를 액세스할 필요가 있는 경우라면 Low() 함수(문자열의 하위 경계를 리턴)의 값을 따르는 상수를 정의할 수 있습니다.

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

이 상수의 값은 2 혹은 3이 되는데, 이를 통해 문자열의 세번째 문자를 액세스할 수 있게 됩니다. 이 상수를 정의한 후로는 세번째 문자를 다음과 같은 방법으로 액세스할 수 있습니다.

```
s1[thirdChar]
```

문자열의 요소들을 반복해야 하는 경우도 비슷한 상황입니다. 예를 들면 전통적인 루프를 다음과 같은 방법으로 대체할 수 있습니다.

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

  // 0 기반 문자열일 경우의 for 문
  for I := 0 to Length(S) – 1 do
    use(S[I]);

  // for-in 루프 (델파이 2006 이후부터 가능)
  for ch1 in S do
    use (ch1);

  // Chars를 이용한 전통적인 for 문 (모든 컴파일러 설정에서 가능, XE3 이후 버전)
  for I := 0 to S.Length - 1 do
    use(S.Chars[I]);

  // Low 및 High 함수로 융통성 있는 경계값을 지정 (XE3 이후 버전에서 동작)
  for I := Low(S) to High(S) do
    use(S[I]);
```

노트: Low 및 High

Low(s)는 0 기반 문자열인 경우 0을 리턴하며 1 기반 문자열에서는 1을 리턴합니다. High(s)는 0 기반 문자열인 경우에는 Length(s) - 1을 리턴하고 1 기반 문자열에서는 Length(s)를 리턴합니다. 빈 문자열이 파라미터로 전달된 경우, Low는 동일한 값을 리턴하지만 High는 0 기반 문자열에서는 -1을, 1 기반 문자열에서는 0을 리턴합니다. string 데이터 타입을 Low에 전달하면 현재의 설정을 알아낼 수 있습니다. 반면 타입을 High에 넘기는 것은 아무 의미가 없습니다.

2.6: TSTRINGHELPER 고유 타입 헬퍼 사용하기

델파이 XE3에서부터 지원하기 시작한 또다른 접근 방법은, 문자열 데이터 타입에 대한 헬퍼입니다. 이것은 XE3에서 도입된 새로운 언어 기능으로, 기존의 클래스와 레코드뿐만 아니라 네이티브 타입에도 사용자정의 메소드를 추가할 수 있게 해주는 기능입니다. 그 문법은 델파이 개발자들에게 약간 생소합니다. 다음은 사용자정의 타입을 이용한 예제입니다.

```
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;
```

델파이 XE3에는 이 언어 기능과 함께 문자열 타입을 위한 실용적인 구현으로 *TStringHelper* 헬퍼가 추가되어 있습니다. *TStringHelper*는 SysUtils 유닛에 정의되어 있으며 *Compare, Copy, IndexOf, Substring, Length, Insert, Join, Replace, Split* 등등을 제공합니다. 예를 들면 아래와 같은 코드를 작성할 후 있습니다.

```
procedure TForm4.Button1Click(Sender: TObject);
var
  s1: string;
begin
  // 변수에 대해
  s1 := 'Hello';
  if s1.Contains('ll') then
    ShowMessage(s1.Substring(1).Length.ToString);
  // 상수에 대해
  Left := 'Hello'.Length;
  // 메소드 체이닝
  Caption := ClassName.Length.ToString;
end;
```

이들 모든 메소드들(*Chars* 인덱스 속성을 포함)은 관련 컴파일러 지시어의 값이

무엇이든 0 기반의 표기를 사용한다는 것을 알아둡시다.

노트: 위 코드에서는 메소드 체이닝을 사용하고 있습니다. 메소드 체이닝이란 해당 타입의 객체에 적용되어 자신 객체를 리턴하거나 동일 타입의 다른 객체를 리턴하도록 메소드들을 정의한 것입니다.

2.7 1바이트 문자열 다루기

2.1 섹션에서 말했던 대로, 모든 1 바이트 문자열들은 델파이 ARM 컴파일러에서 지원되지 않습니다. 하지만 그렇다고 해서 1 바이트 문자열 데이터를 다룰 수 없다는 말은 물론 아니며, 단지 네이티브 데이터 타입으로는 할 수 없다는 뜻입니다. 실질적으로 따져보면, *AnsiString*, *AnsiChar*, *PAnsiChar* 등의 데이터 타입을 사용할 수 없게 됩니다.

예를 들어, 유니코드 UTF8 형식을 사용할 필요가 있다고 가정해봅시다. UTF8 파일을 읽을 경우에는 인코딩을 지원하는 *TTextReader* 인터페이스를 바탕으로 더 고수준의 접근 방법을 이용할 수 있습니다.

```
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);
```

이 방식은 작성하기도 더 쉽지만 1 바이트 UTF8 문자열을 직접 다루는 동작을 숨깁니다. 다음과 같은 더 복잡한 저수준 코드를 이용하여 동일한 동작을 할 수도 있으며, 이 코드에서는 파일 읽기의 내부 동작이 나타납니다.

```
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);
```

이런 데이터 구조를 메모리에서 다루기를 원하는 경우가 있을 수도 있겠습니다(예를 들면 직접 데이터 조작을 하기 위해 저수준 함수 호출을 하는 경우). 이런 경우 우리는 바이트의 동적 배열을 사용할 것을 강력히 권합니다. 배열을 자신만의 데이터 구조로 감싸서 현재의 동작 방식과 비슷하게 보이도록 할 수도 있습니다.

```
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;
```

이 레코드는 연산자 오버로딩을 이용하며, *TUTF8Encoding* 클래스를 기반으로 구현할 수 있습니다. *TUTF8Encoding* 클래스는 UTF8 바이트 배열을 UTF16 문자열로, 혹은 그 반대로 변환하는 준비된 메소드들을 제공합니다.

이런 레코드를 작성하면 내장된 *UTF8String* 타입이 존재하지 않는 델파이 ARM 컴파일러에서도 다음과 같은 코드를 컴파일할 수 있게 됩니다.

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

3. 자동 참조 카운트

"긴 문자열"(AnsiString)이 도입된 델파이 2 버전 이후로 델파이에는 참조 카운트에 기반한 메모리 관리 방식이 있었습니다. 이 문서의 앞에서 설명했던 것처럼 문자열은 참조 카운트를 이용하며 모든 참조가 스코프를 벗어나면 메모리에서 제거됩니다. 델파이 3부터는 객체를 가리키기 위해 인터페이스 타입의 변수를 이용하는 경우에 한해 객체에 대해서도 부분적으로 참조 카운트가 지원되게 되었습니다. 가장 마지막에는 동적 배열도 참조 카운트를 이용하게 되었습니다.

따라서 델파이 업계에서 참조 카운트는 새로운 것은 아닙니다만, 델파이 ARM 컴파일러에서는 처음으로 모든 클래스와 객체에 대해 자동 참조 카운트를 완전하게 지원하게 되었습니다. 자세한 내용으로 들어가기 전에 이 주제에 대한 간단한 소개부터 해보겠습니다.

자동 참조 카운트(Automatic Reference Counting; 이하 ARC)란 무엇일까요? 앞서 LLVM에 대한 섹션에서 링크했던 페이지에서 볼 수 있듯이, ARC란 더 이상 필요하지 않는 객체를 명시적으로 해제할 필요 없이 객체의 생존기간을 관리하는 방법들 중의 하나입니다. 객체(예를 들어 지역 변수)에 대한 참조가 스코프 바깥으로 나가면 해당 객체가 자동으로 파괴됩니다. 델파이는 이미 문자열, 그리고 인터페이스 타입 변수로 참조되는 객체에 대해서는 참조 카운트를 지원해왔습니다. 따라서 객체에 대해 말하자면 윈도우 델파이에서 ARC와 가장 가까운 것은 인터페이스입니다. (하지만 ARC는 클래식 델파이에서 인터페이스 타입 변수를 이용해서는 쉽게 해결하기 어려운 순환 참조 같은 이슈를 해결하기 위한 더 많은 유연성을 가지고 있습니다. 잠시 후에 설명하겠습니다)

가비지 컬렉션(GC)와 달리, ARC는 확정적이고, 객체들은 애플리케이션 흐름 내에서 생성되고 파괴되며, 별도의 백그라운드 쓰레드에 의해 일어나는 것이 아닙니다. 이 방식에는 장점과 단점이 모두 있지만, GC와 ARC를 구체적으로 비교하는 것은 이 문서의 범위를 많이 벗어나므로 여기서는 다루지 않습니다.

노트: 어떤 컴파일러에 ARC가 적용되었나?
LLVM 기반의 새로운 컴파일러는 기본적으로 ARC를 이용하지만, 클래식 컴파일러 아키텍처에 기반하고 있는 iOS 시뮬레이터에 사용되는 컴파일러도 ARC를 이용한다는 것을 알아두십시오(기술적으로는 인텔 CPU 및 맥 OSX 운영체제를 위한 컴파일러입니다만). 따라서 시뮬레이터와 디바이스의 메모리 관리는 일치합니다.

3.1: ARC 코딩 스타일

새로운 컴파일러가 참조 카운트를 지원하므로, 한 메소드 내에서 임시 객체를 참조할 때 메모리 관리를 완전히 무시해버림으로써 코드를 크게 단순화시킬 수 있습니다.

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

저는 테스트를 위해 *TMySimpleClass*에 소멸자를 추가하여 폼에 로그를 남기도록 했습니다(제 데모에서는 메소드 자체에서 로그를 남기도록 했습니다만 여기서는 생략했습니다). 프로그램이 end 문을 만날 때, 즉 *MyObj* 변수가 스코프를 벗어날 때 객체의 소멸자가 호출됩니다.

어떤 이유로든 메소드의 끝에 이르기 전에 해당 객체를 그만 사용하고 싶을 경우에는, 그 변수를 *nil*로 설정하면 됩니다.

```
class procedure TMySimpleClass.SetNil;
var
  MyObj: TMySimpleClass;
begin
  MyObj := TMySimpleClass.Create;
  MyObj.DoSomething(False); // True이면 예외 발생
  MyObj := nil;
  // 다른 작업들 진행
end;
```

이 경우, 해당 객체는 메소드의 끝에 이르기 전에 파괴되며, 정확하게 우리가 해당 변수를 nil로 설정한 지점에서 일어납니다. 하지만 try-finally 블럭이 없는 이 코드의 *DoSomething* 프로시저 내에서 예외가 발생하면 어떻게 될까요? 그런 경우에는 nil을 대입하는 문장을 건너뛰겠지만, 그래도 해당 객체는 메소드가 종료될 때 파괴됩니다.

요약하자면, 참조 카운트 동작은 객체를 변수에 대입할 때와 변수가 스코프를 벗어날 때 일어나며, 이것은 스택 기반의 지역 변수이든 컴파일러에 의해 추가된 임시 변수이든 다른 객체의 필드이든 무관합니다. 파라미터에 대해서도 마찬가지입니다. 객체를 함수에 파라미터로 넘기면 객체의 참조 카운트는 증가되고 함수가 종료되어 리턴되면 감소됩니다.

노트: 파라미터 전달을 최적화하려면

문자열과 마찬가지로 const를 이용하여 파라미터 전달을 최적화할 수 있습니다. 상수로 전달되는 객체는 참조 카운트 오버헤드를 일으키지 않습니다. 윈도우에서 객체 파라미터에 const를 사용하는 것은 아무런 효과가 없으므로 여러분의 코드에서 객체와 문자열 모두 const 파라미터로 넘기도록 바꾸는 것도 좋은 방법입니다. 다만, 참조 카운트의 오버헤드는 매우 적기 때문에 의미 있는 정도의 속도 향상을 기대할 수는 없습니다.

참조 카운트가 지원되는 플랫폼에서는 아래의 새 속성을 이용하여 객체의 참조 카운트를 알아낼 수 있습니다. (**임프 역주**: TObject 클래스에서 구현되어 있음)

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

노트: 인터락 연산의 속도
쓰레드에 안전(thread safe)하기 위해서는, 객체 참조 카운트의 증가와 감소는 인터락(interlock) 혹은 쓰레드 세이프 연산을 이용해서만 이루어져야 합니다. 인텔 CPU가 LOCK 명령의 실행에서 모든 파이프라인/CPU가 지연되어 느려지는 문제가 있었던 적이 있었습니다. 최근의 인텔 CPU들에서는 적절한 캐시 라인만 락이 걸립니다. 이런 상황은 모바일 플랫폼들에서 사용되는 ARM CPU에서도 비슷합니다. 증가 및 감소 연산이 쓰레드에 안전하다는 사실이 이제 객체 인스턴스가 쓰레드에 안전하다는 것을 의미하지는 않습니다. 이것은 단지 모든 쓰레드들이 변경을 즉시 알게 되며 이전 값을 변경하는 일이 없도록 참조 카운트 인스턴스 변수가 적절히 보호된다는 의미일 뿐입니다.

노트: ARC와 컴파일러의 호환성
라이브러리 개발 등의 경우를 위해 ARC가 지원되는 환경과 지원되지 않는 환경 각각에 대해 최선의 코드를 작성하고 싶다면, 두 경우를 식별하기 위해 {$IFDEF AUTOREFCOUNT} 지시어를 사용할 수 있습니다. 이것은 중요한 지시어로서, 새로운 컴파일러를 식별하는 NEXTGEN과는 달리 클래식 델파이 컴파일러에도 ARC가 구현될 미래에 대해서도 대비할 수 있습니다(iOS 시뮬레이터에서도 이미 사용중입니다).

3.2: ARC에서의 FREE 및 DISPOSEOF 메소드

델파이 개발자들은 Free 메소드와 try-finally 블럭으로 감싸는 코딩 패턴을 사용해왔습니다. 대부분의 개발자들이 이와 같은 패턴의 코드를 대단히 많이 갖고 있고, 또한 여전히 윈도우 델파이와 호환되는 코드가 필요할 수 있기 때문에, ARC 환경에서의 Free의 사용에 대해 짚고 넘어갈 필요가 있습니다. 결론부터 말하자면 여러분의 기존 코드는 여전히 동작하지만, 어떻게 동작하는지 이해하기 위해 읽어둘 필요가 있습니다.

예를 들어, 여러분은 지금까지 위에서 봤던 코드를 보통 다음과 같이 작성해왔을 것입니다.

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

```
      MyObj.Free;
    end;
  end;
```

클래식 델파이 컴파일러에서는 *Free*는 *TObject*의 메소드로서 현재의 참조가 nil이 아닌지 확인하고 그런 경우 *Destroy* 파괴자를 호출하여 적절한 파괴자 코드를 실행한 후 메모리에서 객체를 제거하는 동작을 합니다.

차세대 컴파일러에서는 그와 달리 *Free* 호출은 변수에 *nil*을 대입하는 동작으로 바뀌었습니다. 해당 객체에 대한 마지막 참조일 경우, 이전과 마찬가지로 파괴자를 호출한 후 메모리에서 제거됩니다. 다른 참조가 남아있을 경우 아무 일도 발생하지 않습니다(물론 참조 카운트는 감소됩니다).

비슷하게, 다음과 같은 호출은,

```
FreeAndNil(MyObj);
```

객체에 *nil*을 대입하고, 그 객체를 참조하는 다른 변수가 없을 경우만 객체를 파괴합니다. 여러분이 프로그램의 다른 곳에서 사용되고 있는 객체를 파괴하려 하지는 않을 것이므로 대부분의 경우 이런 동작은 적절합니다. 하지만 다른 참조가 있을 수 있다는 점을 무시하고 즉시 코드를 즉시 실행하려 하는 경우도 있을 수 있습니다(파일이나 데이터베이스 연결을 닫으려 하는 경우에 그럴 수 있겠지요).

다르게 말하자면, 모바일에서 별 의미가 없기는 하지만, *Free*나 *FreeAndNil* 호출은 완벽하게 무해하고 기존의 델파이 프로그램의 이들 호출을 그대로 유지하며 모바일 플랫폼으로 옮길 수 있습니다. 하지만 다른 접근 방법이 필요한 경우가 드물게 있을 수 있지요.

강제로 파괴자를 실행할 수 있게 하기 위해(실제 객체를 메모리에서 소멸시키지 않고) 새 컴파일러는 *dispose* 패턴을 도입했습니다. 다음과 같이 호출하면,

```
MyObject.DisposeOf;
```

다른 참조가 남아있는 경우에도 강제로 파괴자 코드가 호출되게 됩니다. 이 시점에서 객체는 특수한 상태가 되어, 추가로 정리(dispose) 작업의 경우나 참조 카운트가 0이 되어 메모리가 실제로 해제되는 때에도 파괴자가 다시 호출되지 않습니다. 이렇게 "정리된(disposed)" 상태(혹은 "좀비" 상태)는 상당히 중요하므로, 여러분은 *Disposed* 속성을 이용하여 객체의 상태를 알아낼 수 있습니다.

노트: Win32에서의 DisposeOf
이 새 메소드는 윈도우와 맥 컴파일러에도 존재하지만, 이들 컴파일러에서 *DisposeOf* 메소드는 단지 *Free* 메소드를 호출하기만 합니다. 다르게 말하자면, 이 새 메소드는 서로 다른 플랫폼들 사이의 소스코드 호환성을 높이기 위해서 도입되었습니다.

왜 dispose 패턴이 필요할까요? 요소들의 집합의 경우나 다른 컴포넌트에 소유된(owned) 컴포넌트를 가정해봅시다. 일반적으로 사용되는 패턴은 아이템 자체의 해제와 집합으로부터의 제거 양쪽 모두를 위해 집합에서 특정 아이템을 "파괴"하는 것입니다. 다른 흔한 상황은 집합이나 컴포넌트 오너를 파괴하고 모든 소유된 요소나 컴포넌트들을 정리(dispose)하는 것입니다. 이런 경우에는, 소유된 객체에 대한 다른 참조가 존재해도 파괴를 시도하고 최소한 파괴자 코드는 실행하게 됩니다.

정리된(disposed) 후에 파괴된 인스턴스를 사용하면 에러가 발생할 수 있지만, 클래식 델파이 컴파일러에서의 동작과 크게 다르지는 않지요. 클래식 컴파일러에서 인스턴스를 해제한 후에 다른 참조를 사용해도 역시 에러가 발생하니까요. 클래식 델파이 컴파일러들에서는 객체에 대한 참조가 2개 있는 상황에서 그중 하나로 해제(free)를 했을 때 다른 참조가 여전히 유효한지 확인할 방법이 없었습니다. 새 컴파일러에서는 *Disposed*를 사용하면 객체의 상태를 알아낼 수 있습니다.

```
myObj := TMyClass.Create; // 첫번째 객체 참조
myObj.SomeData := 10;

myObj2 := myObj; // 동일 객체에 대한 다른 참조

myobj.DisposeOf; // 클린업을 강제함

if myobj2.Disposed then // 다른 참조의 상태를 체크
  Button1.Text := 'disposed';
```

앞에서, 클래식 델파이 컴파일러들에서 사용되는 클래식 try-finally 블럭은 새 컴파일러에서도 역시 잘 동작한다고 언급했습니다. 특정한 경우에 다른 참조의 존재와 무관하게 최대한 빨리 파괴자 코드의 실행을 강제하려 할 때는 dispose 패턴을 사용할 수 있습니다(물론 이전 버전의 델파이에서 코드를 재컴파일하지 않을 경우).

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

```
      MyObj.DisposeOf;
    end;
  end;
```

클래식 컴파일러들에서는 *DisposeOf*가 *Free*를 호출하여 기존과 동일하게 동작합니다. ARC 환경에서는 기대한 시점에서(즉 클래식 컴파일러와 동일한 시점에서) 파괴자를 호출하며, 하지만 메모리는 ARC 메커니즘에 의해 관리됩니다. 이런 방식은 멋지지만 동일한 코드를 델파이의 이전 버전들에서 사용할 수는 없게 됩니다. *FreeAndNil* 프로시저를 재정의하여 버전에 따라 *Free* 혹은 *DisposeOf*를 호출하도록 할 수 있습니다. 아니면 표준 *Free* 호출을 그대로 둬도 대부분의 경우에는 문제가 없습니다.

노트: Disposed 플래그의 저장소
Disposed 플래그의 실제 저장소를 위해서는 추가로 필드를 사용하지 않고 *FRefCount* 필드를 이용합니다. *FRefCount* 필드의 두 번째 비트가 파괴 추적 목적과 연관되어 사용되며, 그런 이유로 참조 카운트의 최대값은 이론적으로 2^30으로 제한되지만, 실제로 그 한계에 도달할 일은 현실적으로는 없겠지요.

*Free*와 *DisposeOf*의 차이점을 구별하는 한 가지 방법은, ARC 환경에서 두 동작의 목적을 살펴보는 것입니다(클래식 델파이 컴파일러들에서 일어나는 것과 비교하여). *Free*를 사용하는 목적은 단순히 특정 참조를 인스턴스로부터 "떼어내는" 것입니다. *Free*는 어떤 정리나 메모리 해제도 내포하지 않습니다. 이것은 단지 코드 블럭에서 그 참조가 더 이상 필요하지 않다는 것을 의미합니다. 이 동작은 보통 스코프를 벗어나면서 일어나지만 Free를 명시적으로 호출하여 호출할 수도 있습니다.

반대로, *DisposeOf*는 인스턴스에게 "*스스로 정리하라*"고 명시적으로 지시하는 프로그래머의 수단입니다. *DisposeOf*는 반드시 메모리 해제를 수반하지 않으며, 단지 명시적으로 인스턴스의 "*정리*"를 할 뿐입니다(특정 파괴자 코드를 실행). 해당 인스턴스가 최종적으로 사용중인 메모리를 해제하는 것은 일반적인 참조 카운트 방식에 의존합니다.

다른 말로 하자면, ARC 환경에서는 Free는 "인스턴스 참조에 중점을 두는" 동작이며, 반면 *DisposeOf*는 "인스턴스에 중점을 두는" 동작입니다. *Free*는 "인스턴스가 어떻게 되든 난 모르겠고, 어쨌든 난 더 이상 그게 필요하지 않아" 라는 의미입니다. *DisposeOf*는 "이 인스턴스가 메모리 외의 리소스를 잡고 있다면 해제해야 하니 내부적으로 스스로 정리하라"라는 의미입니다. (여기서 말하는 메모리 외의 리소스에는 파일 핸들, 데이터베이스 핸들, 소켓 등이 있겠지요)

*DisposeOf*가 필요한 다른 경우는, 복잡한 참조 사이클을 위한 적절한 정리 및 해제를 명시적으로 일으키는 것입니다. 다음 섹션에서 설명하는 약한 참조(weak reference)를 사용하면 더 깔끔하고 명시적이 되기는 하지만, 다른 인스턴스들에게 자신들의 참조를

놓도록 알려주기 위한 명시적인 유발 혹은 알림이 필요한 상황이 있을 수 있습니다.

노트: 포인터와 객체를 섞어쓸 때 주의
예를 들어 객체를 포인터에 대입하고, 객체 변수를 다른 다른 객체에 재활용하고, 다음에 포인터를 다시 그 객체 변수에 대입하면, 그 객체는 더 이상 존재하지 않습니다. 참조 카운트가 0이 되어 파괴되었기 때문입니다(포인터는 참조를 카운트를 하지 않으며 참조 카운트를 증가시키지 않기 때문). RTL의 *TStringList.Exchangeltems* 메소드가 바로 이런 이유로 변경되었습니다. 이 메소드는 이전에는 새 위치로 이동되는 객체에 대한 임시 "참조"를 저장해두기 위해 포인터를 이용했었습니다.

3.3: 약한 참조(WEAK REFERENCE)

ARC에서 또 다른 중요한 개념은 약한 참조(weak reference)의 역할입니다. 두 객체가 각각의 속성으로 서로를 참조하고 있고, 외부 변수가 그 중 첫 번째를 참조하고 있는 경우를 가정해봅시다. 첫 번째 객체의 참조 카운트는 2일 것이며(외부 변수 및 두 번째 객체의 필드), 두 번째 객체의 참조 카운트는 1이겠지요(첫 번째 객체의 필드). 이제, 외부 변수가 스코프를 벗어나면, 두 객체의 참조 카운트는 모두 1이 되고 영원히 메모리에 남게 됩니다.

이런 상황을 해결하려면 순환 참조를 깨야 하는데, 이 작업을 언제 실행할지 모르기 때문에 결코 간단하지 않습니다(마지막 외부 참조가 스코프를 벗어날 때 실행해야 하는데, 객체는 이 시점을 알 수가 없습니다). 이런 상황, 그리고 여러 비슷한 경우들에 대한 해법이 약한 참조를 이용하는 것입니다.

약한 참조는 객체의 참조 카운트를 증가시키지 않는 참조입니다. 앞의 경우에서, 두 번째 객체로부터 첫 번째 객체로의 참조가 약한 참조라면, 외부 변수가 스코프를 벗어나면 두 객체 모두 파괴됩니다.

간단한 경우를 아래의 코드로 살펴봅시다.

```
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;
```

아래는 다른 클래스의 객체를 생성하는 TMyComplexClass 클래스의 생성자입니다.

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

FOwnedBy 필드가 약한 참조이기 때문에 자신이 가리키는 객체(이 경우에는 현재 객체, 즉 *self*)의 참조 카운트를 증가시키지 않습니다. 이 클래스 구조에 따라 다음과 같이 코드를 작성할 수 있게 되죠.

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

약한 참조가 적절히 사용되었으므로 이렇게 하면 메모리 누수(memory leak)가 발생하지 않습니다.

약한 참조를 더 살펴보기 위해 델파이 RTL의 *TComponent* 클래스의 선언을 봅시다.

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

클래식 델파이 컴파일러에서 weak 어트리뷰트는 무시된다는 사실도 알아둡시다. 하지만 소유자 객체의 파괴자에는 소유된 객체도 함께 해제되도록 하는 적절한 코드를 추가하는 것을 잊지 마십시오. 앞에서 봤듯이, Free 호출은 델파이의 클래식 컴파일러와 ARM 컴파일러 모두 허용됩니다(물론 그 효과는 다르긴 합니다). 또한 두 컴파일러들

모두에서 대부분의 상황에서 적절합니다.

약한 참조를 사용할 때는 위의 예제에서 봤듯이 약한 참조 자체가 *nil*인지 검사해서는 안됩니다. 그러고 싶다면, 먼저 약한 참조를 강한 참조(strong reference)에 대입하고(내부적으로 특정 확인 작업들을 합니다), 다음으로 strong 참조를 확인하는 것입니다. 예를 들어 위에서 약한 참조인 *FOwner*의 경우에는 다음과 같이 코드를 작성할 수 있습니다.

```
var
  TheOwner: TComponent;       // 강한 참조(strong reference) 변수
begin
  TheOwner := FOwner;
  if TheOwner <> nil then
    TheOwner.ClassName;       // 안전함
end;
```

3.4 메모리 문제를 검사하려면

iOS 델파이 ARM 컴파일러에서 메모리 관리가 동작하는 방식 때문에, 모든 것을 제대로 컨트롤하고 있다고 확신하기 위해서는 몇가지 옵션들을 검토해볼 가치가 있습니다. 더 나아가기 전에 알아둘 중요한 것 하나. 윈도우가 아닌 플랫폼에서 델파이는 FastMM 메모리 관리자를 사용하지 않으므로, 프로그램 종료시에 메모리 누수 체크를 위해 전역 플래그 *ReportMemoryLeaksOnShutdown*을 설정하는 것은 무의미합니다. OS X 및 iOS 플랫폼에서 델파이는 *libc* 네이티브 라이브러리의 *malloc* 및 *free* 함수를 직접 호출합니다.

iOS 플랫폼에서 아주 좋은 해결책은 Apple의 Instruments 툴을 이용하는 것인데, 이것은 물리적인 디바이스에서 동작중인 여러분의 애플리케이션의 모든 측면을 감시하는 완전한 추적 시스템입니다. 이 툴에 대해서는 Daniel Magin와 Daniel Wolf는 델파이의 관점에서 이 툴을 자세히 설명하는 동영상이 아래 링크에 있습니다.

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

메모리 누수를 일으키는 잠재적인 이슈들 중 하나는 객체들 사이의 순환 참조이며, 여러분의 애플리케이션이 어떻게 동작하는지 알아내도록 도와줄 수 있는 작은 함수가 있습니다. 이것은 Classes 유닛에 있는 *CheckForCycles* 입니다.

```
procedure CheckForCycles(const Obj: TObject; const
```

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

이것은 여러분이 종료 코드에서 보통 사용하던 함수는 아니고, 개발 및 디버깅 동안에 테스트의 목적입니다. 이 프로시저의 두 번째 파라미터는 익명 메소드(anonymous method)로서, 객체의 클래스, 메모리 주소, 사이클 내 객체의 스택을 파라미터로 받습니다. 아래는 그 사용법에 대한 기본적인 예제로, 이전에 설명했던 클래스에서 약한 참조를 없앤 것입니다(약한 참조에서는 사이클이 없습니다).

```
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 어트리뷰트

매우 특수한 상황이 있는데, 함수가 참조 카운트가 0으로 설정된 객체를 리턴할 수도 있는 상황입니다(예를 들면 객체의 생성 동안). 이런 경우에는, 컴파일러가 객체를 곧바로 삭제하지 않도록 하기 위해(변수에 대입되어 참조 카운트가 1이 되기 전에), 해당 객체를 "unsafe"로 표시해두어야 합니다. 이것은 코드를 안전하게 하기 위해 참조 카운트를 일시적으로 무시하기 위한 것입니다.

이런 동작은 새로운 특정한 어트리뷰트, *[Unsafe]*으로 가능하며, 아주 특수한 상황에서만 필요합니다.

```
var
  [Unsafe] Obj1:  TObject;

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

특이한 예로, System 유닛에서는 이에 해당하는 지시어 *unsafe*가 이 어트리뷰트를 대체하는데, 이것은 단지 동일 유닛에서 어트리뷰트가 정의되기 전에는 그 어트리뷰트를

사용할 수 없기 때문입니다. 예를 들면 *TObject* 클래스의 저수준 *InitInstance* 클래스 함수가 이런 경우입니다. 이 함수는 객체에 메모리를 할당하기 위해 사용되며 다음과 같이 선언되어 있습니다.

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

unsafe 지시어는 위의 예처럼 System 유닛에서만 사용할 수 있도록 제한되어 있습니다.

3.6 저수준 참조 카운트 조작

대부분의 경우 여러분은 여러분의 코드에 참조 카운트를 적용하고, 필요하다면 약한 참조와 unsafe 어트리뷰트를 추가할 수도 있겠지만, 특정한 상황에는 객체를 위한 메모리를 직접 할당하고 자신만의 방식으로 관리해야 할 수도 있습니다. 비슷한 경우로, 참조 카운트의 방법으로는 제대로 관리되지 않고 실질적인 참조가 없는 객체를 메모리에 유지해야 하는 경우가 있을 수도 있습니다. 그런 경우에는(아주 드물지만) *TObject* 클래스의 두 개의 퍼블릭 가상 메소드를 호출하여 참조 카운트를 강제로 변경할 수 있습니다.

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

두 메소드 모두 동작 완료 후의 참조 카운트를 리턴합니다.

노트: 참조 카운트의 속도
위의 두 함수는 델파이에서 참조 카운트 코드의 핵심을 이루며, 필요할 때마다 컴파일러에 의해 자동으로 호출됩니다. 한 객체를 새 변수에 대입하면, 그 오버헤드는 가상 메소드 테이블(virtual method table)의 함수에 대한 단일 호출이며, 그로 인해 그 객체 자체의 필드 값이 증가됩니다. 이것은 Apple의 ObjectiveC의 현재의 ARC 구현을 포함한 다른 구현 방식들에 비해 훨씬 빠릅니다.

이 메소드들을 사용하는 한 가능성은, 여러분이 객체 타입으로 취급 혹은 캐스트하려 하는 메모리 블럭(외부 API로 할당되었을 수도 있음)을 가지고 있을 경우입니다. RTL에서 볼 수 있는 다른 예는, 한 객체의 데이터를 포인터를 이용하여 복사할

때입니다.

```
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 ARC에서 TOBJECT의 생성 및 파괴 메소드들의 정리

아래는 *TObject* 클래스에서 생성 및 파괴와 관련된 메소드들의 정리이며, 새로운 메소드들과 클래식 메소드들 모두 나열했습니다(조건 컴파일 지시어들은 생략).

```
type
  TObject = class
  public
    constructor Create;
    procedure Free;
    procedure DisposeOf;
    destructor Destroy; virtual; // ARC에서는 protected 섹션에 있음
    property Disposed: Boolean read GetDisposed;
    property RefCount: Integer read FRefCount; // ARC인 경우만 존재
    // 저수준 조작용 메소드들
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    classfunction NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    function ObjAddRef: Integer; virtual;
    function_ObjRelease: Integer; virtual;
```

서로 다른 플랫폼들에서 참조 카운트의 경우와 아닌 경우 양쪽에 대해 각 메소드의 구현의 자세한 내용을 파고 들어갈 수도 있겠지만, 그것은 이 새 언어 기능 소개에서는 너무 깊이 들어가는 것이겠지요.

3.8: 보너스 기능: 클래스에 대한 연산자 오버로드

메모리 관리에서 ARC를 사용하면 아주 재미있는 부작용이 있는데, 컴파일러가 함수에서 리턴된 임시 객체들의 생존기간을 관리할 수 있다는 것입니다. 그런 한 예는 연산자에서 리턴된 임시 객체입니다. 사실, 새로운 델파이 컴파일러에서 완전히 새로운 기능 하나는 클래스를 위한 연산자를 정의할 수 있다는 것입니다. 이것은 델파이 2006 이후로 레코드에서 사용 가능해졌던 것과 동일한 문법과 모델로 가능합니다.

노트: 클래스에 대한 연산자 오버로드를 지원하는 컴파일러는?
이 언어 기능은 iOS ARM 컴파일러에서 동작하지만, Mac용 iOS 시뮬레이터에서도 동작합니다. 물론 이 코드를 클래식 컴파일러로 윈도우나 Mac으로 컴파일할 수는 없습니다. 클래식 컴파일러에 이 기능을 추가하는 것이 그렇게 끔찍하게 어려운 일은 아니지만, 연산자들은 많은 임시 변수들을 만들어내므로 자동화된 메모리 관리 메커니즘(ARC나 가비지 컬렉션) 없이 이 기능을 구현하는 것은 말이 안되는 것입니다.

예를 들어, 다음과 같은 간단한 클래스를 살펴봅시다.

```
type
  TNumber = class
  private
    FValue: Integer;
    procedure SetValue(const Value: Integer);
  public
    property Value: Integer read FValue write SetValue;
    class operator 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;
```

이 클래스를 다음과 같이 사용할 수 있습니다.

```
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: 인터페이스와 클래스를 섞어 쓰기

과거에는, 인터페이스 변수와 표준 객체 변수는 서로 다른 메모리 관리 모델을 이용했기 때문에 일반적으로 두 방식을 섞어 쓰지 않도록 권장했었습니다(인터페이스와 객체 변수 혹은 파라미터가 메모리의 동일 객체를 참조하는 등).

ARC를 갖춘 새로운 ARM 컴파일러에서는 객체와 인터페이스 변수의 참조 카운트가 통합되었기 때문에 두 가지를 간단히 섞어 쓸 수 있게 되었습니다. 이렇게 함으로써 델파이 ARC 플랫폼에서 인터페이스는 ARC가 없는 플랫폼에서보다 더욱 강력하고 유연해졌습니다.

4. 기타 언어 변경들

문자열 타입 변경과 객체 메모리 관리 외에도, 새로운 델파이 ARM 컴파일러에는 여러분이 간단히 채용할 수 있는, 이미 적용되었거나 곧 적용될 변경 사항들이 있습니다.

- 머지 않아 *with* 문은 델파이 언어에서 deprecated 되거나 제거될 것입니다. 여러분은 with를 지금부터 코드에서 제거해나갈 수 있겠고, 대부분의 델파이 개발자들은 이것이 좋은 생각이라고 동의할 것입니다. 이 키워드의 숨겨진 함정들 때문이죠.
- 포인터 사용을 줄이거나 없애십시오. 우리는 자동 메모리 관리로 가고 있기 때문에 직접적인 포인터 사용을 피할 것을 권합니다. *TList* (내부적으로 포인터에 기반) 대신 제네릭 컨테이너 클래스를 사용하는 것이 델파이 RTL 라이브러리가 진행중인 마이그레이션의 좋은 예이며, 델파이 개발자들도 자신의 코드에서 비슷한 변환 작업을 할 것을 추천합니다.

- 어셈블리 코드를 제거하십시오. 어셈블리 코드는 새로운 컴파일러로 이식이 불가능하며 ARM CPU에 적용되지도 않습니다.
- volatile 어트리뷰트는 다른 쓰레드에 의해 변경될 필드를 표시하기 위해 사용되므로, 코드 생성 과정이 별도의 쓰레드에서 공유되지 않는 레지스터나 다른 임시 메모리 주소의 값 복사를 최적화하지 않습니다.

5. RTL 관련 고려사항들

지금까지 LLVM에 대한 소개와 컴파일러의 변경 사항들에 대해 설명했고, 이 문서의 마지막 파트는 모바일 플랫폼을 지원하기 위한 런타임 라이브러리(RTL)와 그 변경 사항에 초점을 둡니다. 이 마지막 부분은 앞부분들에 비해서 광범위하거나 자세하지는 않을 것이며, 런타임 라이브러리의 측면에서 모바일 플랫폼을 위해 마이그레이션을 하거나 새 애플리케이션을 작성할 때 여러분이 마주칠 수 있는 이슈들을 간단히 훑어볼 것입니다(유저 인터페이스나 데이터베이스, 네이티브 디바이스 센서 등에 대해서 다루지는 않습니다).

이 짧은 섹션에서는, 다른 운영체제를 지원하기 위한 일반적인 제안들, 파일 액세스와 관련된 정보, 패키지와 라이브러리에 대한 지원 등을 거론합니다.

5.1: RTL과 크로스플랫폼

몇가지 권고안들을 설명해보겠습니다.

- 고수준 컴포넌트나 클래스가 있다면 최대한 직접 API 호출을 피하십시오. 이렇게 함으로써 코드를 다른 플랫폼으로 포팅하기 쉽게 됩니다(당장 혹은 언젠가는).
- 크로스플랫폼 유닛들을 우선적으로 사용하십시오. 예를 들어 파일 관리를 위해서는 옛 스타일의 파일 관리 파스칼 루틴보다는 IOUtils 유닛의 레코드들과 그 클래스 메소드들을 사용할 것을 추천합니다. 바로 아래에서 IOUtils 유닛 사용의 몇가지 예를 볼 수 있습니다.
- 여러분의 코드를 Mac이나 모바일로 포팅 가능하게 하려면 윈도우스럽고 윈도우에 한정된 API를 피하십시오. 좋은 예는 MSXML을 사용하지 않고 델파이 기반의 ADOM XML 처리 엔진을 사용하는 것입니다. 이 엔진은 모바일에서도 동작합니다. 다른 예는 몇몇 소켓 및 웹 라이브러리들 대신 Indy를 우선 사용하는 것입니다.

말할 필요도 없이, COM, ActiveX, ADO, BDE, 그리고 다른 윈도우에 한정된 델파이 기술들은 모바일에서 동작하지 않습니다(물론 Mac에서도 안됩니다).

- *Generics.Collections* 유닛에 선언된 제네릭 컨테이너 클래스들을 사용하십시오. 옛 스타일의 Contnrs 유닛은 모바일 플랫폼에는 존재하지 않습니다. 이것은 Contnrs 유닛이 포인터 리스트(제네릭이 아닌 *TList* 클래스)에 기반하여 ARC 메모리 관리 모델에서 제대로 동작하지 않기 때문입니다.
- TStringList의 각 요소에 객체를 연결하여 사용하는 것은 지원되긴 하지만, 그보다는 제네릭 딕셔너리를 사용할 것을 강력하게 권합니다. 문자열을 키로 하여 *TDictionary<string,TMyClass>*와 같이 사용할 수 있습니다. 딕셔너리를 사용하는 것이 더 깔끔할 뿐만 아니라, 딕셔너리는 해시 테이블을 사용하므로 대부분의 경우에 훨씬 빠르기도 합니다. 아래에서 더 자세히 설명할 것입니다.
- 포인터 기반 구조와 메소드의 사용을 피하십시오. 네이티브 API 호출에 필요한 경우가 아니라면 말입니다.

위의 목록은 분명히 훨씬 길어질 것입니다. 당장은 이 정도만 나열했으며, 다음으로 파일 액세스와 라이브러리에 대해 따로 자세히 살펴봅시다.

5.2: 파일 액세스

델파이의 몇 버전 전부터, 전통적인 파스칼의 파일 액세스 루틴들(한 폴더 내에서 검색하기 위한 FindFirst와 FindNext처럼)은 고차원 레코드들로 대체되었으며, *IOUtils* (Input/Output utilities) 유닛에 모여있습니다. IOUtils를 사용하는 것은 여러분의 코드를 크로스플랫폼으로 만들기 위해 추천되는 방식입니다.

이 유닛에는 대부분 클래스 메소드로 정의된 3개의 레코드가 있는데, 각각 .NET 클래스들과 대응됩니다.

- *TDirectory* => System.IO.Directory
- *TPath* => System.IO.Path
- *TFile* => System.IO.File

*TDirectory*는 폴더를 뒤져 파일과 서브폴더를 찾기 위한 것이라는 것을 분명하게 알 수 있지만, *TPath*와 *TFile*의 차이점은 감이 잘 오지 않을 수 있겠습니다. *TPath*는 파일

이름과 디렉토리 이름을 다루기 위해 사용되며, 드라이브, 패스나 확장자 등을 제외한 파일 이름을 뽑아내거나 UNC 경로를 다루는 메소드들을 가지고 있습니다. 반면 *TFile* 레코드는 파일의 시간과 속성, 파일 조작, 파일 쓰기나 복사 등을 할 수 있게 해줍니다.

예를 들어, 여러분이 모바일 플랫폼에서 여러분의 애플리케이션과 함께 특정 파일을 함께 배포한다면, 윈도우에서 사용자의 문서 폴더를 액세스하는 것과 같은 방법으로 애플리케이션의 "documents" 폴더를 액세스할 수 있습니다.

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

폴더와 파일을 찾는 기능도 지원됩니다. 다음의 코드는 지정된 초기 폴더(*BaseFolder*) 아래의 폴더들을 읽어들인 후 한 단계만 아래로 이동하여 서브폴더의 파일들을 읽습니다.

```
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 스트링리스트를 제네릭 딕셔너리로 바꾸자

오랫동안 저 자신을 포함한 많은 델파이 개발자들은 *TStringList* 클래스를 지나치게 많이 사용해왔습니다. 단순한 문자열의 리스트로 사용하는 목적 외에도 이름/값 쌍들의 리스트로도 사용하고, 문자열과 연관된 객체들을 리스트로 저장하고 그 객체들을 검색하기 위한 목적으로도 사용해왔습니다.

델파이는 제네릭을 완벽하게 지원하므로, 맥가이버 칼처럼 사용해온 *TStringList* 대신 전용의 맞춰진 컨테이너 클래스로 바꾸는 것이 낫습니다. 예를 들면, 문자열 키와 객체 값을 가진 제네릭 딕셔너리가 *TStringList* 보다 더 깔끔하고 안전하며(타입 캐스트가 덜 일어나기 때문), 또 더 빠르게 실행됩니다(해시 테이블을 이용).

이들 사이의 차이점을 알아보기 위해 다음의 예제를 살펴봅시다.

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

이 리스트들은 아래와 같은 코드로 임의의 같은 값이 채워집니다.

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

리스트의 각 요소를 가져와 각각에 대해 이름으로 검색을 하는 두 메소드를 이용하여 속도 테스트를 진행했습니다. 두 메소드 모두 문자열의 리스트를 검색하지만 첫 번째는 스트링리스트에서 객체를 찾고, 반면 두 번째는 딕셔너리를 사용합니다. 첫 번째 경우 해당 타입으로 되돌리기 위해 *as* 타입 캐스트가 필요한 반면 딕셔너리에서는 이미 해당 클래스 타입에 연결되어 있다는 점에 주목하십시오. 아래는 두 메소드의 메인 루프입니다.

```
theTotal := 0;
for I := 0 to sList.Count - 1 do
begin
  aName := sList[I];
  // 이름을 검색
  anIndex := sList.IndexOf(aName);
  // 객체를 얻어냄
  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];
  // 객체를 얻어냄
  anObject := sDict.Items[aName];
  Inc(theTotal, anObject.Value);
end;
```

딕셔너리의 해시된 키에 비해 정렬된 스트링리스트(바이너리 서치)는 얼마나 많은 시간이 걸릴까요? 놀랍지도 않지만 딕셔너리가 더 빠릅니다. 윈도우 플랫폼에서 테스트한 결과를 밀리초 단위로 보여드립니다(모든 플랫폼에서 비슷한 결과가 나옵니다).

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

입력된 값들이 동일하므로 결과도 동일하지만, 소요된 시간은 상당히 다르죠. 딕셔너리는 스트링리스트에 비해 약 ***4분의 1의 시간***이 걸렸습니다(요소 수 백만 개로 테스트한 결과입니다).

물론, 이 예제와 이 짧은 섹션은 제네릭 딕셔너리와 컴파일러 수준에서 제네릭이 도입된 후 델파이 RTL에 추가된 최신 데이터 구조들의 강력함을 보여주는 데 있어서 빙산의 일각에 불과합니다. 이런 새로운 구조들은 모든 플랫폼에서 좋은 선택이지만, 메모리 관리 모델의 변경 때문에 iOS에서는 더욱 더 적절합니다.

5.4: 라이브러리 및 패키지

델파이의 강력한 기능들 중 하나는 더 모듈화된 방식으로 애플리케이션을 배포하기 위해 런타임 패키지를 이용하는 것입니다. 패키지는 특수한 목적의 동적 링크 라이브러리로, 윈도우에서는 DLL이며 Apple 플랫폼들에서는 *dylib*입니다. 윈도우와 OS X에서 런타임 패키지를 common 폴더에 배포하여 여러 애플리케이션들에 공유하도록 하거나 혹은 서로 다른 프로그램들 사이의 잠재적인 충돌을 피하기 위해 특정 애플리케이션 폴더에 배포할 수 있습니다.

iOS 플랫폼에서는 두 가지 모두 허용되지 않습니다. 물리적인 iOS 디바이스에 공유 라이브러리를 배포할 수 없으며(iOS 시뮬레이터에서는 사용할 수 있습니다만), 이것은 오직 Apple만이 운영체제 레벨에서 할 수 있는 작업이기 때문입니다. 동적 라이브러리를 애플리케이션에 추가할 수도 없는데 이것은 실행 파일만이 메인 프로그램이 될 수 있기 때문입니다.

이 제한은 런타임 패키지에 해당하는 문제가 아니라 더 일반적인 문제입니다. 예를 들어

midas.dll이나 dbExpress 드라이버 같은 표준 델파이 라이브러리들은 애플리케이션 실행 파일에 정적으로 링크됩니다. InterBase 클라이언트 라이브러리도 마찬가지입니다. 사실, 컴파일러는 정적 라이브러리에 대한 참조를 인식하고 최종 실행 파일에 링크하는 방법을 가지고 있습니다(기술적인 이슈이므로 이 문서에서 꺼내고 싶지는 않군요).

6. 결론

여러분이 이 문서에서 살펴본 것처럼, LLVM 아키텍처에 기반한 ARM용 델파이 컴파일러가 발표됨에 따라, 델파이 언어는 중대한 이행을 진행하고 있습니다. 하위 호환성을 유지하려는 노력을 계속하고 있지만, 우리는 개발자들이 전진하고 있는 새로운 기능들을 완전히 받아들이기를 기대하고 있습니다.

이 문서에서 설명한 언어 변경 사항들과 특히 ARC 지원은 델파이가 앞으로 더 나아갈 수 있게 해줄 것입니다. 이들 변경들의 일부는 새 플랫폼에 의한 것이며, 다른 일부는 델파이 언어의 거친 부분을 다듬고 델파이에 새롭고 사용자들에게 편리한 기능들을 추가하기 위한 것입니다.

필자에 대하여

마르코 칸투(Marco Cantù)는 최근 엠바카데로 테크놀로지에 델파이 프로덕트 매니저로 합류하였습니다. 그는 베스트셀러인 마스터링 델파이(Mastering Delphi) 시리즈의 저자이며 최근 몇 년간은 델파이의 몇 버전에 대한 *Handbook* 시리즈를 직접 출판하기도 했습니다(2007 버전부터 XE 버전).

마르코는 컨퍼런스 강연자로 자주 등장해왔으며, 델파이에 대한 수많은 아티클들의 저자이며, 전세계의 기업들에서 고급 델파이 강좌들(델파이 웹 개발 포함)을 진행해왔습니다. 마르코의 블로그는 *http://blog.marcocantu.com* 이며, 트위터는 *@marcocantu*, 메일 주소는 *marco.cantu@embarcadero.com* 입니다.

역자에 대하여

박지훈.임프(Jeehoon Imp Park)는 한국의 메이저 델파이/C++빌더 커뮤니티인 볼랜드포럼의 설립자이자 대표시삽입니다. 델파이 및 C++빌더 개발 컨설턴트로

활동하고 있는 그는 기업 업무시스템 개발 프레임워크 개발을 주력으로 하고 있으며, 패키지 소프트웨어 개발과 기타 분야에도 많은 경험을 가지고 있습니다. 임프는 엠바카데로사의 한국 에이전트인 데브기어의 공동 창업자이며, 델파이 언어 문법서인 "델파이 프로그래밍 언어"를 번역 출간하기도 했습니다. 임프의 블로그는 *http://blog.devquest.co.kr/imp* 이며, 메일 주소는 *Jeehoon.Imp.Park@gmail.com*, 페이스북은 *www.facebook.com/Jeehoon.Imp.Park* 입니다.

엠바카데로 테크놀로지에 대하여

엠바카데로 테크놀로지사는 애플리케이션 개발자들과 데이터베이스 전문가들을 위한 수상 경력의 툴들을 제공하는 선도적인 공급사로서, 플랫폼과 프로그래밍 언어에 관계 없이 시스템들을 제대로 설계하고 더 빠르게 구축하며 더 잘 운영할 수 있도록 해줍니다. 포춘 100대 기업들 중 90개, 전세계 3백만 이상의 커뮤니티가 생산성을 높이고 비용을 줄이며 변경 관리 및 준수, 혁신을 가속화시키기 위해 엠바카데로 제품을 사용하고 있습니다. 1993년에 설립된 엠바카데로는 샌프란시스코에 본사를 두고 있으며 전세계에 지사를 운영하고 있습니다. 엠바카데로 웹 사이트는 www.embarcadero.com 입니다.