embarcadero
Developer Camp

【A1】Delphiテクニカルセッション

Delphi 2009 Handbook以降のDelphi言語新機能

東洋テクニカルシステム株式会社
システム開発部
福士　光

embarcadero
Developer Camp

アジェンダ

アジェンダ

- Delphi 2009/2010/XE/XE2で追加された言語およびRTLの新機能の中から注意すべきもの、興味深いものを取り上げて、その機能や使用方法などを見ていきます。
  - VCL/FMX/DataSnapなどのライブラリやIDE、デバッガなどについては基本的に扱いません。
  - C++Builder/RadPHP/Prismについても扱いません。

embarcadero
Developer Camp

Delphi 2009の新機能

Delphi 2009の新機能

- Delphi 2009の新機能のうち、ここでは以下のものを取り上げます。
  - Unicodeの全面的な採用
  - ジェネリックス
  - 無名メソッド　デモ
  - コンパイラの変更
  - RTLの強化

  - 詳しくは「Delphi 2009 Handbook」をお読みください。

Unicodeの全面的な採用 (1)

- 文字列型および文字型の変更
  - 標準のString(=UnicodeString)はUTF-16になりました。
  - 従来の文字列はAnsiStringとして使用可能です。
  - WideString(WindowsのCOM BSTRと同等)は従来のままです(WideString≠UnicodeString)。
  - 文字はChar(=WideChar)とAnsiChar、ポインタはPChar(=PWideChar)とPAnsiCharになります。
  - 文字列型(AnsiStringおよびUnicodeString)のメモリレイアウトが変更されています。
    - 要素サイズ (オフセット = -10)
    - コードページ (オフセット = -12)

Unicodeの全面的な採用 (2)

- コードページ (1)
  - 文字列にはコードページが付くようになりました。
  - AnsiStringにはデフォルトで実行環境のコードページが設定されます。
    - グローバル変数DefaultSystemCodePage(Systemユニット)に格納されています。
    - 英語版Windowsでは1252、日本語版Windowsでは932など。
      - http://msdn.microsoft.com/en-us/library/windows/desktop/dd317756.aspx
  - コードページはStringCodePage/SetCodePage関数(Systemユニット)で取得、設定できます。

Unicodeの全面的な採用 (3)

- コードページ (2)
  - コードページの異なる文字列間の代入では自動的に変換が行われます。
  - デフォルト以外のコードページを持つAnsiStringは以下のように定義します。

```
type
  ShiftJISString = type AnsiString(932);   // CP932:  shift_jis
  Latin1String   = type AnsiString(1252);  // CP1252: windows-1252

var
  S1: ShiftJISString;
```

  - コードページを指定しないRawByteStringも定義されています (CodePage = $FFFF)。

Unicodeの全面的な採用 (4)

- 文字列あるいはそれに準ずるものとその使い分け

String (UnicodeString)	普通はこれを使います
AnsiString	ANSI(Shift_JIS)文字列との互換性を特に要求されるとき
UCS4String (array of UCS4Char)	UTF-32(UCS-4)で扱いたいとき 文字列型ではなく動的配列なので0オリジンとなっており、注意が必要です
UTF8String (AnsiString(65001))	UTF-8で扱いたいとき
RawByteString (AnsiString($ffff))	メソッドのパラメータとして特定のコードページを前提とせず無変換のまま受け取りたいようなとき
TBytes (array of Byte)	通信電文のような、本来は文字列ではないものを扱うとき

Unicodeの全面的な採用 (5)

- 移行の際はコンパイラの出力する警告を無視せずに解決しておきましょう
  - W1057: 文字列の暗黙的なキャスト
  - W1058: データ損失の可能性がある文字列の暗黙的なキャスト
  - W1059: 文字列の明示的なキャスト
  - W1060: データ損失の可能性がある文字列の明示的なキャスト
  - W1062: 指定されたワイド文字列定数を縮小変換した結果、情報が失われました
  - W1063: 指定された AnsiChar 定数を WideChar に拡大変換した結果、情報が失われました
  - W1064: 指定された AnsiString 定数を拡大変換した結果、情報が失われました

Unicodeの全面的な採用 (6)

- これよりも詳しいことはRAD Studioマイグレーションセンターの Unicode関連記事や過去のデベロッパーキャンプのセッション資料を参照してください。
  - URLはこの資料の末尾の参考文献を参照。

ジェネリックス (1)

- ジェネリックスとは
  - C++のテンプレート、.NET FrameworkやJava(J2SE 5.0以降)のジェネリックスに相当します。
  - 型パラメータによるプログラミングが可能になります。
    - 型毎にサポートクラスを作っていたような場合に有効です。
  - 文法、ライブラリは.NET Frameworkに類似しています。
  - 実装はC++のテンプレートと同様です。
    - 実際に使用している型パラメータ毎にコードが展開されます。
    - C++や.NET Frameworkとの違いについてはCraig Stuntzさんによる以下の記事を参照してください。
      - http://blogs.teamb.com/craigstuntz/2009/10/01/38465/

ジェネリックス (2)

- ジェネリックスをサポートするコレクションクラス (1)
  - Systemユニット
    - `TArray<T>`
  - Generics.Collectionsユニット
    - `TList<T>`
    - `TQueue<T>`
    - `TStack<T>`
    - `TPair<TKey,TValue>`
    - `TDictionary<TKey,TValue>`
    - `TObjectList<T: class>`
    - `TObjectQueue<T: class>`
    - `TObjectStack<T: class>`
    - `TObjectDictionary<TKey,TValue>`
    - `TThreadedQueue<T>`

ジェネリックス (3)

- ジェネリックスをサポートするコレクションクラス (2)
  - 標準のコレクションクラスで不足の場合はAlexandru Ciobanu さんの“Generic collections library” (delphi-coll)がお勧めです。
    - delphi-coll - Generic collections library for Delphi 2010 and XE - Google Project Hosting
      - http://code.google.com/p/delphi-coll/
    - Delphi XE2では最新版(1.2)をSVNでチェックアウトする必要があります。

無名メソッド (1)

- 無名メソッド(anonymous method)とは (1)
  - C++11のラムダ、.NET Framework 2.0の匿名メソッドあるいは3.0のラムダに相当します。
  - パラメータとしてメソッド(のエントリアドレス)を渡すような状況で、無名の(名前があってもよい)コードブロックを渡すことができます。
    - コードブロックの型は“reference to ...”になります
    - ジェネリックスを使用したTProcあるいはTFunc(SysUtilsユニット)を利用することもできます。

無名メソッド (2)

- 無名メソッド(anonymous method)とは (2)
  - コードブロックで呼び出し元の変数を参照できます

    →キャプチャ
  - キャプチャによりその変数の生存期間が延長されます

    →クロージャ
    - TInterfacedObjectから派生し、メソッドInvokeから呼び出されるコードブロックとキャプチャした変数を含むクラスをコンパイラが自動的に生成します。
    - 生成されたインスタンスは参照カウントが0になるまで存続します。

無名メソッド (3)

- 関数ポインタを使う今までのやりかただと...

```
type
  TPerson = class(TObject)
  private
    FCarNo: Integer;
    FName: String;
    FBirthday: TDateTime;
  public
    property CarNo: Integer read FCarNo write FCarNo;
    property Name: String read FName write FName;
    property Birthday: TDateTime read FBirthday write FBirthday;
  end;

  TForm1 = class(TForm)
    ...
  private
    FList: TObjectList;
    procedure ShowList;  // 表示用
    ...
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  FList := TObjectList.Create(True);
  ...
```

無名メソッド (4)

比較用の関数を
必要なだけ用意

```
function SortByCarNoAsc(Item1, Item2: Pointer): Integer;
begin
  Result := TPerson(Item1).CarNo - TPerson(Item2).CarNo;
end;

function SortByCarNoDesc(Item1, Item2: Pointer): Integer;
begin
  Result := TPerson(Item2).CarNo - TPerson(Item1).CarNo;
end;

function SortByNameAsc(Item1, Item2: Pointer): Integer;
begin
  Result := CompareStr(TPerson(Item1).Name,TPerson(Item2).Name);
end;

function SortByNameDesc(Item1, Item2: Pointer): Integer;
begin
  Result := CompareStr(TPerson(Item2).Name,TPerson(Item1).Name);
end;

function SortByBirthdayAsc(Item1, Item2: Pointer): Integer;
begin
  Result := Trunc(TPerson(Item1).Birthday - TPerson(Item2).Birthday);
end;
// (以下略)
```

無名メソッド (5)

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0:
    begin
      if CheckBox1.Checked = False then
      begin
        FList.Sort(SortByCarNoAsc);
      end
      else
      begin
        FList.Sort(SortByCarNoDesc);
      end;
    end;

    1:
    begin
      if CheckBox1.Checked = False then
      begin
        FList.Sort(SortByNameAsc);
// (略)
  end;
  ShowList;
end;
```

比較用の関数の
どれかを渡す

無名メソッド (6)

- 無名メソッドを使うと...

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  FList.SortList(
    function (Item1, Item2: Pointer): Integer
    begin
      Result := 0;
      case RadioGroup1.ItemIndex of
        0:
        begin
          Result := TPerson(Item1).CarNo - TPerson(Item2).CarNo;
        end;
        1:
        begin
          Result := CompareStr(TPerson(Item1).Name,TPerson(Item2).Name);
        end;
// (略)
      if CheckBox1.Checked = True then
      begin
        Result := -Result;
      end;
    end
  );
  ShowList;
end;
```

比較用の関数を直接記述

フォーム上のコントロールの
プロパティを参照することもできる

無名メソッド (7)

- さらにジェネリックス版なら...

```
uses
  ..., System.Generics.Defaults, System.Generics.Collections, ...

type
  TForm3 = class(TForm)
    ...
  private
    FList: TObjectList<TPerson>;
    procedure ShowList;
    ...
  end;

procedure TForm3.FormCreate(Sender: TObject);
begin

  FList := TObjectList<TPerson>.Create(True);
  ...
```

ジェネリックス版TObjectList
(要素はTPerson)

無名メソッド (8)

```
procedure TForm3.Button1Click(Sender: TObject);
begin
  FList.Sort(TComparer<TPerson>.Construct(
    function (const Item1, Item2: TPerson): Integer
    begin
      Result := 0;
      case RadioGroup1.ItemIndex of
        0:
        begin
          Result := Item1.CarNo - Item2.CarNo;
        end;
        1:
        begin
          Result := CompareStr(Item1.Name,Item2.Name);
        end;
// (略)
      if CheckBox1.Checked = True then
      begin
        Result := -Result;
      end;
    end
  ));
  ShowList;
end;
```

TComparer<T>.Constructを
呼び出して、IComparer<T>を持つ
TDelegatedComparerを生成

無名メソッド (9)

- 無名メソッドのいいところ
  - 直感的なコード記述が可能になります
    - パラメータとして(別の場所に置いた関数の名前ではなく)コードブロックそのものを記述できます。
  - 呼び出し元のコンテキストの参照が容易にできます
    - 通常はDataなどのパラメータを経由して渡しますが、静的キャストを使うため型安全ではありません。
    - そもそも追加的な情報を渡す方法がない場合もあります(サンプルのケース)。

無名メソッド (10)

- 無名メソッドのいまひとつなところ
  - × 無名メソッドの定義が煩雑になる場合があります
    - パラメータが多い場合などは結構面倒に…。
  - × 無名メソッド版が用意されていなければ使えません。
    - class helperなどを利用して自分で無名メソッドの呼び出しを用意すればいいのですが。
  - × オーバヘッドもないわけではありません。
    - 実行速度については気にしなくてもよい(8%程度)。
    - 無名メソッドごとに約500バイトのメモリが必要。

      (DELPHI 2009 HANDBOOKより)
  - × セミコロンの付け方が微妙です。

無名メソッド (11)

- どのような状況で使えばいいのでしょうか？
  - 無理に使う必要はありませんが…
  - コールバックをより手軽にコーディング
  - ワーキングスレッドを簡単に生成
    - `TThread.CreateAnonymousThread`
  - メインスレッドとの同期
    - `TThread.Synchronize` (無名メソッド版)
  - マルチスレッドでキャプチャを有効利用

コンパイラの変更

- “Pointer Math”
  - Delphi 2007までではCのように配列とポインタを同列に扱うことができるのはPChar型だけでした。
  - {$POINTERMATH ON}でPChar以外の型付きポインタに対する演算を許可します(デフォルトの状態は{$POINTERMATH OFF})。
  - PCharがPWideCharになったため、既存コードで文字列以外に使用しているPChar型のポインタはPByte型に置き換えるか、{$POINTERMATH ON}として本来のデータ型に対するポインタに置き換える必要があります。

RTLの強化 (1)

- Cライクな整数型の別名 (Systemユニット)

新しい型名	元になる型	
Int8	ShortInt	8bit 符号あり
Int16	SmallInt	16bit 符号あり
Int32	LongInt	32bit 符号あり
Int64		64bit 符号あり
UInt8	Byte	8bit 符号なし
UInt16	Word	16bit 符号なし
UInt32	LongWord	32bit 符号なし
UInt64		64bit 符号なし
NativeInt		32bit/64bit 符号あり
NativeUInt		32bit/64bit 符号なし

RTLの強化 (2)

- プラットフォーム依存の整数型 (Systemユニット)
  - `NativeInt` (符号あり)
  - `NativeUInt` (符号なし)
    - 32bit環境では32bitの整数、Delphi XE2以降の64bit環境では64bitの整数になります(ポインタと同じサイズの整数)。
    - Delphi 2009以降でのみ正しく実装されています。
    - Delphi 7～2007の実装は正しくないので、互換性を維持する必要があるコードでは`{$IF RTLVersion >= 20.0}...{$ELSE}...{$IFEND}`などのガードが必要です。

RTLの強化 (3)

- `TStreamReader/TStreamWriter`クラス (Classesユニット)
  - 指定された文字エンコーディングに従って値を読み書きするためのクラスです。
  - .NET FrameworkのStreamReader/StreamWriterに相当します。
- AnsiStringsユニット
  - 互換性維持のためのAnsiString型に対する処理がまとめられているユニットです。
- Exit手続き (Systemユニット)
  - “`Exit(`<*戻値*>`);`”という記述が可能になりました。
    - Delphi 2007以前との互換性の観点からはあまりお勧めしません。

RTLの強化 (4)

- 例外の強化 (SysUtilsユニット)
  - 例外チェーン(Exception.RaiseOuterException)
    - 捕捉した例外のコピーをInnnerExceptionプロパティに格納した、新しい例外を生成します。
    - 例外の持つ情報を失うことなく別の例外を生成できます。
    - InnerExceptionはネストできます。
    - BaseExceptionプロパティが最初に生成された例外です。
  - 例外の前処理(Exception.RaisingException)
    - protected virtualと定義されています。
    - 派生した例外クラスでoverrideすることで、例外クラスのインスタンスが実際に予約語raiseで送出される直前に処理を行うことができます。

embarcadero
Developer Camp

Delphi 2010/XE/XE2の新機能

Delphi 2010/XE/XE2の新機能

- Delphi 2010/XE/XE2の新機能のうち、ここでは以下のものを取り上げます。
  - 拡張RTTIと属性 デモ
  - コンパイラの変更
  - ユニットスコープ名の導入
  - 基本型の変更
  - RTLの強化
  - クロスプラットフォーム

拡張RTTIと属性 (1)

- 拡張RTTI(“Extended” RTTI)と属性(attribute)とは
  - Delphi 2010で導入。
  - .NET Frameworkのメタデータとカスタム属性に相当します。
  - メタプログラミングでより多くのことが可能になります。
  - RTTIユニットに必要なクラス型、レコード型などが定義されています。
  - Delphi 2010以降で生成した実行ファイルが大きくなってしまう原因のひとつです。

拡張RTTIと属性 (2)

- クラス型(class)またはレコード型(record)が対象です
  - 型情報そのものはIntegerやBooleanなどの単純型を含め、全ての型に存在しています。
- 実行時に型情報とインスタンスへのポインタを元に各種の操作を行います

拡張RTTIと属性 (3)

- {$RTTI ...}コンパイラ指令
  - プロパティ、フィールド、メソッドのそれぞれに対して、拡張RTTIをどの可視性(published/public/protected/ private)のものに付けるのかを制御します。
  - デフォルトでは以下の範囲に付けられています(Systemユニットで定義)。

可視性	private	protected	public	published
フィールド	○	○	○	○
プロパティ	×	×	○	○
メソッド	×	×	○	○

  - {$RTTI EXPLICIT ...}で(継承元クラスの指定とは独立して)拡張RTTIを付ける範囲を指定できます。

拡張RTTIと属性 (4)

- TRTTIContext
  - 全ての拡張RTTI操作はTRTTIContextから始まります。
  - 高度なレコード型として定義されています。
  - 内部リソースの管理、解放のため、クラスのコンストラクタ、デストラクタのようにclass function Createとprocedure Freeを呼び出すことが推奨されています。
  - GetTypeメソッド
    - 指定されたクラス型のRTTIオブジェクト(TRTTITypeから派生したクラスのインスタンス)を取得

```
function GetType(AClass: TClass): TRttiType;
```

拡張RTTIと属性 (5)

- TRTTIType
  - RTTIオブジェクトの基底クラス
  - GetPropertiesメソッド
    - 所属するクラスのプロパティのRTTI情報を全て取得

    ```
    function GetProperties: TArray<TRttiProperty>;
    ```
  - GetFieldsメソッド
    - 所属するクラスのフィールドのRTTI情報を全て取得

    ```
    function GetFields: TArray<TRttiField>;
    ```
  - GetMethodsメソッド
    - 所属するクラスのメソッドのRTTI情報を全て取得

    ```
    function GetMethods: TArray<TRttiMethod>;
    ```
  - 階層順に(継承先から継承元に向かって)RTTI情報がリストアップされます。

拡張RTTIと属性 (6)

- TRTTIType (続き)
  - GetDeclaredPropertiesメソッド

    ```
    function GetDeclaredProperties: TArray<TRttiProperty>;
    ```
  - GetDeclaredFieldsメソッド

    ```
    function GetDeclaredFields: TArray<TRttiField>;
    ```
  - GetDeclaredMethodsメソッド

    ```
    function GetDeclaredMethods: TArray<TRttiMethod>;
    ```
  - そのクラスで定義したRTTI情報だけがリストアップされます。

拡張RTTIと属性 (7)

- TRttiNamedObject
  - 名前付きRTTIオブジェクトの基底クラス
  - Nameプロパティ
    - 対象(メンバ)の名前

```
property Name: string;
```

- TRTTIMember
  - メンバ(プロパティ、フィールド、メソッド)のRTTI情報の基底クラス(TRttiNamedObjectから派生)
  - Visibilityプロパティ
    - メンバの可視性

```
property Visibility: TMemberVisibility;
```

拡張RTTIと属性 (8)

- TRTTIProperty
  - クラスのプロパティのRTTI情報(TRTTIMemberから派生)
  - IsReadable/IsWritableプロパティ
    - 読み込み/書き込み可能かどうか(プロパティのread/write指定子に対応)

```
property IsReadable: Boolean;
property IsWritable: Boolean;
```

  - GetValue/SetValueメソッド
    - ポインタで指定されたインスタンスのプロパティを読み込み/書き込み

```
function GetValue(Instance: Pointer): TValue;
procedure SetValue(Instance: Pointer; const AValue: TValue);
```

拡張RTTIと属性 (9)

- TRTTIField
  - クラスのフィールドのRTTI情報(TRTTIMemberから派生)
  - GetValue/SetValueメソッド
    - ポインタで指定されたインスタンスのフィールドを読み込み/書き込み

```
function GetValue(Instance: Pointer): TValue;
procedure SetValue(Instance: Pointer; const AValue: TValue);
```

拡張RTTIと属性 (10)

- TRTTIMethod
  - クラスのメソッドのRTTI情報(TRTTIMemberから派生)
  - MethodKindプロパティ
    - メソッドの種別(コンストラクタ、デストラクタ、procedure、function など)

```
property MethodKind: TMethodKind;
```

  - Invokeメソッド
    - メソッドの呼び出し

```
function Invoke(Instance: TObject;
                const Args: array of TValue): TValue;
function Invoke(Instance: TClass;
                const Args: array of TValue): TValue;
function Invoke(Instance: TValue;
                const Args: array of TValue): TValue;
```

拡張RTTIと属性 (11)

- TValue
  - 高度なレコード型
  - 拡張RTTIでデータを格納するのに使われます(値やパラメータ、戻値など)
  - バリアントもどき(“バリアント型の軽量版”)
  - 実際のデータはFDataフィールド(TValueDataレコード型、共用体)に格納しています
  - 格納するデータが配列の場合はそれぞれの要素もTValue型になります(TValueが入れ子になる)。

拡張RTTIと属性 (12)

- TValue
  - Kindプロパティ
    - 型の種類を取得

```
property Kind: TTypeKind;
```

  - TypeInfo/TypeDataプロパティ
    - 型の情報を取得

```
property TypeInfo: PTypeInfo
property TypeData: PTypeData;
```

拡張RTTIと属性 (13)

- TValue
  - IsEmptyプロパティ
    - データが格納されているかどうか。

    ```
    property IsEmpty: Boolean;
    ```
  - IsXXXXメソッド
    - 格納されているデータの状態を問い合わせる。

    ```
    IsObject/IsInstanceOf/IsClass/IsOrdinal/IsType/IsArray
    ```
  - AsXXXX/TryAsXXXXメソッド
    - 格納されているデータを特定の型で取得する。

    ```
    AsObject/AsClass/AsOrdinal/AsType/AsInteger/AsBoolean
    AsExtended/AsInt64/AsInterface/AsString/AsVariant/AsCurrency
    TryAsOrdinal/TryAsType
    ```

拡張RTTIと属性 (14)

- TValue
  - 暗黙の型変換 (implicit conversion)
    - データを格納する(直接代入で対応する型のclass operator Implicitが呼び出される)。

    string/Integer/Extended/Int64/TObject/TClass/Boolean
  - FromXXXXメソッド
    - データを格納する。

    FromVariant/From<T>/FromOrdinal/FromArray
  - ToStringメソッド
    - データをとりあえず文字列化。

    function ToString: string;

拡張RTTIと属性 (15)

- サンプル 1: クラスのメンバ(フィールド、プロパティ、メソッド)の列挙と型情報の取得

```
procedure TForm1.ShowRTTI(obj: TObject; Strings: TStrings);
var
  ctx: TRTTIContext;
  prp: TRttiProperty;
  fld: TRttiField;
  mtd: TRttiMethod;
  prms: TArray<TRttiParameter>;
  prm: TRttiParameter;
  Str: String;
begin
  with Strings do
  begin
    Clear;
    ctx := TRttiContext.Create;
    try
      Add(Format('Class: %s',[obj.ClassName]));
      { Enumerate properties }
      Add('Properties');
      for prp in ctx.GetType(obj.ClassType).GetProperties do
      begin
        Str := Format(' %s: %s [%s] %s ',
```

拡張RTTIと属性 (16)

```
                [prp.Name,
                 TEnumHelper.GetEnumName(prp.Visibility),
                 SRWString[prp.IsReadable,prp.IsWritable],
                 prp.PropertyType.ToString]);
    Add(Str);
  end;
  Add('---');
  { Enumerate fields }
  Add('Fields');
  for fld in ctx.GetType(obj.ClassType).GetFields do
  begin
    Str := Format(' %s: %s %s ',
                [fld.Name,
                 TEnumHelper.GetEnumName(fld.Visibility),
                 fld.FieldType.ToString]);
    Add(Str);
  end;
  Add('---');
  { Enumerate methods }
  Add('Methods');
  for mtd in ctx.GetType(obj.ClassType).GetMethods do
  begin
    Str := Format(' %s: %s %s ReturnType: ',
                [mtd.Name,
                 TEnumHelper.GetEnumName(mtd.Visibility),
                 TEnumHelper.GetEnumName(mtd.MethodKind)]);
```

拡張RTTIと属性 (17)

```
{ Return type }
if mtd.ReturnType <> nil then
begin
  Str := Str + mtd.ReturnType.ToString;
end
else
begin
  Str := Str + '(n/a)';
end;
{ Parameter types }
Str := Str + ' ParamTypes: (';
prms := mtd.GetParameters;
if Length(prms) > 0 then
begin
  for prm in prms do
  begin
    if pfOut in prm.Flags then
    begin
      Str := Str + 'out ';
    end
    else if pfConst in prm.Flags then
    begin
      Str := Str + 'const ';
    end
    else if pfVar in prm.Flags then
    begin
```

拡張RTTIと属性 (18)

```
            Str := Str + 'var ';
          end;
          if prm.ParamType <> nil then
          begin
            Str := Str + prm.ParamType.ToString;
          end;
          Str := Str + ', ';
        end;
        System.Delete(Str,Length(Str) - 1,2);
      end
      else
      begin
        Str := Str + 'n/a';
      end;
      Str := Str + ')';
      Add(Str);
    end;
    Add('---');
  finally
    ctx.Free;
  end;
 end;
end;
```

拡張RTTIと属性 (19)

- サンプル 2: フィールド、プロパティのデータの取得

```
if prp.PropertyType.TypeKind <> tkInterface then
begin
  v := prp.GetValue(obj);
  case v.Kind of
    tkEnumeration, tkSet:
    begin
      Str := Str + String(v.TypeInfo.Name) + '.' + v.ToString;
    end;
    tkChar, tkString, tkLString, tkWString, tkUString, tkWChar:
    begin
      Str := Str + '''' + v.ToString + '''';
    end;
    tkRecord:
    begin
      Str := Str + String(v.TypeInfo.Name) + ' ' + v.ToString;
    end
    else
    begin
      Str := Str + v.ToString;
    end;
  end;
end;
```

拡張RTTIと属性 (20)

- サンプル 3: フィールド、プロパティのデータの設定

```
procedure TForm3.SetPropertyValue(obj: TObject; const Name: String; const Value: String);
var
  ctx: TRTTIContext;
  prp: TRttiProperty;
  v: TValue;
  EnumValue: Integer;
begin
  v := TValue.Empty;
  ctx := TRttiContext.Create;
  try
    prp := ctx.GetType(obj.ClassType).GetProperty(Name);
    if prp <> nil then
    begin
      case prp.PropertyType.TypeKind of
        tkInteger: v := StrToInt(Value);
        tkChar:    v := TValue.From<AnsiChar>(AnsiChar(Value[1]));
        tkFloat:   v := StrToFloat(Value);
        tkString:  v := TValue.From<ShortString>(ShortString(Value));
        tkWChar:   v := Value[1];
        tkLString: v := TValue.From<AnsiString>(AnsiString(Value));
        tkWString: v := WideString(Value);
        tkInt64:   v := StrToInt64(Value);
        tkUString: v := Value;
```

拡張RTTIと属性 (21)

```
        tkEnumeration:
        begin
          EnumValue := GetEnumValue(prp.PropertyType.Handle,Value);
          if EnumValue >= 0 then
          begin
            v := TValue.FromOrdinal(prp.PropertyType.Handle,EnumValue);
          end;
        end;
      end;
      if v.IsEmpty = False then
      begin
        prp.SetValue(obj,v);
      end;
    end;
  finally
    ctx.Free;
  end;
end;
```

拡張RTTIと属性 (22)

- サンプル 4: メソッドの呼び出し(Invoke)

```
procedure TForm4.InvokeMethod(obj: TObject; const Name: String; const Param1: String;
                              const Param2: String; const Param3: String; const Param4:
String);
var
  ctx: TRTTIContext;
  mtd: TRttiMethod;
  prms: TArray<TRttiParameter>;
  v: array of TValue;
begin
  ctx := TRttiContext.Create;
  try
    mtd := ctx.GetType(obj.ClassType).GetMethod(Name);
    if mtd <> nil then
    begin
      prms := mtd.GetParameters;
      if Length(prms) <= 4 then
      begin
        SetLength(v,Length(prms));
        if Length(prms) >= 1 then
        begin
          v[0] := MakeParam(prms[0],Param1);
        end;
        if Length(prms) >= 2 then
        begin
```

拡張RTTIと属性 (23)

```
          v[1] := MakeParam(prms[1],Param2);
        end;
        if Length(prms) >= 3 then
        begin
          v[2] := MakeParam(prms[2],Param3);
        end;
        if Length(prms) >= 4 then
        begin
          v[3] := MakeParam(prms[3],Param4);
        end;
      end;
      mtd.Invoke(obj,v);
    end;
  finally
    ctx.Free;
  end;
end;
```

拡張RTTIと属性 (24)

```
function MakeParam(prm: TRttiParameter; const Value: String): TValue;
var
  EnumValue: Integer;
begin
  case prm.ParamType.TypeKind of
    tkInteger: Result := StrToInt(Value);
    tkChar:    Result := TValue.From<AnsiChar>(AnsiChar(Value[1]));
    tkFloat:   Result := StrToFloat(Value);
    tkString:  Result := TValue.From<ShortString>(ShortString(Value));
    tkWChar:   Result := Value[1];
    tkLString: Result := TValue.From<AnsiString>(AnsiString(Value));
    tkWString: Result := WideString(Value);
    tkInt64:   Result := StrToInt64(Value);
    tkUString: Result := Value;
    tkEnumeration:
    begin
      EnumValue := GetEnumValue(prm.ParamType.Handle,Value);
      if EnumValue >= 0 then
      begin
        Result := TValue.FromOrdinal(prm.ParamType.Handle,EnumValue);
      end;
    end;
  end;
end;
```

拡張RTTIと属性 (25)

- どのような場合に拡張RTTIを使うといいのでしょうか？
  - クラスに対する汎用な処理の記述
    →ORマッパやXMLへのシリアライズ/デシリアライズ
    →クラス構造のツリー表示

拡張RTTIと属性 (26)

- 拡張RTTIを使うときに気をつけたほうがいいこと
  - 拡張RTTIを扱うコードは遅い。
    - PropInfoプロパティ(TPropInfo構造体へのポインタPPropInfo)を保持しておくことで拡張RTTIに頼る部分を減らし、パフォーマンスを向上することができます。
    - 詳細はtales(Lynaたん)さんのblogで解説されています。
      - http://d.hatena.ne.jp/tales/20110820/1313852303
  - 配列に対するサポートが(まだ)不足している
    - 静的配列のフィールドはうまく扱えません。
    - 動的配列のフィールドは通常のプロパティ並に扱えるので、配列プロパティ、静的配列のフィールドの代替として動的配列のフィールドを用意してエイリアス的に使うという回避策も有効です。

拡張RTTIと属性 (27)

- 属性による注釈付け (1)
  - クラス型あるいはレコード型そのものに属性(attribute)で注釈を付ける(annotation)ことができます。
  - クラス型あるいはレコード型のメンバ(フィールド、プロパティ、メソッド)にも属性で注釈を付けることができます。
  - カスタム属性(custom attribute)を宣言して使います。
    - (プロパティやフィールドの値ではなく)属性クラスの型で区別します。
      →例外ハンドラを記述するときに例外オブジェクトの型で区別を行うのと同様です。
    - コンストラクタで渡した値(定数のみ)をフィールドまたはプロパティに保存して参照することもできます。
    - `TCustomAttribute`クラスから派生したカスタム属性クラスを宣言して使用します。

拡張RTTIと属性 (28)

- 属性による注釈付け (2)
  - 実行時にクラス型やレコード型、あるいはそのメンバに付けられている属性を抽出することができます。
  - .NET Frameworkと同様の記法を使います。

```
[<Attr>]
[<Attr>(<parameterlist>)]
```

  - 末尾の“Attribute”、コンストラクタの“.Create”を省略できます。

```
[<Attr>Attribute.Create]
[<Attr>Attribute.Create(<parameterlist>)]
```

拡張RTTIと属性 (29)

- 属性による注釈付け (3)
  - 属性のコンストラクタ
    - 継承元となるTCustomAttributeのコンストラクタはパラメータを持ちませんが、派生したクラスでは別形式のコンストラクタを定義することで値を渡すことができます(フィールドやプロパティでその値を保持します)。

```
constructor Create(const AFooValue: String);
```

    - コンストラクタのパラメータには定数しか使えません(文字列定数はOK)。
    - ポインタであっても定数なら使えるはずですが、実際には内部エラーが発生してコンパイルできません。

拡張RTTIと属性 (30)

- 属性による注釈付け (4)
  - 属性は検索、抽出されるときに実体が生成されます。
    →検索、抽出しなければ性能上のペナルティはありません。
    →実行バイナリ、占有メモリのサイズのペナルティはあります(属性クラスのコードや属性情報)。

- 属性はどのような状況で使うのでしょうか？
  - 拡張RTTIによる処理で…
    - 同じ型から派生していても区別して処理したいとき。
    - 同じ型のメンバでも区別して処理したいとき。

拡張RTTIと属性 (31)

- サンプル 5: 属性による注釈付け

```
type
  DateTimeAttribute = class(TCustomAttribute);
  DateAttribute = class(TCustomAttribute);
  TimeAttribute = class(TCustomAttribute);
  TTestData = class(TObject)
  private
    FPlace: String;
    FTemperature: Double;
    [DateTime]
    FDateTime: TDateTime;
    function  GetDate: TDateTime;
    procedure SetDate(const Value: TDateTime);
    function  GetTime: TDateTime;
    procedure SetTime(const Value: TDateTime);
  public
    constructor Create(APlace: String; ATemperature: Double; ADateTime: TDateTime);
    property Place: String read FPlace write FPlace;
    property Temperature: Double read FTemperature write FTemperature;
    [DateTime]
    property DateTime: TDateTime read FDateTime write FDateTime;
    [Date]
    property Date: TDateTime read GetDate write SetDate;
    [Time]
    property Time: TDateTime read GetTime write SetTime;
  end;
```

拡張RTTIと属性 (32)

```
v := prp.GetValue(obj);
case v.Kind of
  // ...
  tkFloat:
  begin
    if FindAttribute(prp.GetAttributes,DateTimeAttribute) = True then
    begin
      Str := Str + FormatDateTime('yyyy/mm/dd hh:nn:ss.zzz',v.AsExtended);
    end
    else if FindAttribute(prp.GetAttributes,DateAttribute) = True then
    begin
      Str := Str + FormatDateTime('yyyy/mm/dd',v.AsExtended);
    end
    else if FindAttribute(prp.GetAttributes,TimeAttribute) = True then
    begin
      Str := Str + FormatDateTime('hh:nn:ss.zzz',v.AsExtended);
    end
    else
    begin
      Str := Str + v.ToString;
    end;
  end;
  // ...
end;
```

拡張RTTIと属性 (33)

```
function FindAttribute(attrs: TArray<TCustomAttribute>; attrcls: TClass): Boolean;
var
  attr: TCustomAttribute;
begin
  for attr in attrs do
  begin
    if attr is attrcls then
    begin
      Result := True;
      Exit;
    end;
  end;
  Result := False;
end;
```

拡張RTTIと属性 (34)

- TRttiInstanceTypeクラス (Rttiユニット)
  - Delphi XE2で変更。
  - GetImplementedInterfaces/ GetDeclaredImplementedInterfacesメソッド
    - 実装されているインタフェースを全て取得

```
function GetImplementedInterfaces: TArray<TRttiInterfaceType>;
function GetDeclaredImplementedInterfaces: TArray<TRttiInterfaceType>;
```

コンパイラの変更 (1)

- クラスコンストラクタ、クラスデストラクタ
  - Delphi 2010で導入。
  - クラス型、レコード型に定義できます。

```
class constructor Create;
class destructor Destroy;
```

  - コンパイラが呼び出しコードを自動生成します。
  - そのクラス/レコード型が使用されている場合だけ呼び出されます。
  - initialization/finalization部にコードを記述することで引き起こされる不必要なコード、データのリンクを回避できます。
  - ジェネリックスクラス/レコードでは複数回呼び出されることがあります。

コンパイラの変更 (2)

- delayed指令によるDLLの遅延ロード
  - Delphi 2010で導入。
  - delayed指令を付加したexternal関数は、最初に呼び出されたときにロードとエントリポイントの取得が行われます。
  - SetDliFailureHookメソッド(Systemユニット)でロード、エントリポイントの取得に失敗したときの処理に介入できます。
  - Allen BauerさんによるDelayExceptユニットを使うとエラーがDLL名や関数名を持つ例外に変換されて送出されるので便利です。
    - http://blogs.embarcadero.com/abauer/2009/08/29/38896

コンパイラの変更 (3)

- {$SCOPEDENUMS ON}コンパイラ指令
  - Delphi 2010で導入。
  - .NET Frameworkのように列挙型の値シンボルにスコープ(列挙型名)をつけることを強制します。
  - 部分範囲型、集合型の基本型にする場合は{$SCOPEDENUMS ON}として定義できません。
  - 列挙型の値シンボルにプレフィックスをつける従来のやりかたと、{$SCOPEDENUMS ON}によりプレフィックスがなくてもスコープを強制することで曖昧さが排除できるやりかたのどちらも選べるようになりました。

```
Label1.Alignment := taLeftJustify;
FSupported := TUncertainState.Yes;
```

"Yes"という名前が他の列挙型と重複していても問題ない

ユニットスコープ名の導入 (1)

- ユニットスコープ名 (1)
  - Delphi XE2で導入。
  - ライブラリのユニットがカテゴリ毎に分類されたユニットスコープに所属するようになりました。
    - データベース: `Bde, Data, Data.Bind, Data.Cloud, Data.Win, Datasnap, Datasnap.Win, Db, IB`
    - FireMonkey: `Fmx, Fmx.Bind`
    - Mac OS X: `Macapi, Posix, System.Mac`
    - システム/ランタイム: `System, System.Bindings, System.Internal, System.Win`
    - VCL: `Vcl, Vcl.Bind, Vcl.Imaging, Vcl.Samples, Vcl.Shell, Vcl.Touch`

ユニットスコープ名の導入 (2)

- ユニットスコープ名 (2)
  - SOAP/COM: `Soap`
  - Web: `Web, Web.Internal, Web.Win`
  - Windows: `Winapi`
  - XML: `Xml, Xml.Internal, Xml.Win`
  - 基本的にはプロジェクトオプションの“Delphiコンパイラ|ユニットスコープ名”の設定により補完されるので、気にする必要はありません。
  - 完全修飾識別子名だけは例外的にユニットスコープ名を含めることを強制されますので注意が必要です。

基本型の変更

- Delphi XE2で64bit環境(Windows x64)がサポートされるようになりました。
- ほとんどの型のサイズ、範囲はそのままです。
- NativeInt、NativeUInt、ポインタ(クラス型や文字列型の変数を含む)、動的配列のインデックスは環境により32bitと64bitのどちらかになります。
- 64bit環境下では、Extended(80bit実数型)はDouble(64bit実数型)に置き換えられ、80bit実数型が必要な場合はTExtended80Recレコード型によるエミュレーションを使用します。

RTLの強化 (1)

- Delphi 2007~XE2のRTLの強化の特徴
  - Delphi for .NETに由来する、あるいは.NET Framework (CLR)にインスパイアされた諸機能が追加されています。
  - 高度なレコード型が多用されています。
    - ネームスペースあるいはアセンブリの代わりに型名を使用しているもの。

      例: `TDirectory` (IOUtilsユニット)
    - クラスのインスタンスは必ずヒープに配置され、(生成ー破棄を管理しなければならないなど)それなりに手間が掛かるのに対して、スタックに配置(自動変数)できて領域の確保、解放が簡単という特徴を利用しているもの。

      例: `TStopwatch` (Diagnosticsユニット)

RTLの強化 (2)

- TStopWatchレコード型 (Diagnosticsユニット)
  - Delphi 2010で導入。
  - “高解像度ストップウォッチ”。
  - 所要時間の計測などに便利です。
- TDirectory、TFile、TPathレコード型 (IOUtilsユニット)
  - Delphi 2010で導入、Delphi XEで強化。
  - ディレクトリ、ファイル、パス操作をまとめてあります。
  - 機能的にはあともう一歩かもしれません…。
- TTimeSpanレコード型 (TimeSpanユニット)
  - Delphi 2010で導入。
  - 従来のTDateTimeではうまくいかない部分もあった、時間間隔を扱うための型です。

RTLの強化 (3)

- TTimeZoneクラス型 (DateUtilsユニット)
  - Delphi XEで導入。
  - タイムゾーン情報を扱います。
  - 現在のタイムゾーン情報を示すクラスプロパティLocalを使用します。
  - Windows Vista SP1以降であればWin32 APIのGetTimeZoneInformationForYear関数を使用してそれなりな結果を得られます(このAPIの仕様そのものが微妙なので、“それなり”にしかなりませんが…)。

RTLの強化 (4)

- TThread.CreateAnonymousThreadメソッド (Classesユニット)
  - Delphi XEで導入。
  - パラメータで指定された無名メソッドを実行するスレッドを生成します。
- SplitStringメソッド (StrUtilsユニット)
  - Delphi XEで導入。
  - TStrings(TStringList)を使わなくても文字列を区切り文字で分割できるようになりました。
  - 分割結果はTStringDynArray(= array of string)なので、for..in文に渡すことができます。

RTLの強化 (5)

- TRegExレコード型 (RegularExpressionsユニット)
  - Delphi XEで導入。
  - 正規表現をサポートします。
  - PCRE(Perl Compatible Regular Expressions)に準拠しています。

  - 小宮秀一さんのSkRegExpもお勧めです。
    - http://komish.com/softlib/skregexp.htm
  - おなじみ富永さんによるTRegExとSkRegExpの比較です。
    - http://ht-deko.minim.ne.jp/tech064.html

RTLの強化 (6)

- TOSVersionレコード型 (SysUtilsユニット)
  - Delphi XE2で導入。
  - OSのバージョンやCPUのアーキテクチャを扱います。
  - 動作環境のバージョンを確認するにはCheckメソッドを使用します(SysUtilsユニットのCheckWin32Version関数に相当しますが、クロスプラットフォームに対応しています)。

```
class function Check(AMajor: Integer): Boolean; overload; static;
  inline;
class function Check(AMajor, AMinor: Integer): Boolean; overload;
  static; inline;
class function Check(AMajor, AMinor, AServicePackMajor: Integer):
  Boolean; overload; static; inline;
```

RTLの強化 (7)

- TSingleRec/TDoubleRec/TExtendedRec/TExtended80Rec レコード型 (Systemユニット)
  - Delphi XE2で導入。
  - 実数型(Single、Double、Extended)の低レベル操作を可能にします。
  - 64bit環境における拡張精度(80bit実数型)を提供します。
- TZipFileクラス型 (Zipユニット)
  - Delphi XE2で導入。
  - Zip形式ファイルをサポートします。
  - ExtractZipFileクラスメソッドでZip形式のファイルを簡単に展開できます。

```
class procedure ExtractZipFile(ZipFileName: string; Path: string);
  static;
```

クロスプラットフォーム (1)

- Delphi XE2でクロスプラットフォーム開発への対応がサポートされました。
- VCLアプリケーションはWindows (x86/x64)専用です。
- FireMonkey(FMX)アプリケーションはWindows/Mac OS X/iOSのいずれにも対応します。
- iOSについてはMac上のXcodeでFPC(Free Pascal Compiler)を使ってビルドする必要があります。
- IDEの配置マネージャとプラットフォームアシスタント(paserver)により、クロスプラットフォーム(Windows x64/Mac OS X)のデバッグが効率的にできるようになりました。

クロスプラットフォーム (2)

- ターゲットプラットフォームへの依存
  - 標準条件シンボルによる条件付コンパイルでターゲットプラットフォームに依存するコードを分離します。

    MSWINDOWS: Windows (x86/x64)

    WIN32: Windows x86

    WIN64: Windows x64

    MACOS: Mac OS X

    MACOS32: Mac OS X (32bit)

    POSIX: POSIX (Mac OS Xを含む)

    POSIX32: POSIX (32bit)

    FPC: iOS(FPC)

    FPCCOMP: iOS(FPC)?

    BIGENDIAN: ビッグエンディアン環境?

Free Pascal Complierでのみ宣言されています

クロスプラットフォーム (3)

- System.SysUtilsユニットの例

```
uses
{$IFDEF MSWINDOWS}
  Winapi.Windows,
{$ENDIF MSWINDOWS}
{$IFDEF POSIX}
  System.Types, Posix.Base, Posix.Dirent, Posix.Dlfcn, Posix.Fcntl, Posix.Errno,
Posix.Langinfo,
  Posix.Locale, Posix.Pthread, Posix.Signal, Posix.Stdio, Posix.Stdlib, Posix.String_,
  Posix.SysStat, Posix.SysTime, Posix.SysTypes, Posix.Time, Posix.Unistd, Posix.Utime,
  Posix.Wchar, Posix.Wctype, Posix.Wordexp, Posix.Iconv,
  System.Internal.Unwinder,
{$ENDIF POSIX}
{$IFDEF MACOS}
  Macapi.Mach, Macapi.CoreServices, Macapi.CoreFoundation,
{$ENDIF MACOS}
  System.SysConst;
```

embarcadero
Developer Camp

Additional time

クラスヘルパの拡張 (1)

- クラスヘルパはDelphi 2007で導入されました。
- クラスを継承することなく、バイナリレベルでの互換性を保ったままで拡張するための機能です。
  - Delphi 2007でDelphi 2006とのバイナリ(*.dcu)の互換性を保ちつつTFormやTApplicationを拡張するために作られました。
- 識別子を解決するときにスコープの範囲を拡張します。
  - 対象のクラスとクラスヘルパではクラスヘルパが優先されます。
- 特定のクラスに対してはたかだか1つのクラスヘルパしか適用できませんが、クラスヘルパを継承した別のクラスヘルパを定義することができます。

クラスヘルパの拡張 (2)

- 継承が許されない場合、従来は...

```
type
  TFoo = class(TObject)
  private
    FBar: Integer;
  public
    property Bar: Integer read FBar write FBar;
  end;

function GetBarAsString(AFoo: TFoo): String;
begin
  Result := IntToStr(AFoo.Bar);
end;

procedure SetBarAsString(AFoo: TFoo; const Value: String);
begin
  AFoo.Bar := StrToInt(Value);
end;

// ...
  S := GetBarAsString(Foo);  // TFooのインスタンスをパラメータとして渡す
```

クラスヘルパの拡張 (3)

- クラスヘルパを使うと...

```
type
  TFooHelper = class helper for TFoo
  private
    function GetBarAsString: String;
    procedure SetBarAsString(const Value: String);
  public
    property BarAsString: String read GetBarAsString write SetBarAsString;
  end;

function TFooHelper.GetBarAsString: String;
begin
  Result := IntToStr(Bar);
end;

procedure TFooHelper.SetBarAsString(const Value: String);
begin
  Bar := StrToInt(Value);
end;

// ...
  S := Foo.BarAsString; // あたかもTFooのメンバであるかのように扱える
```

クラスヘルパの拡張 (4)

- Delphi 2010でレコード型に対しても適用することができるようになりました(レコードヘルパ)。
  - Delphi XE2のRTL/VCLでは以下のものが定義されています(括弧内は対象となるレコード型)。
    - System.SyncObjsユニット
      `TCriticalSectionHelper (TRTLCriticalSection)`
      `TConditionVariableHelper (TRTLConditionVariable)`
    - System.Mac.CFUtilsユニット
      `CFGregorianDateHelper (CFGregorianDate)`
    - System.SysUtilsユニット
      `TGuidHelper (TGUID)`
    - Winapi.D2D1ユニット
      `TD2DMatrix3x2FHelper (TD2DMatrix3X2F)`
    - Vcl.Themesユニット
      `TElementMarginsHelper (TElementMargins)`

クラスヘルパの拡張 (5)

- クラスヘルパ/レコードヘルパでアクセスできる範囲
  - Delphi 2007ではpublicメンバにのみアクセスできます。
  - Delphi 2009では“Self.”で修飾することでprivateメンバにもアクセスできます(protectedはなぜかアクセス不可)。
  - Delphi 2010以降では“Self.”で修飾することでprivateおよびprotectedメンバにもアクセスできます。

- クラス型やレコード型を拡張する上で非常に強力な手段です。
- 使いすぎに注意しましょう。

- Delphi XE3では基本型(整数型、実数型)や文字列型にもレコードヘルパを適用できるようになるようです。

TVirtualMethodInterceptor (1)

- TVirtualMethodInterceptorはDelphi XEで導入されました。
- RTTI上にあるVMT(Virtual Method Table、仮想メソッドテーブル)を動的に置き換えることで、任意のインスタンスの仮想メソッドの呼び出し前、呼び出し後、例外発生時のそれぞれの時点に介入することができます。
  - 静的メソッド、動的メソッドには効果がありません。
- Proxifyメソッドでインスタンスの RTTIを書き換えて仮想メソッドのインターセプトを有効にします。

TVirtualMethodInterceptor (2)

- Delphi XEではOriginalClassプロパティを書き戻すことで、XE2ではUnproxifyメソッドでインターセプトを無効にします。
  - インターセプタを解放する時点で、インターセプトが有効になったままのインスタンスが存在しないようにします(デストラクタが仮想メソッドなので、インターセプトが有効だとインターセプタのインスタンスが参照されてしまうため)。

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
//  PPointer(FFoo)^ := FVmi.OriginalClass;  // Delphi XE
  FVmi.Unproxify(FFoo);
  FVmi.Free;
  FFoo.Free;
end;
```

浮動小数点演算の精度の制御 (1)

- x86
  - 実数型(浮動小数点数)は常にFPU(x87)の拡張精度(80bit)で扱われます。
  - `Single`(32bit)や`Double`(64bit)は自動的に`Extended`(80bit)に昇格され、計算後に必要な精度に再変換されます。
- x64
  - 実数型をSSE2で扱うため、拡張精度(80bit)はライブラリによるエミュレーションになります(前述)。

浮動小数点演算の精度の制御 (2)

- x64
  - 単精度(32bit)を扱うときは、通常は演算毎に倍精度(64bit)への自動昇格と再変換が行われますが、SSE2の命令セットでは単精度のみの演算が可能なため、単精度のみの演算で十分な場合はオーバヘッドが生じます。

```
var
  a, b, c, d: Single;
begin
  d := a * b * c;  // デフォルトではDoubleへの変換3回+Singleへの変換1回が自動的に行われてしまう
```

  - {$EXCESSPRECISION OFF}コンパイラ指令で計算の中間結果の精度の自動変換を抑止できます。
  - x64以外では{$EXCESSPRECISION OFF}は意味を持ちません。

浮動小数点演算の精度の制御 (3)

- $EXCESSPRECISIONコンパイラ指令の効果

```
{$EXCESSPRECISION OFF}
Project1.dpr.20: d := a * b * c;
000000000042800B F30F100535F20000   movss xmm0,dword ptr [rel $0000f235]
0000000000428013 F30F590531F20000   mulss xmm0,dword ptr [rel $0000f231]
000000000042801B F30F59052DF20000   mulss xmm0,dword ptr [rel $0000f22d]
0000000000428023 F30F110529F20000   movss dword ptr [rel $0000f229],xmm0

{$EXCESSPRECISION ON} (Default)
Project1.dpr.25: d := a * b * c;
000000000042804D F3480F5A05F2F10000 cvtss2sd xmm0,qword ptr [rel $0000f1f2]
0000000000428056 F3480F5A0DEDF10000 cvtss2sd xmm1,qword ptr [rel $0000f1ed]
000000000042805F F20F59C1           mulsd xmm0,xmm1
0000000000428063 F3480F5A0DE4F10000 cvtss2sd xmm1,qword ptr [rel $0000f1e4]
000000000042806C F20F59C1           mulsd xmm0,xmm1
0000000000428070 F2480F5AC0         cvtsd2ss xmm0,xmm0
0000000000428075 F30F1105D7F10000   movss dword ptr [rel $0000f1d7],xmm0
```

embarcadero
Developer Camp

参考文献

参考文献 (1)

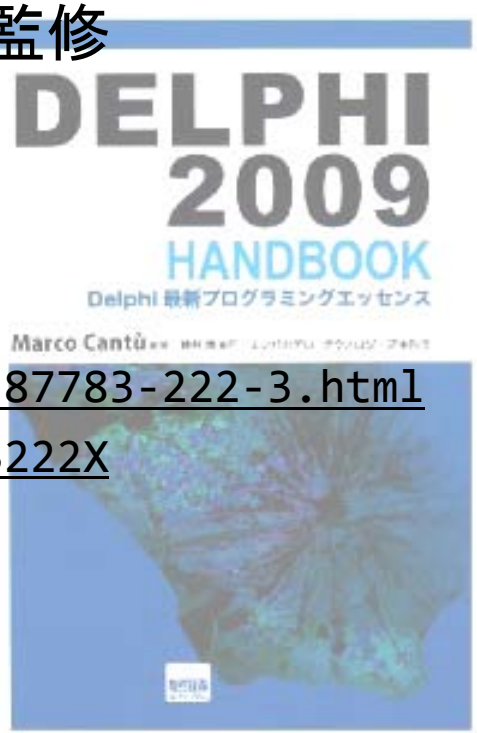
- エンバカデロ・テクノロジーズ公式
  - Delphi 2009 および C++Builder 2009 の新機能
    - http://docwiki.embarcadero.com/RADStudio/XE/ja/Delphi_2009_%E3%81%8A%E3%82%88%E3%81%B3_C%2B%2BBuilder_2009_%E3%81%AE%E6%96%B0%E6%A9%9F%E8%83%BD
  - Delphi 2010 および C++Builder 2010 の新機能
    - http://docwiki.embarcadero.com/RADStudio/2010/ja/Delphi_2010_%E3%81%8A%E3%82%88%E3%81%B3_C%2B%2BBuilder_2010_%E3%81%AE%E6%96%B0%E6%A9%9F%E8%83%BD
  - Delphi XE および C++Builder XE の新機能
    - http://docwiki.embarcadero.com/RADStudio/ja/Delphi_XE_%E3%81%8A%E3%82%88%E3%81%B3_C%2B%2BBuilder_XE_%E3%81%AE%E6%96%B0%E6%A9%9F%E8%83%BD

参考文献 (2)

- エンバカデロ・テクノロジーズ公式
  - Delphi XE2 および C++Builder XE2 の新機能
    - http://docwiki.embarcadero.com/RADStudio/ja/Delphi_XE2_%E3%81%8A%E3%82%88%E3%81%B3_C%2B%2BBuilder_XE2_%E3%81%AE%E6%96%B0%E6%A9%9F%E8%83%BD
  - RAD Studioマイグレーションセンター
    - http://www.embarcadero.com/jp/rad-in-action/migration-upgrade-center
  - デベロッパーキャンプ・アーカイブ - 開催順
    - http://www.embarcadero.com/jp/developer-camp-japan/archive

参考文献 (3)

- DELPHI 2009 HANDBOOK - Delphi最新プログラミングエッセンス
  - Marco Cantu著
  - 藤井 等訳/エンバカデロテクノロジーズ監修
  - カットシステム
  - ISBN978-4-87783-222-3
  - 5,250円
    - http://www.cutt.co.jp/book/978-4-87783-222-3.html
    - http://www.amazon.co.jp/dp/487783222X



参考文献 (4)

- Delphi 2007 Handbook
  - Marco Cantu著
  - CreateSpace
  - ISBN978-1442147034
  - 38.50USD (amazon.co.jpだと3,360円)
    - http://www.marcocantu.com/dh2007/
    - http://www.amazon.com/dp/1442147032
    - http://www.amazon.co.jp/dp/1442147032
  - ebook版は25.00USD
    - http://sites.fastspring.com/wintechitalia/product/delphi2007ehandbook

参考文献 (5)

- Delphi 2009 Handbook
  - Marco Cantu著
  - CreateSpace
  - ISBN978-1440480096
  - 48.50USD (amazon.co.jpだと3,953円)
    - http://www.marcocantu.com/dh2009/
    - http://www.amazon.com/dp/1440480095
    - http://www.amazon.co.jp/dp/1440480095
  - ebook版は30.00USD
    - http://sites.fastspring.com/wintechitalia/product/delphi2009ehandbook

参考文献 (6)

- Delphi 2010 Handbook
  - Marco Cantu著
  - CreateSpace
  - ISBN978-1450597265
  - 43.50USD (amazon.co.jpだと3,830円)
    - http://www.marcocantu.com/dh2010/
    - http://www.amazon.com/dp/1450597262
    - http://www.amazon.co.jp/dp/1450597262
  - ebook版は28.00USD
    - http://sites.fastspring.com/wintechitalia/product/delphi2010handbook

参考文献 (7)

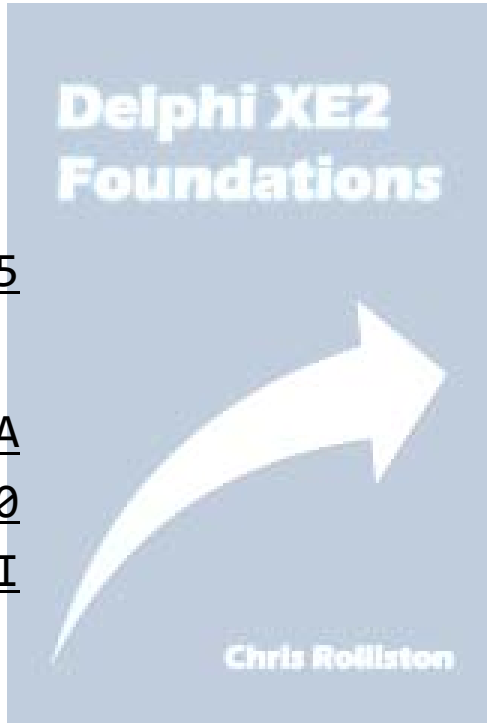
- Delphi XE Handbook
  - Marco Cantu著
  - CreateSpace
  - ISBN978-1463600679
  - 34.50USD (amazon.co.jpだと2,812円)
    - http://www.marcocantu.com/handbooks/#dxeh
    - http://www.amazon.com/dp/1463600674
    - http://www.amazon.co.jp/dp/1463600674
  - ebook版は20.00USD
    - http://sites.fastspring.com/wintechitalia/product/delphixehandbook

参考文献 (8)

- Delphi Handbooks Collection, Second Edition (2007 to XE)
  - Marco Cantu著
  - ebook版(PDF)のみ
  - Delphi Handbookシリーズ(2007/2009/2010/XE)をまとめたもの
  - 74.50USD (約6,090円)
    - http://sites.fastspring.com/wintechitalia/product/delphihandbookscollection

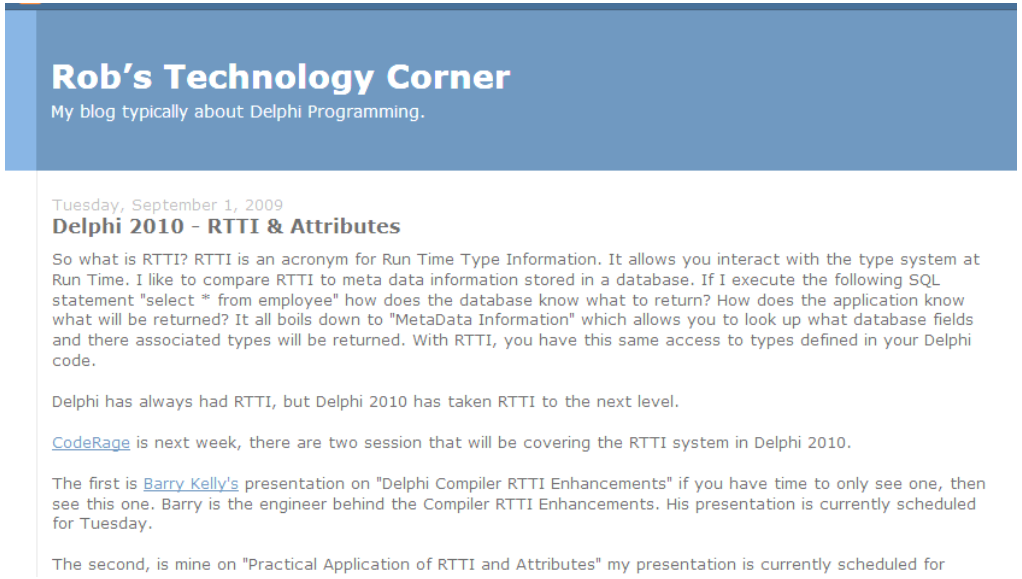
参考文献 (9)

- Delphi XE 2 Foundations
  - Chris Rolliston著
  - CreateSpace
  - ISBN978-1477550892
  - 49.99USD (約3,921円)
    - http://delphifoundations.com/
    - http://www.amazon.com/dp/1477550895
  - Kindle版は3分冊で各9.99USD
    - http://www.amazon.com/dp/B008BNNOBA
    - http://www.amazon.com/dp/B008BNV5R0
    - http://www.amazon.com/dp/B008BO0TFI



参考文献 (10)

- “Rob’s Technology Corner”
  - “Delphi 2010 - RTTI & Attributes”シリーズ
    - http://robstechcorner.blogspot.com/2009/09/so-what-is-rtti-rtti-is-acronym-for-run.html
  - 拡張RTTIと属性について詳しく解説されています。
  - 英語です。



embarcadero
Developer Camp

Q & A

embarcadero
Developer Camp

ありがとうございました