أساسيات باسكال

ماركو كانتو

ترجمة
خالد الشقروني

مقدمة

ماركو كانتو: أساسي باسكال

1اكتوبر 1999: الكتاب معروض في القرص المرافق لدلفي 5 .
التوليف المصدري متوفر .أضيفت قائمة بالأمثلة.

غلاف الكتاب. تقديس ابولو في دلفي،
في لوحة ايطالية من القرن 17

الاصدارات الأولى من كتاب التحكم بدلفي Mastering Delphi ، افضل كتاب دلفي مبيعا قمت بكتابته، كانت تحوي على مقدمة عن لغة باسكال في دلفي. و بسبب قيود الحجم و لأن الكثير من مبرمجي دلفي يبحثون عن معلومات اكثر تقدما، فقد تم في الاصدارات اللاحقة حذف هذا الجزء بالكامل. و حتى يتم تلافي غياب هذه المعلومات، كنت قد بدأت بوضعها في كتاب على الخط، وكان عنوانه ***أساسي باسكال*** .

هذا كتاب مفصّل عن باسكال، و الذي حتى هذه اللحظة سيكون متوفرا مجانا في موقعي على الشبكة (حقيقة لا أعرفر ماذا سيحدث لاحقا، فقد أجد ناشرا). هذا عمل تحت الانجاز، وكل تعليق مرحب به. النسخة الأولى المكتملة من هذا الكتاب، بتاريخ يوليو99 ، نشرت في القرص المرافق لدلفي 5 .

حقوق النسخ

تقسيم الكتاب

فيما يلي التقسيم الحالي للكتاب :

التوليف المصدري

التوليف المصدري لكل الأمثلة المشار اليها في الكتاب متوفرة. التوليف له نفس حقوق النسخ التي للكتاب: يمكنك استخدامه بحرية ايضا لكن بدون نشره في وثائق أخرى او موقع آخر. الوصل بهذا الموقع مرّحب به .

قم بتنزيل التوليف المصدري في ملف مضغوط zip واحد، **EPasCode.zip** (فقط بحجم 26 ك ب) و راجع **قائمة الأمثلة** .

المقترحات والأراء

يرجى اعلامي بأي أخطاء قد تجدها، أيضا بأية موضوعات غير واضحة بصورة كافية للمبتدئ. سأكون قادرا على تخصيص وقت للمشروع بناء على المقترحات و الأراء التي استلمها. ايضا دعني اعرف ما هي الموضوعات الأخرى (لم تغطّى في كتاب التحكم بدلفي 4) التي تودّ أن تراها هنا. مرّة اخرى، تابع مجموعة الأخبارnewsgroup ، المنشورة في موقعي بالشبكة، أو راسل **marco@marcocantu.com** (بوضع ***Essential Pascal*** في خانة الموضوع، و طلبك او تعليقك في مساحة النص) .

شكر وعرفان

إذا قمت بنشر كتاب على الشبكة مجانا، فإن هذا غالبا بسبب تجربة بروس اكل Bruce Eckel مع كتابه التفكير بجافا *Thinking in Java*. انا صديق لبروس وأظنه قام حقا بعمل جيد في هذا الكتاب كما في غيره .

حالما قمت بابلاغ الناس في بورلاند عن المشروع استلمت ايضا الكثير من ردود الفعل الإيجابية. و بالطبع عليّ أن أشكر الشركة على صنعها أولا لسلسلة مجمّعات تربو باسكال و الآن سلسلة دلفي من بيئات التطوير المرئية .

بدأت بالحصول على بعض الأراء القيمة. أوائل القُرّاء الذين ساعدوا بدرجة كبيرة على تحسين هذه المادة هم شارلز وود Charles Wood و ويات وونغ Wyatt Wong. و ساعد مارك غرينهو Mark Greenhaw ببعض التحرير للنص. رافييل-دروج Barranco-Droege عرض العديد من التصحيحات الفنية والتقويم اللغوي. شكرا لهم .

الترجمات

اساسي باسكال تم ترجمته الى بعض اللغات الأخرى، متضمنة اليابانية الألمانية، والفرنسية. النسخ المترجمة ستكون متاحة مجانا على الخط و موصولة بهذا الموقع. اذا كنت مهتما بترجمة اساسي باسكال أو تبحث عن نسخة مترجمة راجع صفحة **الترجمات** .

المؤلف

ماركو كانتوMarco Cantù ؛ يعيش في بياتشينتزا Piacenza بايطاليا. بعد كتابته لكتب و مقالات في س++ و مكتبة كائنات ويندوز Object Windows Library ، عكف على البرمجة بدلفي. هو مؤلف سلسلة كتاب *التحكم بدلفي Mastering Delphi* ، من منشورات سيبكس Sybex ، كما ألف *دليل مطوّري دلفي Delphi Developers Handbook* المتقدم. كتب مقالات لعدة مجلات، ضمنها *مكازين دلفي The Delphi Magazine*، شارك بكلمات في مؤتمرات دلفي و بورلاند حول العالم، و قام بالتدريس في دورات دلفي للمستوى الأساسي و المتقدم .

يمكنك أن تجد مزيدا من التفاصيل عن ماركو و أعماله في موقعه بالشبكة،
www.marcocantu.com.

الفصل 1
تاريخ باسكال

ماركو كانتو:
أساسي باسكال

لغة اوبجيكت باسكال (Object Pascal) التي نستخدمها في دلفي لم يتم اختراعها في 1995 مع ظهور بيئة البرمجة المرئية لبورلاند. هي ببساطة امتداد للغة اوبجيكت باسكال التى كانت موجودة في منتجات باسكال السابقة لبورلاند. الا ان بورلاند لم تقم بابتكار باسكال، ولكنها فقط ساعدت على جعلها اكثر شعبية كما طورتها قليلا ..

هذا الفصل سوف يحوي خلفية تاريخية عن لغة باسكال وتطورها. في الوقت الحالي سيتضمن فقط نُبذ قصيرة جدا .

باسكال ويرذ

تم تصميم لغة باسكال في الأصل سنة 1971 من قبل نيكلوس ويرذ(Niklaus Wirth) ، البروفيسور في معهد زيوريخ التقني بسويسرا. وصممت باسكال بحيث تكون نسخة مبسطة لأغراض تعليمية من لغة أخرى هي الكول Algol ، التي يرجع تاريخها الى 1960 .

عندما تم تصميم باسكال، كانت توجد العديد من لغات البرمجة الأخرى، لكن القليل منها الذي انتشر استعماله: فورتران، س، اسيمبلر، كوبول. الفكرة الرئيسية في اللغة الجديدة كانت التنظيم. أي أن تكون لغة منظمة من خلال مفهوم قوي لأنواع البيانات، و الزام وجود تعريفات مسبقة، وتحكمات هيكلية للبرنامج. كما تم تصميم اللغة ايضا لتكون اداة تعليمية للطلبة في فصول البرمجة .

تربو باسكال

محول باسكال الأكثر شهرة عالميا من بورلاند ، يدعى تربو باسكال Turbo Pascal ، تم تقديمه في 1983، مراعيا فيه تنفيذ كتاب "دليل المستخدم والتقارير لباسكال" لكل من جينسن و ويرذ. وقد اصبح محول تربو باسكال احد اكثر المحولات مبيعا، واكسب اللغة شعبية خاصة بيئات الحواسيب الشخصية، ويرجع الفضل في ذلك الى الموائمة بين البساطة والقوة .

قدم تربو باسكال بيئة تطوير متكاملة (IDE) حيث يمكنك كتابة البرنامج (في محرر نصوص متوافق مع وورد ستار) ، ثم تقوم بتشغيل المحوّل، تتطّلع عل الاخطاء، تقفز مباشرة للعودة لأسطر البرنامج التي تحوي هذه الأخطاء. قد يبدو هذا شيئا عاديا الآن، لكن في السابق كان عليك ان تغلق محر النصوص الذي فيه برنامجك، تعود الى دوس؛ تقوم بتشغيل المحول ذو الأوامر السطرية، تكتب الأخطاء التي تظهر في ورقة خارجية، تعود لفتح محرر النصوص من جديد لتبحث فيه .

أكثر من هذا؛ قامت بورلاند ببيع تربو باسكال بسعر 49 دولار، في الوقت التي كانت فيه ميكروسفت تبيع محول الباسكال الخاص بها ببضع مئات. وقد كان لنجاح تربو باسكال على مدى سنوات الأثر في قرار ميكروسفت بوقف انتاجها لمحول باسكال الخاص بها .

باسكال دلفي

بعد 9 اصدارات من محولات تربو وبورلاند باسكال، والتي من خلالها تطورت اللغة تدريجيا، اصدرت بورلاند دلفي في 1995، ناقلة بذلك باسكال الى لغة برمجة مرئية .

دلفي مدّت في لغة باسكال في عدة مجالات، حيث اضافت بعض الخصائص ذات المنحى الكائني object-oriented و التي تختلف عن بعض المذاقات الأخرى لأوبجكت باسكال، حتى عن تلك التي في محوّل *Borland Pascal with Objects* compiler.

الفصل التالي :البرمجة بباسكال

الفصل 2
البرمجة بباسكال

ماركو كانتو:
أساسي باسكال

قبل ان ننتقل الى موضوع كتابة تعليمات لغة باسكال، من المهم أن نلقي الضوء على بعض عناصر نمط كتابة التعليمات بباسكل. المسألة التي أودّ الإشارة اليها هنا كالتالي: بجانب قواعد اللغة، ما هي الكيفية التي يجب عليك اتباعها لكتابة التعليمات. لاتوجد اجابة واحدة على هذا السؤال، حيث ان الأسلوب الشخصي يمكنه ان يقرر عدة انماط. عموما، هناك بعض المبادئ التي تحتاج لمعرفتها فيما يخصّ وضع التعليقات، حالة الأحرف، المسافات و ما يسمّى بالطباعة الأنيقة . pretty-printing كمبدأ عام، الهدف من أي نمط للكتابة هي الوضوح. ان النمط و القالب الذي تختاره هو شكل من اشكال الاختزال، يشير الى الغرض من جزء ما من التعليمات البرمجية. و الأداة الرئيسية للوصول الى الوضوح هي وحدة النسق بغضّ النظر عن النمط الذي تختاره، كن متأكدّا بأنكّ ستتّبع نفس النسق عبر كامل المشروع البرمجي .

التعليقات

في باسكال، يتم ضمّ التعليقات في أقواس braces أو أقواس parentheses متبوعة بنجمة. دلفي تقبل ايضا نمط التعليقات المتبعة في س++ ، والتي يمكنها ان تمتد الى نهاية السطر :

```
{هذا تعليق}
(* هذا تعليق آخر *)
// هذا تعليق آخر يمتد حتى نهاية السطر
```

الشكل الأول أقصر و أكثر اتّباعا. الشكل الثاني كان مفضلا أكثر في اوربا بسبب عدم وجود رمز القوس السهمي في لوحات المفاتيح. الشكل الثالث من التعليقات تم استعارته من س++ و متوفر فقط في نسخ 32 بت من دلفي. التعليقات المحدودة بنهاية السطر مفيدة جدا للملاحظات القصيرة و لتلك الخاصة بسطر محدد في التوليف .

> خلال سرد الأمثلة في هذا الكتاب سأحاول تعليم التعليقات بأحرف مائلة، (و الكلمات الرئيسية بالتغميق)، لتكون متسقة مع النمط الافتراضي للصياغة في دلفي .

وجود ثلاثة اشكال مختلفة من التعليقات يمكن ان يساعد في بناء تعليقات متداخلة .إذا اردت التعليق على مجموعة أسطر من برنامج من اجل وقفها، وهذه الأسطر تحوي بعض التعليقات السابقة، فإنك لاتستطيع استخدام نفس علامة التعليقات :

```
{  ... code
{comment, creating problems}
... code }
```

مع علامة تعليق ثانية، يمكنك كتابة التعليمات الآتية، و التي هي صحيحة :

```
{  ... code
//this comment is OK
... code }
```

لاحظ أن القوس المفتوح او القوس_نجمة اذا كان تليه علامة الدولار ($)، فسوف يتحول الى توجيه للمُجمّع compiler directive ، كما في .{$X+}

في الواقع، توجيهات المجمّع تعد تعليقات أيضا. مثال ذلك، {$X+ This is a comment} هي صحيحة. هي كلاهما توجيه و تعليق صحيحين، الا أن المبرمج المتعقل سوف يختار ان يفصل بين التوجيهات والتعليقات .

استخدام الأحرف العالية

مجمّع باسكال (بعكس اللغات الأخرى) يغضّ الطرف عن حالة الأحرف (عالية أو منخفضة). لذلك؛ فإن التعريفات التالية Myname ، MyName، myname، myName، و MYNAME كلها متساوية. بشكل عام، هذا يعدّ أمرا ايجابيا، ففي اللغات الحسّاسة لحالة الأحرف، العديد من الأخطاء اللغوية قد تحدث بسبب الإهمال في مراعاة حالة الأحرف .

ملاحظة :ربما الحالة الإستثناء الوحيدة لقاعدة حساسية الأحرف في باسكال هي: إجراء *Register*في حزمة المكونات، لابد لها أن تبدأ بحرف *R* العالي، وذلك للمحافظة على التوافقية مع C++ Builder.

إلا أنه توجد بعض السلبيات. أولا، يجب أن تنتبه لأن تكون هذه التعريفات متساوية بالفعل، لذا يجب أن تتجنب استعمالها كعناصر مختلفة. ثانيا، يجب أن تكون متّسقا قدر الامكان عند استخدامك للأحرف العالية، لتحسين مقروئية برنامجك .

توحيد استخدام حالة الأحرف ليس ملزما من قبل المجمّع، و لكنها عادة حسنة يحبّذ اتّباعها. الاسلوب المتبع هو تكبير الحرف الأول فقط من كل معرّف identifier. و عندما يكون المعرّف مركّبا من عدة كلمات (لايمكنك حشر فراغ في المعرّف) ، فإن كل أول حرف من كل كلمة يجب أن يكون عاليا :

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

هناك حالات أخرى لايأبه لها المجمّع كالفراغات، و الأسطر الفارغة، و المسافات (tabs) التي تقوم بوضعها في البرنامج. كل هذه الحالات مجتمعة تسمّى بالفراغ الأبيض . white space الفراغات البيضاء تستخدم فقط لتحسين مقروئية البرنامج؛ و لا تؤثر في عملية التجميع .

بعكس لغة بيسك BASIC ، فإن باسكال تسمح لك بكتابة تعليمة واحدة موزعة على عدة أسطر، فالتعليمة الطويلة يمكن تجزئتها لتكون في سطرين أو أكثر. السلبية الوحيدة) على الأقل بالنسبة لمبرمجي البيسك) لإمكانية أن تكون التعليمات في أكثر من سطر هي أنه عليك أن تتذكّر بأن تضع فاصلة منقوطة آخر كل تعليمة، أو بدقة أكثر، أن تفصل بين التعليمة والتي تليها. لاحظ أن القيد الوحيد هنا ان الجملة النصية الواحدة لايمكن مدّها لعدّة أسطر .

مرّة أخرى، لاتوجد قواعد ثابتة لاتستخدام الفراغات و التعليمات متعددة الأسطر، فقط بعض الأعراف :

- محرّر نصوص دلفي له خطا عموديا تستطيع وضعه على بعد 60 أو 70 حرف. إذا استخدمت هذا الخطّ كمؤشر لتجنب تجاوزه، فإن برنامجك سوف يبدو أفضل عندما تقوم بطباعته على الورق.

غير هذا فإن الأسطر الطويلة قد يتم تجزئتها من أي موضع حتى من منتصف الكلمة عند طباعة البرنامج .

- عندما يكون للوظيفة أو الإجراء عدة محددات parameters ، فإن العادة المتّبعة هنا هي وضع هذه المحدّدات في سطر آخر .
- تستطيع ترك سطر بكامله أبيضا (فارغا) قبل وضع أي تعليق أو ملاحظة، أو لتقسيم جزء كبير من التعليمات الى أجزاء أصغر. وحتى هذه الفكرة البسيطة بإمكانها تحسين مقروئية البرنامج، سواء على الشاشة أو عند طباعتها .
- استخدم الفراغات لفصل محددات استدعاء وظيفةfunction call ، وربما حتى فراغ قبل فتح الأقواس، ايضا حافظ على فراغات لفصل رموز العمليات في التعابير البرمجية. أنا أعلم أن بعض المبرمجين لن يوافقوا على هذه الفكرة، لكن أنا أصرّ: الفراغات بالمجان؛ لن تدفع شيئا مقابلها. (نعم، أعلم انها تستهلك مكانا للتخزين أو وقتا اضافيا في عملية اتصال الموديم لتحميل أو تنزيل ملف، لكن هذا أصبح لامعنى له في وقتنا الحاضر).

الطباعة الأنيقة

الاقتراح الأخير فيما يخصّ استخدام الفراغات البيضاء له علاقة بالعرف المتبع لنسق تشكيل لغة باسكال، و الذي يعرف بالطباعة الأنيقة .pretty-printing القاعدة بسيطة: كل مرة تحتاج فيها لكتابة تعليمة مركّبة، قم بوضعها بعد هامش فراغين الى اليمين من باقي التعليمة الحالية. التعليمة المركبة داخل تعليمة أخرى مركبة يتم تهميشها بأربع مسافات، وهكذا :

```
if ... then
  statement;

if ... then
begin
  statement1;
  statement2;
end;

if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

الصياغة السابقة تعتمد نسق الطباعة الأنيقة، لكن المبرمجين لديهم تفسيرات مختلفة لهذه القاعدة العامة. بعض المبرمجين مثلا يقومون بتهميش تعليمات begin و end للمستوى التالي مع نفس التعليمات الداخلية، بعضهم يهمش begin و end ثم يقومون بتهميش التعليمات الداخلية لمستوى اضافي، مبرمجون آخرون يضعون begin في نفس سطر شرط . if هذا في معظمه أمر له علاقة بالذوق الشخصي .

نفس نمط التهميش يتّبع عادة عند سرد المتغيرات أو أنواع البيانات، و لمواصلة تعليمة من سطر سابق :

```
type
  Letters = set of Char;
var
  Name: string;
begin
    { long comment and long statement, going on in the
      following line and indented two spaces }
   MessageDlg ('This is a message',
     mtInformation, [mbOk], 0);
```

بـالطّبع، أي مـن هذه القواعد هـي مجرّد إقتـراح لجعل البرنـامج مقروءا بشكل أفضـل مـن قبـل المبرمجين الآخرين، و هي تهمل بالكامل مـن جانـب المجمّـع. لقد حاولـت استخدام هذه القاعـدة بصورة متّسقة في كل اجزاء الأمثلة والبرامج في هذا الكتاب. كما يلاحظ أن البرامج، الأدلـة، و أمثلة المساعدة التي تأتي مع دلفي كلها تتّبع نفس النسق في الصياغة .

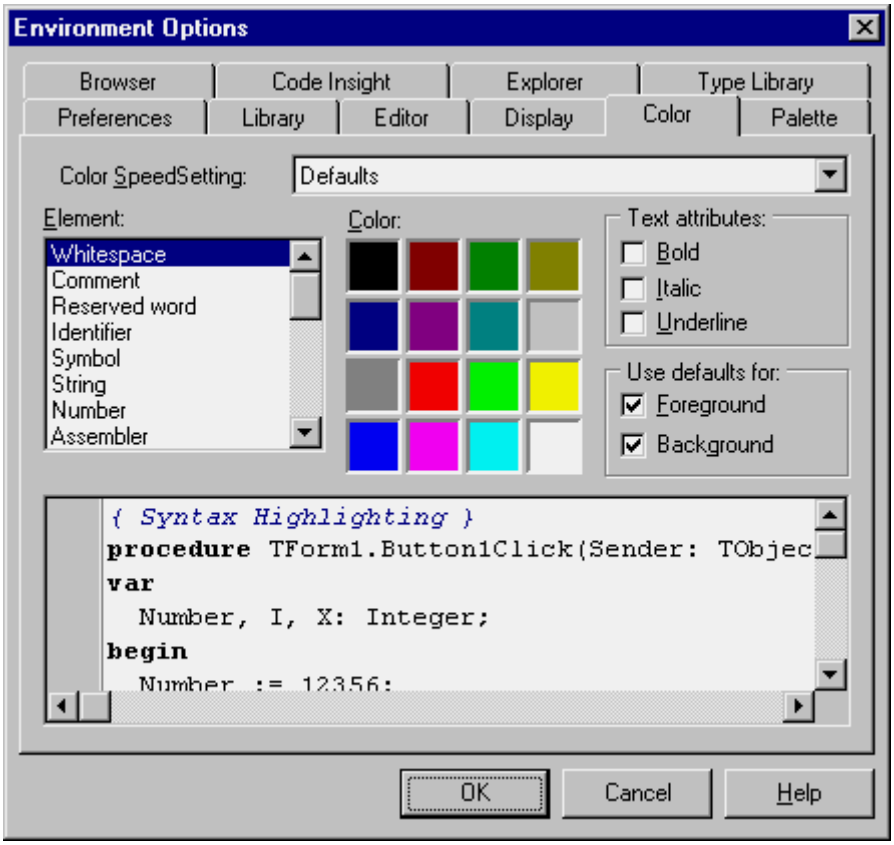
تعليم الألفاظ

لتسهيل قراءة و كتابة توليف code باسكال، يملك محرر دلفي خاصية تسمّى تعليم الألفاظ syntax highlighting . فالكلمات التي تقوم بطباعتها في المحرر، يتم اظهارها باستخدام ألوان مختلفة بحسب معناها في باسكال. عرضا، الكلمات المفتاحية keywords تكون داكنة، النصوص و التعليقات تظهر ملونة (و غالبا مائلة)، و هكذا .

الكلمات المحجوزة، و التعليقات، و النصوص تقريبا هي العناصر الثلاثة الأكثر استفادة من هذه الخاصية. فمن أول نظرة يمكنك ملاحظة كلمة مفتاحية غير صحيحة، أو نصّ غير مقفل بصورة سليمة، أو طول الملاحظة المتعددة الأسطر .

بإمكانك بسهولة تعديل مواصفات تعليم الألفاظ باستخدام صفحة ألوان المحرر Editor page في لوحة خيارات البيئة Environment Options انظر الشكل 2.1). إذا كنت تعمل بمفردك، يمكنك اختيار الألوان التي تفضل. أما إذا كنت تعمل بالتعاون مع مبرمجين آخرين، فالأفضل أن توافقوا جميعا على نسق ألوان نمطي. لقد وجدت ان العمل على حاسوب به تلوين الفاظ مختلف عمّا تعوّدت عليه أمر صعب بالفعل .

الشكل 2.1: لوحة الحوار المستخدمة لتحديد لون تعليم الألفاظ .



ملاحظة: في هذا الكتاب حاولت تطبيق ما يشبه تعليم الألفاظ على أمثلة البرامج. أتمنّى أن يجعلها بالفعل مقروءة بصورة أفضل .

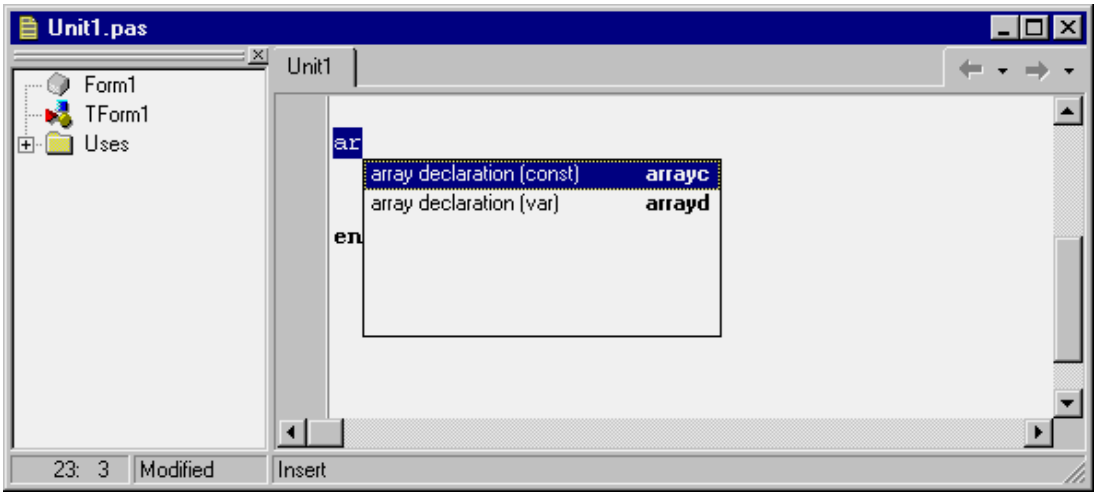
استخدام قوالب اللغة

قدّمت دلفي 3 خاصيّة جديدة ذات علاقة بكتابة شفرة البرامج. عند كتابة تعليمات لغة باسكال تجد نفسك عادة ما تعيد كتابة نفس التتابع من الكلمات الرئيسية، لذلك قدمت بورلاند خاصية جديدة تسمّى قوالب اللغة . Code Templates قوالب اللغة هي ببساطة قطعة من توليف مرتبطة بمفاتيح مختصرة. حيث تقوم بكتابة النص المختصر ثم تتبعها بالضغط علىCtrl+j ، فيظهر التوليف ذو العلاقة مكتوبا بالكامل. مثلا، اذا قمت بكتابة arrayd ، ثم ضغطت على Ctrl+j، فان محرر دلفي سوف يوسع من النص المختصر الى التالي :

```
array [0..] of ;
```

و حيث ان قوالب التوليف المحددة سلفا عادة ما تأتي بنسخ مختلفة لنفس الاختصار، فان النص المختصر ينتهي عموما بحرف يشير الى النسخة التي قد تهمك. عموما يمكنك كتابة فقط جزء من النص المختصر. مثال ذلك، اذا كتبت ar ثم ضغطت على Ctrl+j ، يظهر المحرر قائمة تظهر الخيارت المتوفرة مع وصف موجز لكل اختصار، مثلما هو واضح في الشكل 2.2 .

الشكل 2.2 اختيار قالب التوليف



تستطيع صياغة قوالب التوليف إما بتعديل الموجود منها، أو ببناء قوالب جديدة خاصة بك. واذا قمت بهذا، تذّكر ان نص قالب التوليف عادة ما يحوي حرف '|' ليشير الى الموقع الذي سيقفز له المؤشر بعد انتهاء العملية، حيث تتابع الكتابة لإكمال نص القالب .

تعليمات اللغة

حالما تقوم بتحديد بعض المعرفات، يمكنك استخدامها في تعليمات او في معادلات هي جزء من بعض التعليمات, تقدم باسكال مجموعة من التعليمات والتعبيرات. دعنا أولا نلقي نظرة على الكلمات المفتاحية، و التعبيرات ، و العاملات .

الكلمات المفتاحية

الكلمات المفتاحية هي كل المعرّفات المحجوزة من قبل اوبجكت باسكال، و التي لها دور في اللغة. دليل دلفي (Help) يميّز بين الكلمات المحجوزة والتوجيهات كالتالي :الكلمات المحجوزة لا

يمكن استخدامها كمعرّفات، بينما التوجيهات لا يجب استخدامها لنفس الغرض، حتى لو قبلها المجمّع. عند الممارسة، عليك تجنّب استخدام أية كلمة محجوزة كمعرّف .

في الجدول 2.1 يمكنك رؤية قائمة كاملة بالمعرّفات التي لها دورا خاصا في لغة اوبجكت باسكال (في دلفي 4)، بما في ذلك الكلمات و الكلمات المحجوزة الأخرى .

الجدول 2.1: الكلمات المفتاحية و الكلمات المحجوزة الأخرى في لغة اوبجكت باسكال

الكلمة الرئيسة	الدور
absolute	directive (variables) (متغير) موجّه
abstract	directive (method) (مسار) موجّه
and	operator (boolean) (بولي) معامل
array	type نوع
as	operator (RTTI) معامل
asm	statement تعليمة
assembler	backward compatibility (asm) توافقية مع السابق
at	statement (exceptions) (استثناءات) تعليمة
automated	access specifier (class) (طبقة) محدد دخول
begin	block marker علامة حيّز
case	statement تعليمة
cdecl	function calling convention قاعدة استدعاء وظيفة
class	type نوع
const	declaration or directive (parameters) (محددات) تعريف أو توجيه
constructor	special method مسار خاص
contains	operator (set) (فئةّ) عامل
default	directive (property) (سمة) توجيه
destructor	special method مسار خاص
dispid	dispinterface specifier محدد واجهة اطلاق
dispinterface	type نوع
div	operator عامل
do	statement تعليمة
downto	statement (for) (تعليمةfor(
dynamic	directive (method) (مسار) توجيه
else	statement (if or case) (تعليمةif او case(
end	block marker تعليم حيّز

except	statement (exceptions) تعليمة (استثناءات)
export	backward compatibility (class) توافقية مع السابق
exports	declaration تعريف
external	directive (functions) توجيه
far	backward compatibility (class) توافقية مع السابق
file	type نوع
finalization	unit structure بنية وحدة
finally	statement (exceptions) تعليمة
for	statement تعليمة
forward	function directive توجيه وظيفة
function	declaration تعريف
goto	statement تعليمة
if	statement تعليمة
implementation	unit structure بنية وحدة
implements	directive (property) توجيه (مسار)
in	operator (set) - project strucure بنية مشروع - عامل (فئة)
index	directive (dipinterface) توجيه
inherited	statement تعليمة
initialization	unit structure بنية وحدة
inline	backward compatibility (see asm) توافقية مع السابقة
interface	type نوع
is	operator (RTTI) عامل
label	declaration تعريف
library	program structure بنية برنامج
message	directive (method) توجيه (مسار)
mod	operator (math) عامل (رياضي)
name	directive (function) توجيه (وظيفة)
near	backward compatibility (class) توافقية مع السابق
nil	value قيمة
nodefault	directive (property) توجيه (مسار)
not	operator (boolean) عامل (بولي)
object	backward compatibility (class) توافقية مع السابق
of	statement (case) تعليمة

on	statement (exceptions) تعليمة
or	operator (boolean) (بولي) معامل
out	directive (parameters) (محدّدات) توجيه
overload	function directive توجيه وظيفة
override	function directive توجيه وظيفة
package	program structure (package) (حزمة) بنية برنامج
packed	directive (record)(تسجيلة) توجيه
pascal	function calling convention طرقة استدعاء وظيفة
private	access specifier (class) (طبقة) معيّن لوصول
procedure	declaration تعريف
program	program structure بنية برنامج
property	declaration تعريف
protected	access specifier (class) (طبقة) معيّن لوصول
public	access specifier (class) (طبقة) معيّن لوصول
published	access specifier (class) (طبقة) معيّن لوصول
raise	statement (exceptions) (اعتراضات) تعليمة
read	property specifier معيّن سمة
readonly	dispatch interface specifier معيّن واجهة ارسال
record	type نوع
register	function calling convention طريقة لاستدعاء وظيفة
reintroduce	function directive توجيه وظيفة
repeat	statement تعليمة
requires	program structure (package) (حزمة) بنية برنامج
resident	directive (functions) (وظيفة) توجيه
resourcestring	type نوع
safecall	function calling convention طريقة لاستدعاء وظيفة
set	type نوع
shl	operator (math) (رياضة) عامل
shr	operator (math) (رياضة) عامل
stdcall	function calling convention طريقة لاستدعاء وظيفة
stored	directive (property) (سمة) توجيه
string	type نوع
then	statement (if)

threadvar	declaration تعريف
to	statement (for) تعليمة
try	statement (exceptions) تعليمة (استثناءات)
type	declaration تعريف
unit	unit structure بنية وحدة
until	statement تعليمة
uses	unit structure بنية وحدة
var	declaration تعريف
virtual	directive (method) توجيه (مسار)
while	statement تعليمة
with	statement تعليمة
write	property specifier معيّن سمة
writeonly	dispatch interface specifier معيّن واجهة ارسال
xor	operator (boolean) عامل (بولي)

التعبيرات والعامِلات

لا توجد قاعدة عامة لبناء التعبيرات expressions ، حيث تعتمد اساسا على العامِلات التي تستخدم، و التي لباسكال العديد منها. هناك المنطقي logical والحسابي arithmetic والبولي Booleanوالعلائقي relational ، و عاملات الفئةset ، بالاضافة الى عدد آخر. يمكن استعمال التعبيرات لتحديد القيمة التي ستخصص للمتغير، او لحساب المحدد parameter التابع لوظيفة او اجراء، او لاختبار شرط. و قد تتضمن التعبيرات استدعاء وظائف ايضا. في كل مرة تقوم فيها باجراء عملية على قيمة في معرّف، و ليس استعمال المعرف في حد ذاته، فان هذا يعدّ تعبيرا .

تعد التعبيرات امرا شائعا في لغات البرمجة. التعبير هو أي توليفة من الثوابت constants ، المتغيرات، القيم الحرفية literal ، عاملات، و نتائج الوظائف .التعبيرات يمكن ايضا تمريرها الى المحددات القيمية value parameters في الاجراءات و الوظائف، و لكن ليس دائما الى المحددات المرجعية reference parameters (التي تحتاج الى قيمة يمكن تخصيصها) .

العامِلات و أسبقيتها

إذا سبق لك و أن كتبت برنامجا في حياتك، فانك تعلم بالفعل ماذا تعني كلمة تعبير expression. هنا سوف ألقي الضوء على عناصر محددة في عاملات باسكال. يمكنك رؤية قائمة بعاملات اللغة، مجمعة حسب الأسبقية، في الجدول 2.1 .

على العكس من معظم اللغات الأخرى، فإن عاملات and و or لهما الأسبقية على العاملات العلائقية. لذاك اذا كتبت *a < b and c < d* ، فان المجمّع سيحاول تنفيذ عملية and أولا، منتجا بذلك خطأ تجميع. لهذا السبب عليك وضع كل من تعبير < بين قوسين : *(a < b) and (c < d)*.

بعض العاملات الشائعة لديها معان مختلفة مع انواع بيانات مختلفة. مثال ذلك، العامل + يمكن استخدامه لجمع رقمين، لوصل جملتين، صنع اتحاد بين فئتين، او حتى جمع رصيف offset مع مؤشر . Pchar الا انك لاتستطيع جمع حرفين، كما هو ممكن في لغة .c

عامل آخر غريب وهو . div ففي باسكال، يمكنك تقسيم أي رقمين (حقيقي أو صحيح (بواسطة العامل / ، وسوف تحصل بصورة ثابتة على رقم حقيقي كناتج. اما اذا احتجت الى تقسيم رقمين صحيحين للحصول على ناتج صحيح، استخدم العامل div كبديل .

الجدول 2.2: معاملات لغة باسكال، مجمعة حسب اسبقيتها

عاملات أحدادية (أسبقية عليا)	
@	عنوان المتغير أو الوظيفة (ترجّع مؤشّر)
not	بولي أو ما يخص الجزيئات
العاملات الضربية و ما يخص الجزئيات	
*	ضرب حسابي أو تقاطع فئة
/	تقسيم نقطة عائمة
div	تقسيم عدد صحيح
mod	البواقي (باقي تقسيم عدد صحيح)
as	تلبيس نوع آمن (RTTI)
and	بولي أو ما يخص الجزيئات
shl	ازاحة لليسار فيما يخصّ الجزئيات
shr	ازاحة لليمين فيما يخصّ الجزئيات
العاملات الجمعية	
+	جمع حسابي، اتحاد فئة، ربط جمل، اضافة رصيف مؤشّر
-	طرح حسابي، طرح وتخالف فئة، طرح صف مؤشّر
or	بولي أو ما يخص الجزيئات
xor	بولي أو ما يخص الجزيئات
عاملات العلاقة والمقارنة (أسبقية دنيا)	
=	اختبار مساواة
<>	اختبار عدم مساواة
<	اختبار أقل من
>	اختبار اكبر من
=<	اختبار أقل من أو يساوي، أو فئة فرعية من فئة
>=	اختبار أكبر من أو يساوي، أو فئة عليا تعلو فئة
in	اختبار اذا ما عنصر عضو في فئة
is	اختبار توافقية نوع لكائن (عامل RTTI آخر)

عامِلات الفئة

عـاملات الفئـة تتضـمن اتحـاد(+) union ، طـرح(-) difference ، تقـاطع(*) intersection ، اختبار عضـوية(in) membership ، بالإضـافة الـى مجموعـة من العـاملات العلائقيـة. لاضـافة عنصر لمجموعة، يمكنك جعل اتحاد فئة مع آخرى تملك فقط العنصر الذي تحتاجه. فيما يلي مثال بدلفي له علاقة بنمط الخطّ font styles:

```
Style := Style + [fsBold];
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

كبديل يمكنك استخدام الاجرائين الاعتيادين Include وExclude ، وهما أكثر فاعلية(لكنهما لايمكن استعمالهما مع سمات مكوّن التي تكون من نوع set ، لأنها تحتاج الى محدد ذو قيمة واحدة) :

```
Include (Style, fsBold);
```

ملخّص

الآن وقد عرفنا الخطوط الأساسية لبرنامج باسكال فنحن جاهزون لفهم معانيها بالتفصيل. سوف نبدأ باستكشاف تعريف أنواع البيانات سابقة التحديد و المحددة بالمستعمل، بعدها سننتقل معها الى استخدام الكلمات المفتاحية لبناء تعليمات برمجية .

الفصل التالي :الأنواع، المتغيرات، و الثوابت

الفصل 3
الأنواع، المتغيرات، والثوابت

اعتمدت لغة باسكال الأصلية عل بعض المفاهيم البسيطة، والتي اصبحت الآن عامة في لغات البرمجة. المفهوم الأول هو نوع البيانات .data type النوع يحدد القيم التي يمكن للمتغيرات ان تتخذها، والعمليات التي يمكن انجازها عليها. ان مفهوم النوع أقوى في باسكال مقارنة بلغة س، حيث انواع البيانات الحسابية غالبا ما تكون متبدلة، وهي أقوى بكثير من النسخ الأصلية للغة بيسك، حيث لاتملك مثل هذا المفهوم .

المتغيرات

تتطلّب باسكال ان تكون كل المتغيرات معرّفة قبل استخدامها. وحتى في الوقت الذي تعرّف فيه المتغير، يجب أن تحدّد نوع البيانات. ها هنا بعض نماذج تعريف المتغيرات :

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

المصطلح *var* يمكن استخدامه في اماكن مختلفة في الكود، كأن يكون في بداية توليف وظيفة أو اجراء، او ان يتم تعريف المتغيرات محليا local في الروتين ، أو داخل الوحدة لتعريف متغيرات جامعة .global بعد مصطلح *var* تأتي قائمة اسماء المتغيرات، متبوعة بشارحة واسم نوع البيانات. يمكنك كتابة اكثر من اسم متغير واحد في السطر الواحد، كما هو في آخر تعليمة أعلاه .

حالما تقوم بتحديد متغير من نوع ما، تستطيع ان تقوم فقط بمباشرة العمليات الداعمة لنوع بياناته. على سبيل المثال، يمكنك استعمال القيمة البولية للاختبار و القيمة الصحيحة في التعبير الرقمي. لايمكنك مزج القيم البولية والصحيحة (كما هو الأمر مع لغة س).

باستخدام تخصيصات بسيطة، نستطيع كتابة التوليف التالي :

```
Value := 10;
IsCorrect := True;
```

لكن التعليمة التالية ليست صحيحة، لأن المتغيرين يملكان نوع بيانات مختلف :

```
Value := IsCorrect; // error
```

اذا حاولت تحويل هذا التوليف، فان دلفي تقوم باصدار خطأ تحويل رفق هذا التوضيح: *Incompatible types: 'Integer' and 'Boolean'* . عادة، مثل هذه الأخطاء هي أخطاء برمجية، لأنه لامعنى لتخصيص قيمة True أو False لقيم من نوع بيانات صحيح. يجب أن لا تلوم دلفي على مثل هذه الأخطاء. هي فقط تنبهك لوجود خطأ ما في التوليف .

بالطبع، غالبا مايمكن تبديل قيمة متغير من نوع بيانات الى نوع مختلف. في بعض الحالات، هذا التبديل يكون آليا، لكن عادة ما تحتاج الى استدعاء وظائف محددة في النظام لتغيير التمثيل الداخلي للبيانات .

تستطيع في دلفي ان تخصص قيمة تمهيدية لمتغير جامع global variable أثناء تعريفك له. مثلا، تستطيع كتابة :

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

تقنية التمهيد initialization هذه تصلح فقط للمتغيرات الجامعة، وليس للمتغيرات المعرفة داخل نطاق اجراء أو مسار .

الثوابت

تسمح باسكال ايضا بتعريف ثوابتا constants لتسمية القيم التي لاتتغير خلال عمل البرنامج. لتعريف ثابت لاتحتاج لتحديد نوع البيانات، فقط تخصيص قيمة ابتدائية .المحوّل سيتفحّص القيمة وآليا يستخدم نوع بياناته المناسب. ها هنا بعض امثلة التعريفات :

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantù';
```

يقرر دلفي نوع البيانات للثابت بناء على قيمته. في المثال أعلاه، الثابت Thousand يفترض ان يكون من نوع صحيح صغير SmallInt ، اصغر نوع صحيح يمكنه احتواء القيمة. اذا اردت الطلب من دلفي استخدام نوع محدد؛ يمكنك ببساطة اضافة اسم النوع في التعريف، كما هو في :

```
const
  Thousand: Integer = 1000;
```

عندما تقوم بتعريف ثابت، يستطيع المحوّل أن يختار بين أن يخصص موقعا في الذاكرة للثابت، و يحفظ فيه قيمته، أو أن ينسخ قيمته الحقيقية في كلّ مرة يتم فيها استعمال الثابت. الأسلوب الثاني يبدو معقولا خاصة بالنسبة للثوابت البسيطة .

ملاحظة: نسخ 16-بت من دلفي تسمح لك بتغيير قيمة الثابت محدد النوع في زمن التشغيل، كما لو كان متغيرا. نسخة 32-بت لازالت تسمح بهذا السلوك من اجل التوافقية مع السابق وذلك عندما تقوم بتمكين موجّه المحوّل *$J* ، او بالتعليم على مؤشر *Assignable typed constants* في صفحة المحوّل في نافذة خيارات المشروع . Project Options وبالرغم من أن هذا هو التوصيف الافتراضي، فانه ينصح بقوة كفنيّات البرمجة أن لاتستعمل هذه الخدعة. ان تخصيص قيمة جديدة لثابت يمنع كل تشذيبات المحول على الثواب. واذا اضطررت لهذا، ببساطة قم بتعريف متغيرات، كبديل .

ثوابت الجمل الموردية

عندما تحدد ثابت جملة، فبدلا من كتابة :

```
const
  AuthorName = 'Marco Cantù';
```

يمكنك بدءاً من دلفي 3 أن تكتب التالي :

```
resourcestring
  AuthorName = 'Marco Cantù';
```

في كلتا الحالتين انت تحدد ثابتا؛ قيمة لا تقم بتغييرها خلال زمن التشغيل. الفرق فقط في كيفية الانجاز. ثابت الجملة المحدد بواسطة الموجّه *resourcestring* يخزّن ضمن موارد البرنامج resources، في جدول للجمل .

لكي ترى هذه الامكانية فعليا، قم بالاطلاع على مثال ResStr ، والذي له زرّا رفق التوليف التالي :

```
resourcestring
  AuthorName = 'Marco Cantù';
  BookName = 'Essential Pascal';

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage (BookName + #13 + AuthorName);
end;
```

ناتج الجملتين يظهر في سطرين منفصلين لأن الجملتين مفصولتين بالحرف الدّال لسطر جديد *newline*(مشار اليه بقيمته الرقمية في *#13* وهو ثابت نوع حرف).

الجانب المثير في هذا البرنامج هو انك اذا تفحصّته بمستكشف للموارد (resource explorer هناك واحد متوفر في الأمثلة التي تأتي مع دلفي) سوف ترى الجمل الجديدة ضمن الموارد. هذا يعني بأن الجمل ليست جزءا من التوليف المحول ولكنها خزّنت في منطقة منفصلة في الملف التنفيذي (ملف EXE).

> **ملاحظة:** باختصار، مزايا الموارد هي في الكفاءة في مناولة الذاكرة التي تقوم بها ويندوز، وفي إمكانية توطينlocalizing البرنامج (ترجمة الجمل إلى لغات مختلفة) بدون الحاجة إلى تعديل توليفها المصدري source code.

أنواع البيانات

في باسكال توجد عدة أنواع بيانات سابقة التحديد predefined data types ، والتي يمكن تقسيمها الى ثلاث مجموعات: الأنواع التراتبية *ordinal types* ، الأنواع الحقيقية *real types* و الجمل *strings*. سوف نناقش الانواع التراتبية في الأقسام التالية، بينما يتم تغطية الجمل لاحقا في هذا الفصل. في هذا القسم سوف أقدم ايضا بعض الأنواع المحددة من قبل مكتبات دلفي (ليست محددة من قبل المحوّل) ، والتي يمكن اعتبارها انواع سابقة التحديد predefined types.

تتضمن دلفي ايضا نوع بيانات بدون نوع *non-typed* ، تسمّى متباين *variant* ، سيتم مناقشتها في الفصل العاشر من هذا الكتاب. غريب جدا ان يكون المتباين نوعا بدون ما يناسبه من تحقق من النوع. لقد تم ادخال هذا النوع في دلفي 2 لمناولة آليات او ال اي OLE Automation .

الأنواع التراتبية

الأنواع التراتبية Ordinal types مبنية على مفهم الترتيب أو التوالي و التتابع .ليس بامكانك فقط مقارنة قيمتين لمعرفة أيهما الأكبر، ولكن يمكنك أيضا معرفة القيمة التي تلي او تسبق قيمة أخرى، أو تقوم بحساب أدني أو أعلى قيمة محتملة .

أكثر ثلاث أنواع تراتبية سابقة التحديد هي الصحيحInteger ، البوليBoolean ، و الحرف Char.عموما، هناك عددا من الأنواع الأخرى ذات العلاقة والتي لها معنى مشابها ولكن لها تمثيلا ومدى قيم مختلفين. جدول 3.1 التالي يعرض انواع البيانات التراتبية المستخدمة لتمثيل الأرقام .

جدول 3.1: أنواع البيانات التراتبية للأرقام

Size الحجم	Signed معلّم Range المدى	Unsigned غير معلّم Range المدى
8 bits	ShortInt -128 to 127	Byte 0 to 255
16 bits	SmallInt -32768 to 32767	Word 0 to 65,535
32 bits	LongInt -2,147,483,648 to 2,147,483,647	LongWord (بدءاً من دلفي 4) 0 to 4,294,967,295
64 bits	Int64	
16/32 bits	Integer	Cardinal

كما ترى، هذه الأنواع لها علاقة بالتمثيلات المختلفة للأرقام، حسب عدد الجزئيات bits المستخدمة للتعبير عن القيمة، وحسب وجود أو غياب جزئية العلامة. القيم المعلّمة signed valuesيمكنها أن تكون موجبة أو سالبة، لكن لها مدى أصغر من القيم، وذلك لأن المتاح من الجزئيات للقيمة نفسها قد نقصت بواحدة. تستطيع أن ترجع الى مثال المدى الذي سيناقش في القسم التالي، من أجل معرفة المدى الفعلي لقيم كل نوع .

المجموعة الأخيرة (والمشارة ب 32/16) تشير الى القيم التي لها تمثيلا مختلفا في نسخ 16-بت و 32-بت من دلفي. الصحيح Integer و الرئيسي Cardinal يستعملان بكثرة، لأنهما يطابقان التمثيل الفطري للأرقام في المعالج الحسابي CPU .

الأنواع الصحيحة في دلفي 4

في دلفي 3، الأرقام ذات 32 بت غير المعلّمة والمشار اليها بنوع رئيسي cardinal كانت سابقا قيم 31 بت، بمدي يبلغ حتى 2 قيقابايت. قدمت دلفي 4 نوع جديد رقمي غير معلّم، LongWord، والذي يستخدم فعليا قيمة 32 بت تبلغ حتى 4 قيقابايت. وأصبح نوع كاردينال الآن اسما مرادفا لنوع لونغوورد الجديد. لونغوورد يسمح ب 2 GBاكثر اضافية من البيانات يمكن عنونتها من قبل رقم غير معلّم، كما أشير اليه سابقا. أكثر من هذا، فإنه يماشي التمثيل الفطري للأرقام في المعالج الحسابي .

نوع آخر جديد تم ادخاله في دلفي 4 و نوعint64 ، والذي يمثل أعدادا صحيحة تبلغ 18 رقم. هذا النوع الجديد مدعوم بالكامل من قبل بعض اجرائيات routines الأنواع التراتبية (مثلHigh و(Low ، والإجرائيات الرقمية (مثل Inc و(Dec ، وإجرائيات تبديل الجمل-string conversion) مثل .(IntToStr ومن أجل التبديل العكسي، من جملة ألى رقم، هناك وظيفتين جديدتين :*StrToInt64* و *StrToInt64Def* .

البولي

القيم البولية غير النوع البولي نادرة الاستعمال. بعض القيم البولية لديها تمثيل خاص وذلك لمتطلبات وظائف ويندوز .Windows API functions الأنواع هيByteBool، WordBool و LongBool.

في دلفي 3 ومن أجل التوافق مع فيجوال بيسك و آلياتOLE ، فان انواع البياناتـ*ByteBool* ، *WordBool*، و *LongBool* تم تعديلهم لتمثيل القيم *True* بـ -1، بينما القيمة *False* لازالت 0. نوع البيانات *Boolean* ظلّت كما هي (*True*هي 1، *False* هي 0). إذا قمت باستعمال سبك للنوع typecast صريح في برنامج بدلفي 2، فإن نقل البرنامج الى نُسخ لاحقة من دلفي قد ينتج عنه بعض الأخطاء .

الأحرف

أخيرا هناك تمثيلين مختلفين للأحرف *ANSIChar* :و .*WideChar* النوع الأول يمثل أحرفا ذات جزئيات ثمان bit8-، تتماشى مع مجموعة أحرف انسي ANSI والمستخدمة تقليديا من قبل ويندوز؛ التمثيل الثاني الأحرف 16-جزئية ، وتتماشى مع أحرف يونيكود الجديدة Unicode والمدعومة بالكامل من قبل ويندوز ن ت، وجزئيا من قبل ويندوز 95 و 98. معظم الوقت سوف تستعمل ببساطة نوع حرف *Char* ، والذي في دلفي 3 تطابق *ANSIChar* . ليكن معلوما، على أي حال، ان أول 256 من حروف يونيكود توافق تماما حروف انسي ANSI .

أحرف الثوابت يمكن تمثيلها بمجموعة رموزها، كما في*'k'* ، او بمجموعة أرقامها، كما في*#78* . الأخيرة يمكن التعبير عنها ايضا ياستخدام الوظيفية *Chr* ، كما في *Chr (78)* . التبديل المعاكس يمكن اجراؤه بواسطة الوظيفة *Ord* .

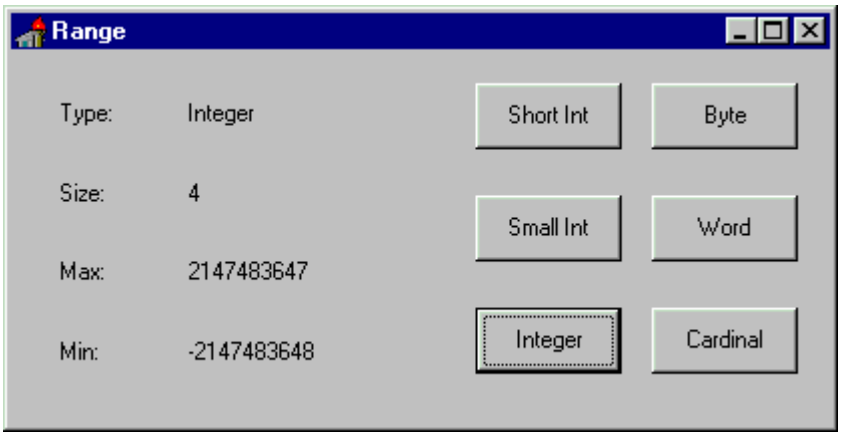
بصفة عامة يكون من الأحسن استخدام مجموعة الرموز عند الإشارة الى الأحرف، والأرقام، والعلامات. عند الإشارة الى أحرف خاصة، ستستخدم مجموعة الأرقام عموما بدلا من ذلك. القائمة التالية تتضمن بعض أكثر الأحرف الخاصة استعمالا :

- tabulator *#9* جدولة
- newline *#10* سطر جديد
- carriage return (enter key) *#13* مفتاح الإدخال

مثال Range

لإعطائك فكرة عن الاختلاف من مدى لآخر في بعض الأنواع التراتبية، قمت بكتابة برنامج دلفي بسيط أسميته .Range بعض النتائج تظهر في الشكل 3.1 .

الشكل 3.1: مثال Range يظهر بعض المعلومات حول انواع البيانات التراتبية (الأرقام الصحيحة في هذه الحالة).



برنامج Range مبني على نموذج form بسيط، به ستة أزرار) buttons كلّ مسمّاة حسب نوع البيانات التراتبية) وبعض الملصقات labels لمختلف المعلومات، كما هو مبيّن في الصورة 3.1. استخدمت بعض الملصقات لتحوي نصّا ثابتا، الأخرى لعرض المعلومات عن النوع في كلّ مرّة يُضغط فيها على زرّ .

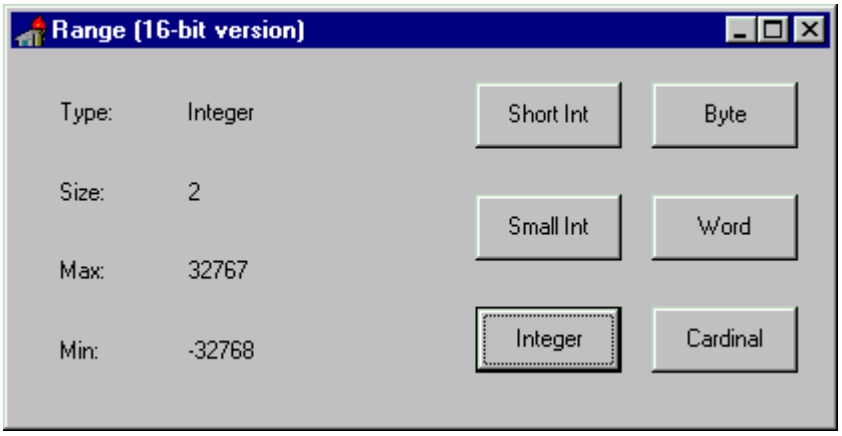
في كلّ مرّة تضغط فيها على الأزرار، يقوم البرنامج بتحديث الملصقات بحسب الناتج .ملصقات مختلفة تعرض نوع البيانات، عدد البايت المستعملة، و أقصى وأدني قيمة يمكن للنوع أن يخزّنها. كل زرّ يملك مايخصّه من مسار method تجاوب الحدث *OnClick* لأن التوليف المستخدم لحساب القيم الثلاث يختلف قليلا من زرّ لآخر. مثلا، هاهنا التوليف المصدري لحدث *OnClick*للزرّ الخاص بنوع الصحيح (*BtInteger*) :

```
procedure TFormRange.BtnIntegerClick(Sender: TObject);
begin
  LabelType.Caption := 'Integer';
  LabelSize.Caption := IntToStr (SizeOf (Integer));
  LabelMax.Caption := IntToStr (High (Integer));
  LabelMin.Caption := IntToStr (Low (Integer));
end;
```

إذا كنت تملك بعض الخبرة في البرمجة بدلفي، يمكنك تفحّص التوليف لفهم كيفية عمله. بالنسبة للمبتدئين، يكفي ملاحظة الإستخدام للوظائف الثلاثة *SizeOf* :و *High*و *Low*. نواتج الوظيفتين الأخيرتين هما تراتبيات من نفس النوع (في هذا الحالة، أعداد صحيحة)، وناتج وظيفة *SizeOf* هي دائما عدد صحيح .القيمة المرتجعة من هذه الوظائف الثلاث تترجم أولا إلى جُمل بإستخدام وظيفة*IntToStr* ، ثم تُنسخ في لافتات captions الملصقات الثلاث .

المسارات المرتبطة بالأزرار الأخرى تشبه كثيرا المسار المذكور أعلاه. الإختلاف الفعلي الوحيد هو نوع البيانات الذي تم تمريره كمحدد لمختلف الوظائف. الصورة 3.2 تعرض ناتج تنفيذ البرنامج تحت ويندوز 95 بعد أن تم إعادة تحويله recompile بواسطة نسخة 16-بت من دلفي. بمقارنة الشكل 3.1 و الشكل 3.2، يمكنك رؤية الإختلاف بين نوع البيانات الصحيح ذو 16-بت وذلك ذو 32-بت .

الشكل 3.2: ناتج نسخة 16-بت من مثالRange ، يعرض مرّة أخرى معلومات عن العدد الصحيح .



حجم نوع الصحيح *Integer* يتباين بحسب المعالج الحسابي ونظام التشغيل المستخدم. في ويندوز 16-بت، المتغير الصحيح يسع 2 بايت. بينما في ويندوز 32، سعة الصحيح 4 بايت. لهذا السبب، عندما تعيد تحويل مثالRange ، ستحصل على نتائج مختلفة .

التمثيلين المختلفين لنوع الصحيح ليست بمشكلة، طالما أن برنامجك لايضع أية إفتراضات عن حجم الصحيح. إذا ما حدث وقمت بحفظ عدد صحيح في ملف بإستخدام نسخة ثم حاولت استرجاعه بنسخة أخرى، فسوف تواجه بعض المتاعب. في هذه الحالة، يجب أن تختار نوع بيانات مستقل عن بيئة التشغيل (مثل *LongInt* أو *SmallInt*). لأغراض الحسابات الرياضية أو توليف عام، أفضل مراهنة لديك هو أن تلتزم تمثيل الصحيح النمطي لبيئة التشغيل المعينة-- هذا يعني، ان تستخدم نوع الصحيح-- لأنه المفضّل لدى المعالج الحسابي. نوع الصحيح *Integer* يجب أن يكون خيارك الأول عندما تعالج أعدادا صحيحة. و لا تستخدم تمثيلا مختلفا إلا إذا وجدت سببا قاهرا لذلك .

إجرائات الأنواع التراتبية

بعض إجرائيات النظام system (إجرائيات محددة في لغة باسكال وفي وحدة النظام في دلفي system unit) يمكنها تتعامل مع الأنواع التراتبية ordinal types. هي معروضة في الجدول 3.2. المبرمجون بلغة س++ سوف يلاحظون بأن النسختين من إجرائية*Inc* ، مع محدد أو اثنين، يطابقان معاملات ++ و =+ (نفس الأمر مع إجرائية *Dec*).

الجدول 3.2: إجرائيات نظام للأنواع التراتبية

الإجرائية	الغرض
Dec	تخفيض decrease المتغير الذي يتم تمريره كمحدد، بمقدار واحد أو بمقدار قيمة المحدد الثاني الاختياري.
Inc	زيادة increase المتغير الذي يتم تمريره كمحدد، بمقدار واحد أو بمقدار القيمة المعطاة.
Odd	يرجع إثبات إذا كانت القيمة المعطاة عددا فرديا odd.
Pred	يرجع القيمة التي تسبق تلك المعطاة بحسب الترتيب المقرر في نوع البيانات، السابق predecessor.
Succ	يرجع القيمة التي تلي تلك المعطاة، التالي successor.
Ord	يرجع رقما يدل على ترتيب order القيمة المعطاة ضمن مجموعة القيم في نوع البيانات.
Low	يرجع أدنى low قيمة ضمن مدى النوع التراتبي المعطى كمحدد.
High	يرجع أعلى high قيمة ضمن مدى نوع البيانات التراتبي.

لاحظ ان بعض هذه الإجرائيات، عندما يتم تطبيقها على الثوابت constants ، فإن المحوّل يقوم بتقييمها آليا واستبدالها بقيمتها. مثلا اذا قمت باستدعاء *High(X)* حيث *X* معرّفة كصحيح، فالمحوّل يستطيع ببساطة تبديل التعبير بآخر يمثل أعلى قيمة محتملة لنوع بيانات الصحيح .

الأنواع الحقيقية

تقوم الأنواع الحقيقية بتمثيل أرقام النقطة العائمة بعدة أشكال. أصغر حجم تخزين تمثلها الأرقام الوحيدة *Single* ، والتي تنفذ بقيمة ذات 4-بايت. ثم هناك الأرقام النقطة العائمة المضاعفة *Double*، المنفذة بعدد 8 بايت، والأرقام الممتدّة *Extended* ، والمنجزة بعدد 10 بايت. كل هذه أنواع بيانات نقطة عائمة مع اختلاف في الضبط والدقة precision.

في دلفي 2 ودلفي 3 نوع الحقيقي Real له نفس التعريف الذي في نسخة 16-بت؛ لقد كانت بنوع 48-بت. لكن تم تخفيض استخدامه من قبل بورلاند، والتي اقترحت بأن تقوم باستعمال أنواع الوحيد والمضاعف والممتد بدلا منه. سبب اقتراحهم هذا هو ان الشكل القديم ذو 6-بايت ليس مدعوما من قبل معالجات انتل Intl كما انه ليس معروضا ضمن القائمة الرسمية للأنواع الحقيقية والتي أصدرتها IEEE. ولكي يتم تلافي المشكلة تماما، قامت دلفي 4 بتعديل التعريف الخاص بنوع الحقيقي حتى يمثل الشكل القياسي لرقم عائم النقطة ذو 8 بايت (64-بت).

بالإضافة إلى ميزة استخدام تعريفا قياسيا متفق عليه، هذا التغيير يسمح للمكونات components باصدار سمات properties مبنية عل نوع حقيقي، الشيء الذي لم يكن دلفي 3 يسمح به. أما العيوب فقد تبرز مشاكل التوافقية. في حالة الضرورة، وعند الاصرار على طريقة دلفي 2 و 3 في تعريف النوع، يمكنك تجاوز احتمال انعدام التوافقية؛ وذلك باستخدام خيار المجمّع التالي :

```
{$REALCOMPATIBILITY ON}
```

هناك أيضا نوعان غريبان من أنواع البيانات: *Comp* و يصف رقم صحيح كبير جدا باستخدام 8 بايت (والذي يمكنه احتواء أرقام ذات 18 خانة عشرية)؛ و *Currency* عملة (ليست متوفرة في دلفي 16-بت) وهي تشير الى قيمة بنقطة عشرية ثابتة مع أربع خانات عشرية، و بنفس تمثيل 64-بت كما في نوع *Comp* . كما يوحي الاسم، نوع بيانات عملة *Currency* أضيف لمناولة القيم النقدية شديدة الدقة، مع أربع خانات عشرية .

لا نستطيع بناء برنامج يشبه مثال Range بتطبيق أنواع بيانات حقيقية، لأننا لا يمكننا استخدام وظائف *High* و *Low* أو *Ord* مع متغيرات نوع حقيقي. الأنواع الحقيقية تمثل (نظريا) مجموعة لانهائية من الأرقام، بينما الأنواع التراتبية تمثل مجموعة ثابتة من القيم .

ملاحظة :دعني أشرح ذلك بطريقة أفضل. عندما يكون لديك الرقم الصحيح 23، يمكنك أن تقرّر ما هي القيمة التي تليه. الأرقام الصحيحة نهائية (لديها مدى محدد ولديها ترتيب). الأرقام عائمة النقطة هي غير نهائية حتى ضمن المدى القصير، وليس لديها ترتيب: في الواقع، كم توجد قيمة بين 23 و 24 ؟ و ما هو الرقم الذي يلي 23.46 ؟ هل هو 23.47، 23.461، أو 34.4601 ؟ هذا الأمر يصعب معرفته بالفعل .

لهذا السب، يبدو الأمر معقولا حين نسأل عن ترتيب موضع حرف *w* ضمن مدى نوع بيانات حرفchar ، و لكن ليس من المعقول ابدا أن نسأل نفس السؤال عن الرقم 7143.1562 ضمن مدى نوع بيانات النقطة العائمة. بالرغم انه بالتأكيد تستطيع معرفة ما إذا كان; رقم حقيقي ما لديه قيمه أعلى من قيمة رقم آخر، فإنه من غير المنطقي أن نسأل عن عدد الأرقام الحقيقية الموجودة قبل رقم ما (هذا معني وظيفة *Ord*).

الأنواع الحقيقية لديها دورا محدودا في ذلك الجزء من التوليف البرمجي الخاص بواجهة المستخدم user interface (الجانب الخاص بويندوز)، لكنها مدعومة بالكامل من قبل دلفي، بما في ذلك جانب قواعد البيانات. ان دعم مواصفات IEEE القياسية لأنواع النقطة العائمة تجعل من لغة اوبجكت باسكال مناسبة تماما لنطاق واسع من البرامج التي تتطلب حسابات رقمية. إذا كنت مهتما بهذا الجانب، يمكنك إلقاء نظرة على الوظائف الرياضية المقدمة من دلفي وذلك في ملف وحدة system (انظر Delphi Help من أجل تفاصيل أكثر).

ملاحظة : لدلفي أيضا ملف وحدة *Math* التي تحدد إجرائيات رياضية أكثر تقدما، تغطي وظائف حساب المثلثات (مثل وظيفة *ArcCosh*) ، مالية (مثل وظيفة *InterestPayment*) ، و إحصائية (مثل إجرائية *MeanAndStdDev*). هناك العديد من هذه الإجرائيات، بعضها تبدو غريبة بالفعل بالنسبة لي، مثل إجرائية (*MomentSkewKurtosis* سأدع هذا الأمر لك لمعرفته).

التاريخ والوقت

تستخدم دلفي أيضا أنواعا حقيقية real types لمناولة معلومات التاريخ والوقت .وليكون الأمر أكثر دقّة حدّدت دلفي نوع بيانات *TDateTime*. وهو نوع نقطة عائمة، لأن النوع يجب أن يكون واسعا بما يكفي لإحتواء السنوات، الأشهر، الأيام، الساعات، الدقائق، والثواني، نزولا إلى دقة تبلغ جزء من ألف من الثانية، كلّ ذلك في متغير واحد. التواريخ تخزّن كإجمالي عدد الأيام منذ 1899/30/12 (مع قيم سالبة تشير إلى التواريخ ما قبل 1899) في الجزء الصحيح integerمن قيمة *TDateTime*.

نوع *TDateTime* ليس نوعا مسبق التحديد بحيث يفهمه المحوّل، لكن قد تمّ تعريفه في ملف وحدة system كالتالي:

```
type
  TDateTime = type Double;
```

إستخدام *TDateTime* يعدّ بسيط، لأن دلفي تحوي عددا من الوظائف التي تتعامل مع هذا النوع. يمكنك أن تجد قائمة بهذه الوظائف في الجول 3.3 .

الجدول 3.3: إجرائيات نظام لنوع TDateTime

الإجرائية	البيان
Now	استرجاع التاريخ والوقت الحالي في قيمة على هيئة.TDateTime
Date	استرجاع فقط التاريخ الحالي.
Time	استرجاع فقط الوقت الحالي.
DateTimeToStr	تحويل قيمة التاريخ والوقت الى جملة، باستخدام المعلومات المبدئية؛ من أجل تحكم أكثر في التحويل استخدم وظيفة.FormatDateTime
DateTimeToString	نسخ قيم التاريخ والوقت في حيّز جملة string buffer وفق المعلومات الابتدائية.
DateToStr	تحويل جانب التاريخ في قيمة TDateTime الى جملة.
TimeToStr	تحويل جانب الوقت في قيمة TDateTime الى جملة.
FormatDateTime	صياغة التاريخ والوقت باستخدام صيغة محدّدة، تستطيع ان تحدد أية قيمة تريد عرضها وأية صيغة تستعمل، منتجة صياغة غنية للجملة.
StrToDateTime	تحويل جملة تحوي معلومات التاريخ والوقت الى قيمةTdateTime ، مظهرة رفضا exception في حالة وجود خطأ في صيغة الجملة.
StrToDate	تحويل جملة تحوي قيمة تاريخ الى صيغة.TDateTime
StrToTime	تحويل جملة تحوي قيمة وقت الى صيغة.TDateTime
DayOfWeek	يسترجع رقم ترتيب اليوم في الأسبوع حسب قيمة TDateTime المعطاة.
DecodeDate	استرجاع قيم السنة، الشهر، و اليوم من قيمة تاريخ.
DecodeTime	استرجاع قيمة الوقت.

EncodeDate	تحويل قيم السنة، الشهر، و اليوم إلى قيمة TDateTime.
EncodeTime	تحويل قيم الساعة، الدقيقة، الثانية، وأجزاء الثانية إلى قيمة TDateTime.

من أجل أن ترى كيف يتم استعمال نوع البيانات هذا و بعض الإجرائيات الخاصة به، قمت ببناء برنامج بسيط، أسميته TimeNow. النموذج الرئيسي في هذا المثال به زرّ Button و قائمة ListBox. عندما يبدأ البرنامج يقوم آليا بحساب وعرض الوقت والتاريخ الحالي. في كلّ مرّة يتم فيها الضغط على الزرّ، يقوم البرنامج بعرض الزمن المنقضي منذ بدء البرنامج .

فيما يلي التوليف الموصول بحدث OnCreate الخاص بالنموذج :

```
procedure TFormTimeNow.FormCreate(Sender: TObject);
begin
  StartTime := Now;
  ListBox1.Items.Add (TimeToStr (StartTime));
  ListBox1.Items.Add (DateToStr (StartTime));
  ListBox1.Items.Add ('Press button for elapsed time');
end;
```

التعليمة الأولى تنادي وظيفة *Now* ، التي تسترجع التاريخ و الوقت الحاليين. القيمة المسترجعة تُخزّن في متغيّر *StartTime* ، والذي تم تعريفه كمتغير خارجي global كالتالي :

```
var
  FormTimeNow: TFormTimeNow;
  StartTime: TDateTime;
```

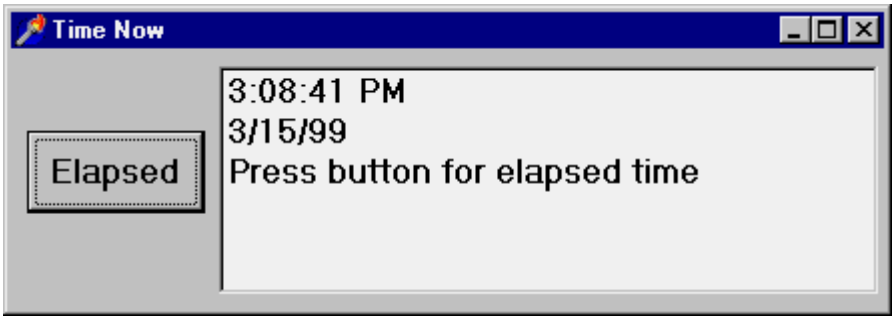
لقد أضفت فقط التعريف الثاني، حيث أن دلفي تقوم بتوفير الأول. حيث يكون مبدئيا كالتالي :

```
var
  Form1: TForm1;
```

عند تغير اسم النموذجform ، يتم تحديث هذا التصريح آليا. ان استخدام متغيرات جامعةglobal لا يعدّ حقيقة أفضل الطرق: سيكون من الأفضل لو تم استخدام حقل خاص private field تابع لطبقة النموذجform class ، موضوع له علاقة ببرمجة المنحى الكائني object-oriented programmingتم مناقشته في كتاب التحكم بدلفي 4 . Mastering Delphi

التعليمات الثلاث التالية تضيف ثلاثة عناصر الى مكوّن القائمة يمين النموذج، مع النتائج التي تراها في الشكل 3.3. السطر الأول يحتوي على الجزء الخاص بالوقت في قيمة *TDateTime* محوّلا الى جملة، الثاني يحتوي جزء التاريخ من نفس القيمة .في نهاية التوليف أضيف تذكير بسيط.

الشكل 3.3: ناتج مثال TimeNow عند البدء .



الجملة الثالثة يستبدلها البرنامج عندما يضغط المستعمل على زرّ" Elapsed المنقضى :"

```
procedure TFormTimeNow.ButtonElapsedClick(Sender: TObject);
var
  StopTime: TDateTime;
begin
  StopTime := Now;
  ListBox1.Items [2] :=  FormatDateTime ('hh:nn:ss',
    StopTime - StartTime);
end;
```

التوليف يسترجع الوقت الجديد ويحسب الفرق بينه وبين قيمة الوقت المخزّن عند ابتداء البرنامج. ولأننا نحتاج الى استخدام القيمة التي سبق حسابها في سياق حدث مختلق، كان علينا أن نخزّنها في متغير جامع. في الواقع توجد بدائل أفضل، مبنية على مفهوم الطبقات .classes

ملاحظة :التوليف الذي يقوم باستبدال القيمة الحالية في الجملة الثالثة يستعمل دليل .2 index السبب في ذلك ان بنود القائمة مبنية على الصفر :are zero-based البند الأول رقمه 0، الثاني رقم 1، و الثالث رقم 3. سنرى المزيد من ذلك عندما نناقش المصفوفات .

بجانب استدعاء *DateToStr* و *TimeToStr* يمكنك استعمال وظيفة *FormatDateTime* الأكثر قوة، كما فعلت في المسار method الأخير أعلاه (انظر ملف مساعدة دلفي من أجل تفاصيل محددات الصياغة). لاحظ أيضا أن قيم الوقت والتاريخ تُبدّلان إلى جُمل بحسب توصيف الدوليّات international في ويندوز. دلفي يقرأ هذه القيم من النظام، و يوزعها في عدد من الثوابت الجامعة global constants المُعرّفة في ملف وحدة .SysUtils نذكر منها :

```
DateSeparator: Char;
ShortDateFormat: string;
LongDateFormat: string;
TimeSeparator: Char;
TimeAMString: string;
TimePMString: string;
ShortTimeFormat: string;
LongTimeFormat: string;
ShortMonthNames: array [1..12] of string;
LongMonthNames: array [1..12] of string;
ShortDayNames: array [1..7] of string;
LongDayNames: array [1..7] of string;
```

يوجد المزيد من الثوابت الجامعة تتعلق بالعُملة و صياغة رقم النقطة العائمة. يمكنك الحصول على قائمة كاملة بها من ملف مساعدة دلفي تحت العنوان *Currency and date/time formatting variables*.

انواع ويندوز الخاصة

أنواع البيانات السابقة التحديد والتي سبق أن استعرضناها حتى الآن هي جزء من لغة باسكال. تتضمن دلفي أنواع بيانات أخري مجددة من قبل ويندوز. أنواع البيانات هذه ليست جزءاً مكمّلا في اللغة، ولكنها جزءا من مكتبات libraries ويندوز. أنواع ويندوز تتضمن أنواع ابتدائية جديدة (مثل *DWORD* و*UINT* ، و العديد من التسجيلات) records أو بنيات(structues ، و عددا من الأنواع المؤشّرةpointer ، وغيرها .

من بين أنواع بيانات ويندوز، فإن النوع الأكثر أهمية تمثله المماسكhandles ، الفصل 9 يناقش ذلك .

تلبيس النوع و تحويلات النوع

كما رأينا، لا يمكنك تخصيص متغير لآخر من نوع مختلف. إذا أردت ذلك، يوجد خياران . الخيار الأول هو تلبيس النوع *typecasting* ، والذي يستخدم رمز وظائفي بسيط ، باسم نوع البيانات المطلوب :

```
var
  N: Integer;
  C: Char;
  B: Boolean;
begin
  N := Integer ('X');
  C := Char (N);
  B := Boolean (0);
```

يمكنك التلبيس بين أنواع البيانات ذات نفس الحجم. عادة مايكون الأمر مأمونا عند التلبيس بين الأنواع التراتبية، أو بين الأنواع الحقيقية، ولكن يمكنك التلبيس بين أنواع مؤشّرة pointer (و أيضا الكينونات objects) طالما تكون مدركا لما تفعله .

التلبيس، بصفة عامة، عادة برمجية خطيرة، لأنه يسمح لك بالوصول إلى قيمة كما لو أنها ممثلة بشكل آخر. و طالما ان التمثيلات الداخلية لأنواع البيانات عموما غير متجانسة، فأنت تخاطر بالتسبب بأخطاء صعبة التتبع. لهذا السبب، يجب عليك عموما تجنب عمليات تلبيس النوع .

الخيار الثاني هو استخدام إجرائيات تحويل النوع. الإجرائيات الخاصة بمختلف أنواع التحويلات تم تلخيصها في الجدول 3.4. بعض هذه الإجرائيات تعمل مع أنواع بيانات سيجري الحديث عنها في الأقسام التالية. لاحظ ان الجدول لا يحوي الإجرائيات الخاصة بالأنواع الخاصة (مثل *TDateTime* أو المتبيان variant) أو الإجرائيات الموجّه خصّيصا للتشكيل والصياغةformating ، مثل الإجرائيات الفعّالة *Format* و*FormatFloat* .

جدول 3.4: إجرائيات النظام الخاصة بتحويل البيانات

الإجرائية	البيان
Chr	تحويل ترقم تراتبي الى حرف.ANSI
Ord	تحويل قيمة نوع تراتبي إلى رقم يشير إلى ترتيبه.
Round	تحول قيمة نوع حقيقي إلى قيمة نوع صحيح، تقريب القيمة.
Trunc	تحول قيمة نوع حقيقي إلى قيمة نوع صحيح، تشذيب القيمة.
Int	ارجاع الجزء الصحيح بقيمة نقطة عائمة.
IntToStr	تحويل الرقم إلى جملة.
IntToHex	تحويل الرقم إلى جملة بتمثيل ستعشري.hexadecimal
StrToInt	تحويل جملة إلى رقم، مع إبداء رفض لو كانت الجملة لا تمثل رقما صحيحا و سليما.
StrToIntDef	تحويل جملة إلى رقم، مع استخدام القيمة الإبتدائية إذا كانت غير سليمة.

Val	تحويل الجملة إلى رقم (إجرائية قديمة في تربو باسكال، محتفظ بها من أجل التوافقية.)
Str	تحويل رقم إلى جملة، باستخدام محددات الصياغة إجرائية قديمة في تربو باسكال، محتفظ بها من أجل التوافقية.)
StrPas	تحويل جملة مقفلة بصفر null-terminated إلى جملة بنسق باسكال-Pascal style. هذا التحويل يتم آليا بالنسبة بالنسبة للجمل نوع AnsiString في دلفي 32-بت. (انظر إلى القسم الخاص بالجمل لاحقا في هذا الفصل).
StrPCopy	نسخ جملة بنسق باسكال إلى جملة مقفلة بصفر. هذا التحويل يتم بتلبيس بسيط لنوع PChar في دلفي 32-بت. (انظر إلى القسم الخاص بالجمل لاحقا في هذا الفصل).
StrPLCopy	ينسخ قسما من جملة بنسق باسكال الى جملة مقفلة بصفر.
FloatToDecimal	تحويل قيمة نقطة عائمة إلى تسجيلة record تتضمن التمثيل العشري (exponent)، أعداد، علامة)
FloatToStr	تحويل قيمة نقطة عائمة إلى ما يمثلها كجملة باستخدام الصياغة الافتراضية.
FloatToStrF	تحويل قيمة نقطة عائمة إلى ما يمثلها كجملة باستخدام صياغة محددة.
FloatToText	تحويل قيمة نقطة عائمة إلى حيّز جملةstring buffer ، باستخدام صياغة حددة.
FloatToTextFmt	مثل الإجرائية السابقة، تحويل قيمة نقطة عائمة إلى حيّز جملة، باستخدام صياغة مُحددة.
StrToFloat	تحويل جملة باسكال إلى قيمة نقطة عائمة.
TextToFloat	تحويل جملة مقفلة بصفر إلى قيمة نقطة عائمة.

ملخّص

في هذا الفصل استكشفنا المفهوم الأساسي للنوع في باسكال. لكن اللغة لديها ميزة أخرى مهمة جدا: انها تسمح للمبرمجين بتعريف أنواع بيانات جديدة خاصة، تدعى بأنواع البيانات المحددة بالمستعمل user-defined data types. هذا هو موضوع الفصل التالي .

الفصل التالي :أنواع البيانات المحدّدة بالمستعمل

الفصل 4
أنواع البيانات المحددة بالمستعمل

ماركو كانتو: أساسي باسكال

اضافة إلى مفهوم النوع، فإن أحد أعظم الأفكار التي قدمتها لغة باسكال هي القدرة على تحديد أنواع بيانات جديدة في البرنامج. يستطيع المبرمجون تحديد أنواع بيانات خاصة بهم بواسطة مشيّدات النوع *type constructors* ، مثل أنواع المدى الفرعيsubrange ، نوع مصفوفة array، نوع تسجيلةrecord ، نوع سرديenumerated ، نوع مؤشّر pointer ، ونوع فئة set.أكثر أنواع البيانات المحددة بالمستعمل أهمية هي الطبقة أو الفصيلةclass ، و التي هي جزء من اضافات الاتجاه الكائني object-oriented في باسكال، و الغير مشمولة في هذا الكتاب .

إذا كنت تعتقد بأن مشيّدات النوع أمر عام في لغات البرمجة، فأنت على صواب، لكن باسكال كانت أول لغة قدّمت هذه الفكرة بطريقة منظمة و دقيقة جدا .

الأنواع المسمّاة وغير المسمّاة

هذه الأنواع يمكن إعطاؤها اسما للرجوع اليه لاحقا أو أن تطبّق على المتغيرات مباشرة. عندما تعطي اسما لنوع، يجب أن توفّر مساحة خاصة في التوليف، مثل الآتي :

```
type
  // subrange definition
  Uppercase = 'A'..'Z';

  // array definition
  Temperatures = array [1..24] of Integer;

  // record definition
  Date = record
    Month: Byte;
    Day: Byte;
    Year: Integer;
  end;

  // enumerated type definition
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);

  // set definition
  Letters = set of Char;
```

بناءات تحديد نوع مشابهة يمكن استخدامها مباشرة لتعريف متغير من غير تسمية صريحة للنوع، كما هو في التوليف التالي :

```
var
  DecemberTemperature: array [1..31] of Byte;
  ColorCode: array [Red..Violet] of Word;
  Palette: set of Colors;
```

ملاحظة :عموما يجب عليك أن تتجنّب استخدام الأنواع غير المسمّاة *unnamed* مثل التوليف السابق، و ذلك لأنك لن تستطيع أن تمرّرها كمحدّدات الى الإجرائيات، أو أن تقوم بتصريح متغيّرات أخرى من نفس النوع. إن قواعد توافقية النوع في باسكال تعتمد في الواقع على أسماء الأنواع، و ليس على ما تم تحديده فعليا من أنواع. إن متغيّرين من نوعين متطابقين لايعتبران متوافقين، إلا إذا كانا نوعاهما يحملان نفس الإسم حرفيا، أما الأنواع غير المسمّاة فإنها تُعطى أسماء داخلية من قبل المجمّع .compiler حاول أن تتعوّد على عمليات تحديد نوع البيانات في كل مرّة تحتاج فيها إلى متغير مركب و غير بسيط، و لن تأسف على الوقت الذي صرفته لذلك .

و لكن ماذا تعني تحديدات النوع هذه؟ سأقدم بعض الشروح لأولئك الذين هم غير متعوّدين على بناءات باسكال للنوع. سوف أحاول أيضا شرح الفروقات لنفس البناءات في لغات البرمجة الأخرى. لذلك قد يهمّك قراءة الأقسام التالية حتى لو كنت معتادا على شكل تحديدات النوع المضروبة كمثال أعلاه. أخيرا، سأعرض بعض أمثلة دلفي و أقدم بعض الأدوات التي ستسمح لك بالوصول إلى معلومات عن النوع بطريقة حية .

أنواع مدى فرعي

نوع المدى الفرعي subrange يحدد مدىً من القيم داخل نوع مدى آخر (من هنا اسم مدى فرعي). يمكنك تحديد مدى فرعي لنوع صحيح، من 1 إلى 10 او من 100 إلى 1000، أو تستطيع ان تحدد مدى فرعي من نوع حرفChar ، كما في :

```
type
  Ten = 1..10;
  OverHundred = 100..1000;
  Uppercase = 'A'..'Z';
```

عند تحديد المدى الفرعي، لا تحتاج إلى تحديد اسم النوع الأساسي. أنت تحتاج فقط إلى تقديم ثابتين من نفس ذلك النوع. النوع الأصلي يجب أن يكون من نوع تراتبيordinal ، و النوع الناتج سوف يكون نوعا تراتبيا آخر .

عندما تكون قد حدّدت مدى فرعيا، يمكنك رسميا أن تخصص قيمة داخل ذلك المدى. هذا التوليف يعتبر صحيحا :

```
var
  UppLetter: UpperCase;
begin
  UppLetter := 'F';
```

ولكن هذا ليس كذلك :

```
var
  UppLetter: UpperCase;
begin
  UppLetter := 'e'; // compile-time error
```

كتابة التوليف السابق ينتج عنه خطأ في زمن التجميع compile-time error بالرسالة التالية:

"Constant expression violates subrange bounds."

تعبير لثابت يتخطّى حدود المدى. أما إذا كتبت بدلا من ذلك التوليف التالي :

```
var
  UppLetter: Uppercase;
  Letter: Char;
begin
  Letter :='e';
  UppLetter := Letter;
```

فإن دلفي ستقوم بتجميعه. و لكن في زمن التشغيل run-time ، و إذا قمت بتمكين خيار المجمّع لتفحّص المدى Range Checking فستحصل على رسالة خطأ *Range check error* . (تمكين الخيار من خلال مربع Project Options ثم صفحة Compiler).

ملاحظة :أقترح بأن تقوم بتمكين خيار المجمّع هذا عندما تقوم بتطوير برنامج ما، حتى يكون هذا البرنامج أكثر متانة و أكثر سهولة عند تنقية أخطائه، ففي هذه الحالة و عند وجود أخطاء سوف تحصل على تنبيهات صريحة و واضحة و ليس مجرّد سلوك لايمكن التكهّن به. عند البناء النهائي للبرنامج تستطيع اخماد هذا الخيار، لجعله أسرع قليلا. عموما، الفرق عمليا صغير جدا، و لهذا السبب أنصح بأن تقوم بتمكين كل خيار ات التفحّص في زمن التشغيل، حتى عند شحن البرنامج. نفس الأمر ينطبق على باقي خيارات الفحص في زمن التشغيل، مثل فحص احتمالات الفوران overflow و التكدّسstack .

الأنواع السردية

تشكّل الأنواع السردية enumerated نوعا تراتبيا آخرا محددة بالمستعمل. فبدلا من الإشارة إلى مدى من نوع موجود، فإنك في السردية تقوم بعرض القيم المحتملة للنوع .بمعنى آخر، السردية هي قائمة بالقيم. فيما يلي بعض الأمثلة :

```
type
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);
  Suit = (Club, Diamond, Heart, Spade);
```

كلّ قيمة في القائمة لها ما يصاحبها من ترتيب*ordinalty* ، بدءاً من الصفر .عندما تطبق وظيفة ordعلى على قيمة من نوع سردي، تتحصّل على نفس هذه القيمة المبتدئة بصفر. مثلا، *Ord(Diamond)*ترجع 1 .

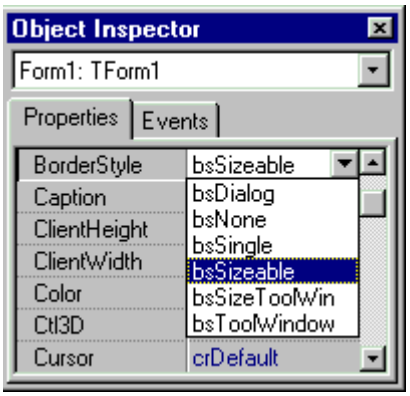
ملاحظة :يمكن للأنواع السردية أن يكون لها تمثيلات داخلية مختلفة. ابتدائيا، تستخدم دلفي تمثيل 8-بت، ما لم يكن هناك أكثر من 256 قيمة مختلفة، في هذه الحالة تستخدم تمثيل 16-بت. يوجد أيضا تمثيل 32-بت، و الذي قد يكون مفيدا لأغراض التوافقية مع مكتبات س و س++. يمكنك في الواقع تغيير السلوك الإبتدائي، و طلب تمثيل أكبر، باستخدام توجيه المجمّع *$Z*.

مكتبة المكوّنات المرئية VCL (Visual Component Library) في دلفي تستخدم الأنواع السردية في عدّة أماكن. مثال ذلك، نمط الحدود لنموذج form معرّف كالتالي :

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsSizeToolWin, bsToolWindow);
```

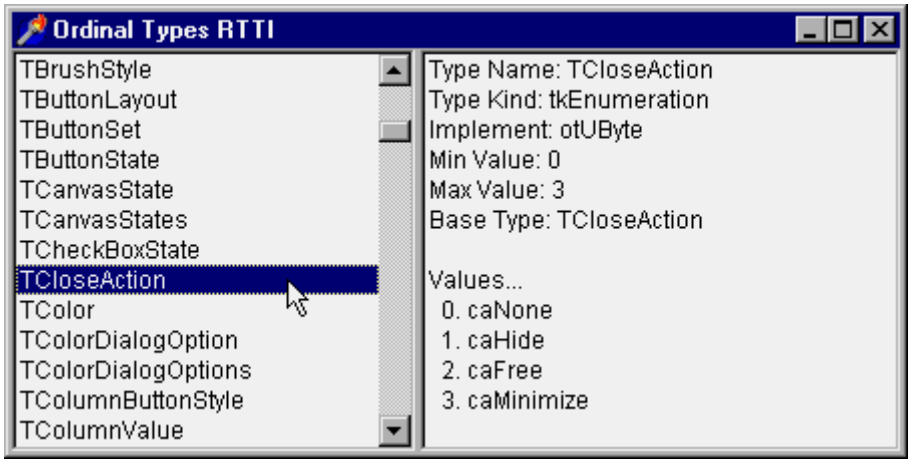
عندما تكون السِمة property قيمتها سردية، عادة ما تختار من قائمة القيم المعروضة في معاين الكائنات Object Inspector ، كما هو معروض في الشكل 4.1 .

الشكل 4.1: سمة نوع سردي في معاين الكائنات Object Inspector



يعرض ملف مساعدة دلفي بصفة عامة القيم المحتملة لأي نوع سردي .enumeration كبديل يمكنك استخدام برنامج *OrdType* والمتوفر في موقع www.marcocantu.com ، ليتسنى لك رؤية ما لدى دلفي من أنواع سردية، فئات، مدى فرعي، أو أي نوع تراتبي آخر. يمكنك رؤية مثال لمخرجات هذا البرنامج في الشكل 4.2 .

الشكل 4.2: معلومات مفصّلة عن نوع سردي، كما يعرضه برنامج OrdType (متوفّر بموقعي على الشبكة).



نوع فئة

أنواع الفئة set تشير إلى مجموعة من القيم. فهي صفّ من القيم المحتملة ذات نفس النوع التراتبي ordinal الذي للفئة. هذه الأنواع التراتبية عادة ما تكون محدودة، و غالبا ما يتم تمثيلها بسردية enumeration أو مدى فرعي .subrange إذا أخذنا المدى الفرعي 1..3، فإن القيم المحتملة للفئة التي ستبنى على هذا المدى سوف تتضمن إما :فقط 1، فقط 2، فقط 3، كلا 1 و 2، كلا 1 و 3، كلا 2 و 3، كل القيم الثلاث، أو لا واحدة منهم .

أي متغيّر عادة ما يضّم أحد هذه القيم المحتملة من المدى الذي يسمح به نوعه. أما المتغير الذي من نوع فئة، يمكنه أن يحوي أكثر من قيمة واحدة، فقد يحوي لا شيء، واحدة، إثنتان، ثلاثة، أو قيم أكثر ضمن المدى. بل يمكنه أن يتضمن كلّ القيم. ها هنا مثال عن فئة :

```
type
  Letters = set of Uppercase;
```

الآن يمكنني أت أعرّف متغيرا من هذا النوع و أخصّص له بعض القيم حسب النوع الأصلي. للإشارة إلى بعض القيم في فئة، تقوم بكتابة سردا مفصول بفاصلةcomma-separated ، و محصورا بين قوسين مربعين .brackets التوليف التالي يعرض تخصيصات لمتغير: بقيم متعدّدة، بقيمة واحدة، و بقيمة فارغة :

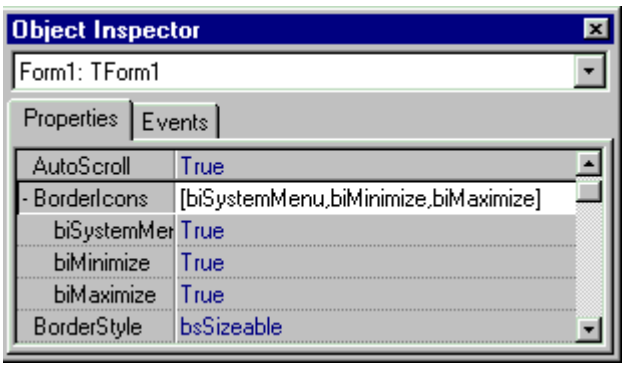
```
var
  Letters1, Letters2, Letters3: Letters;
begin
  Letters1 := ['A', 'B', 'C'];
  Letters2 := ['K'];
  Letters3 := [];
```

في دلفي، تستخدم الفئة عموما للإشارة إلى قيم غير حصرية. مثلا، سطرا التوليف التاليان (و هما جزء من مكتبة دلفي) يعرّفان سردا enumeration من الأيقونات المحتملة لإطار نافذة windowثم ما يوافق ذلك من نوع فئة :

```
type
  TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
  TBorderIcons = set of TBorderIcon;
```

في الواقع، أية نافذة إما أن لا يكون لها أيا من هذه الأيقونات، واحدة منها، أو أكثر من ذلك. عند التعامل مع معاين الكائنات) Object Inspector انظر الشكل 4.3(، بإمكانك إعطاء قيم الفئة بتوسيع الإختيار (بلمسة مزدوجة على اسم السِمة أو بلمس علامة الجمع يسار الاسم) و تبادل تمكين أو إخماد حضور كلّ قيمة .

الشكل 4.3: سمة من نوع فئة في معاين الكائنات Object Inspector



سِمة أخرى porperty مبنية على نوع فئة و هي نمط الخط .font style القيم المحتملة تشير لخط داكنbold ، مائلitalic ، تحته خطunderline ، و خط مقطوع .strikethrough بالطبع نفس الخط قد يكون مائلا و داكنا معا، أو بلا أي نمط، أو أن يملك أنماط الخط كلّها. لهذا السبب تم تعريف نمط الخط كفئة. و يمكنك تخصيص قيم لهذه الفئة في توليف برنامج كالتالي :

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles
```

يمكنك اجراء عمليات على الفئة بطرق مختلفة، بما في ذلك جمع متغيرين من نفس نوع الفئة (لنكون أكثر دقّة، حساب اتحاد فئتين :)

```
Font.Style := OldStyle + [fsUnderline]; // two sets
```

مرّة أخرى، تستطيع استخدام مثال OrdType والموجود في دليل TOOLS في التوليف المصدري للكتاب، لرؤية قائمة بالقيم المحتملة للعديد من الفئات المعرّفة في مكتبة مكونات دلفي .

أنواع مصفوفة

أنواع مصفوفة Array types تحدد قوائما بعدد ثابت من العناصر من نوع محدد. أنت ستستعمل عادة دليلا *index* بين قوسين مربعين للوصول إلى أحد عناصر المصفوفة .الأقواس المربعة تستعمل أيضا لتحديد القيم المحتملة لعدد العناصر عند تعريف المصفوفة. مثلا، يمكنك تحديد مجموعة من 24 صحيحا بالتوليف التالي:

```
type
  DayTemperatures = array [1..24] of Integer;
```

عند تعريف المصفوفة، تحتاج إلى تمرير نوع مدى فرعي subrange بين قوسين مربعين، أو تحديد نوع مدى فرعي جديد خاص باستخدام ثابتين constant من نوع تراتبي. المدى الفرعي هذا يحدد الأدلة indexes المسموح بها في المصفوفة. و حيث انك ستحدد كل من الدليل الأعلى و الدليل الأدنى في المصفوفة، فإن الأدلة لا حاجة لها بأن تبدأ بصفرzero-based ، عكس الحال في س، و س++، و جافا، و لغات برمجة أخرى .

و حيث أن أدلة المصفوفة مبنية على مدى فرعيsubrange ، يمكن لدلفي أن تتفحّص مداها كما رأينا سابقا. فأي ثابت خارج النطاق في مدى فرعي سينتج خطأ عند زمن التجميع؛ كما أن أي دليل خارج النطاق يستخدم في زمن التشغيل ينتج عنه خطأ زمن التشغيل، في حالة تمكين الخيار ذو العلاقة في المجّمع .

باستخدام تعريف المصفوفة أعلاه، يمكنك جعل قيمة المتغير *DayTemp1* من نوع *DayTemperatures*كالتالي :

```
type
  DayTemperatures = array [1..24] of Integer;

var
  DayTemp1: DayTemperatures;

procedure AssignTemp;
begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;
  DayTemp1 [25] := 67; // compile-time error
```

قد يكون للمصفوفة أكثر من بعد واحد، كما في المثال التالي :

```
type
  MonthTemps = array [1..24, 1..31] of Integer;
  YearTemps = array [1..24, 1..31, Jan..Dec] of Integer;
```

نوعي المصفوفة هذين بُنيا على النوع الأساسي نفسه. يمكنك تعريفهما باستخدام أنواع البيانات السابقة، كما في التوليف التالي :

```
type
 MonthTemps = array [1..31] of DayTemperatures;
 YearTemps = array [Jan..Dec] of MonthTemps;
```

هذا التعريف يقلب ترتيب الأدلّة كما هو أعلاه، لكنّه أيضا يسمح بتخصيص كُتل كاملة فيما بين المتغيرات. مثلا، التعليمة التالية تنسخ درجات حرارة temperatures شهر يناير إلى فبراير :

```
var
  ThisYear: YearTemps;
begin
  ...
  ThisYear[Feb] := ThisYear[Jan];
```

يمكنك أيضا تعريف مصفوفة تبدأ بصفر *zero-based* ، نوع مصفوفة حدّها الأدنى يساوي صفر. عموما، استخدام حدّين منطقيين أكثر يعدّ ميزة، حيث لا تضطر إلى استخدام الدليل 2 للوصول إلى العنصر 3، و هكذا. إلا أن ويندوز وبصورة ثابتة تستخدم مصفوفات تبدأ بصفر (لأنها مؤسسة على لغة س)، و مكتبة مكونات دلفي تتجه لفعل نفس الشيء .

إذا احتجت إلى الاشتغال على مصفوفة، يمكنك دائما اختبار حدّيها باستخدام الوظيفتين النمطيتين *Low*و*High* ، و اللتين ترجعان الحدّ الأدنى والحدّ الأعلى. يُنصح بشدّة باستخدام *Low* و *High* عند التعامل مع المصفوفة، خاصة في الحلقاتloops ، ما دامتا تجعلان من التوليف غير محتاج لمعرفة نطاق المصفوفة مقدما. لاحقا يمكنك تغيير النطاق المعرّف لأدلة المصفوفة، و يبقى التوليف الذي يستعمل *Low* و *High* بدون تغيير و مستمرا في عمله. أما إذا كتبت حلقة loop مع تحديد صريح لنطاق المصفوفة، فستجد نفسك مضطرا لتحديث التوليف الخاص بالحلقة كلما تغيّر حجم المصفوفة *Low* .و *High* تجعلان من توليفك أكثر سهولة عند الصيانة و أكثر اعتمادية .

ملاحظة :بالمناسبة، لايوجد أي إرهاق تشغيلي عند إستعمال *Low* و *High* مع المصفوفات. ففي زمن التجميع compile-time يتم ارجاعهما إلى ثابتينconstant ، و ليس إلى استدعاءات حقيقية لوظيفة. هذه الحلول التي تتم في زمن التحويل للتعبيرات و استدعاءات الوظائف تحدث أيضا لكثير من وظائف النظام البسيطة الأخرى .

كثيرا ما تستعمل دلفي المصفوفات في شكل سِمات نوع مصفوفة .array properties نحن رأينا بالفعل سِمة مماثلة في مثالTimeNow ، للوصول إلى سِمة *Items* عناصر في مكوّن.ListBox سأقوم بعرض بعض الأمثلة الاضافية لسِمات نوع مصفوفة في الفصل التالي، عند الحديث عن حلقات loops دلفي .

ملاحظة:أدخلت دلفي 4 المصفوفات الحيوية في اوبجكت باسكال، و هي المصفوفات التي يمكن أن يتغير حجمها في زمن التشغيل متخذة لنفسها ما يناسبها من كمية من الذاكرة، استخدام المصفوفات الحيوية أمر سهل، لكن في مناقشتنا هذه عن باسكال شعرت بأنها ليست موضوعا مناسبا لتغطيته. يمكنك أن تجد وصفا لمصفوفات دلفي الحيوية في الفصل 8 .

Record Types

نوع تسجيلة record types تحدّد مجموعة ثابتة من عناصر ذات أنواع مختلفة. كل عنصر، أو حقل *field* ، له نوعا خاصا به. تعريف نوع تسجيلة تعرض قائمة بكل هذه الحقول، تعطي لكل منها اسما لتستخدمه لاحقا من أجل الوصول إليه .

ها هنا عرضا بسيطا لتعريف من نوع تسجيلة، ثم تعريف لمتغير من هذا النوع، و بعض التعليمات التي تستخدم هذا المتغيّر :

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: Date;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

يمكن اعتبار الطبقات Classes و الكينونات objects امتدادا لنوع تسجيلة. مكتبات دلفي تتجه لاستخدام أنواع طبقة عوضا عن نوع تسجيلة، و لكن يوجد العديد من أنواع تسجيلة في مكتبات APIويندوز .

يمكن لأنواع تسجيلة أن يكون لها جزءا متباينا variant part ؛ عدّة حقول يمكن توجيهها لنفس مساحة الذاكرة، حتى إذا كان لها نوع بيانات مختلف. (يطابق هذا مفهوم union في لغة س.) فكبديل، يمكنك استخدام الحقول أو مجموعات الحقول المتباينة للوصل إلى نفس موقع الذاكرة ضمن التسجيلة، لكنها تتعامل مع هذه القيم من زوايا مختلفة .الإستخدامات الرئيسية لهذه الأنواع هي لتخزين بيانات متشابهة لكن مختلفة، و للحصول على تأثير شبيه بتلبيس النوع typecasting) الأمر الذي يعد أقل فائدة الآن حيث أن تلبيس النوع تم ادخاله أيضا في باسكال). ان استخدام أنواع تسجيلة متباينة تم استبداله بصورة كبيرة بمفاهيم الاتجاه للكائن-object oriented و بتقنيات حديثة أخرى، بالرغم أن دلفي تستخدمها في بعض الحالات الغريبة .

استخدام نوع تسجيلة متباينة ليست آمنة النوع type-safe ، و ليست من العادات البرمجية التي ينصح بها، خاصة مع المبتدئين. بالتأكيد يمكن للمبرمجين الخبراء استخدام أنواع تسجيلة متباينة، بل أن لُبّ مكتبات دلفي تقوم باستخدامها، لكنك، على كل حال، لست محتاجا لأن تتعامل معها حتى تصبح بالفعل خبيرا في دلفي .

Pointers

نوع مؤشّر pointer يحدّد متغيرا يحوي عنوان متغير آخر في الذاكرة ذو نوع بيانات محدد (أو نوع غير محدد). لذا فإن متغير المؤشر و بطريقة غير مباشرة يشير إلى قيمة .تعريف نوع مؤشّر لا يستلزم كلمة مفتاحية معينة، انه يستعمل حرفا خاصا بدلا من ذلك .الحرف أو الرمز الخاص هو علامة الادراج(^) caret :

```
type
  PointerToInt = ^Integer;
```

حالما عرّفت متغيرا مؤشرا، يمكنك أن تخصص له العنوان الخاص بمتغير آخر من نفس النوع، باستخدام العامل @ :

```
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  // change the value in two different ways
  X := 10;
  P^ := 20;
```

عندما يكون لديك المؤشّر *P* ، فإنه بواسطة التعبير *P* أنت تشير إلى موقع في الذاكرة يشير إليه المؤشر، و بواسطة التعبير *P^* أنت تشير إلى المحتويات الفعلية في موقع الذاكرة هذا .لهذا السبب في التوليف السابق فإن *P^* تطابق . *X*

بدلا من الإشارة إلى موقع ذاكرة موجود، يمكن للمؤشر أن يشير إلى مساحة جديدة في الذاكرة يتم تخصيصها بصورة حيّة (في منطقة من محيط الذاكرة) بواسطة الإجرائية .*New* في هذه الحالة، و عند إنتهاء حاجتك للمؤشر، عليك أيضا أن تتخلص من الذاكرة التي قمت بحجزها، بإستدعاء الوظيفة*Dispose* .

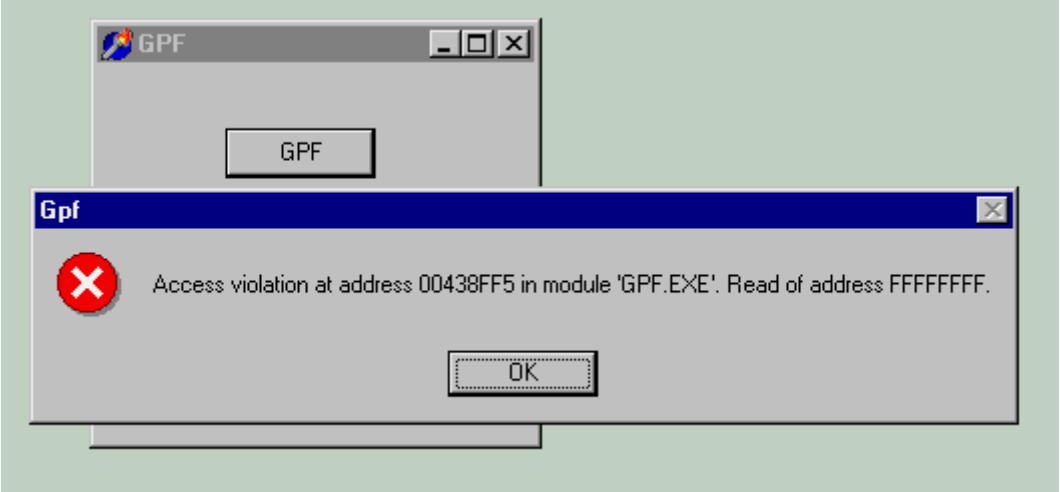
```
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  ShowMessage (IntToStr (P^));
  // termination
  Dispose (P);
end;
```

إذا كان المؤشر خاليا من أية قيمة، يمكنك تخصيص قيمة *nil* لا شيء له، بعدها تستطيع إختبار خلوّ المؤشر لمعرفة إذا ما هو حاليا يشير إلى قيمة. و هذا يستعمل عادة، لأن التأشير الخاطئ بمؤشر فارغ يسبّب انتهاكا لحرمة الدخول) .access violation أيضا يعرف بخطأ حماية عام general protection fault، أو GPF):

```
procedure TFormGPF.BtnGpfClick(Sender: TObject);
var
  P: ^Integer;
begin
  P := nil;
  ShowMessage (IntToStr (P^));
end;
```

يمكنك رؤية مثالا عن تأثير هذا التوليف بتشغيل مثال GPF) أو رؤيته كما في الشكل .(4.4
المثال يحوي أيضا أجزاء التوليف المعروض أعلاه .

الشكل 4.4: خطأ نظام ناتج عن الدخول بمؤشر خال، من مثال GPF.



في نفس البرنامج يمكن أن تجد مثالا لكيفية الوصول الآمن للبيانات. في هذه الحالة الثانية تمّ تخصيص المؤشر لمتغير محلي موجود، و يمكن استخدامه بأمان، لكني أضفت إليه فحص أمان للتأكيد :

```
procedure TFormGPF.BtnSafeClick(Sender: TObject);
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  X := 100;
  if P <>  nil then
    ShowMessage (IntToStr (P^));
end;
```

تعرّف دلفي أيضا نوع بيانات مؤشر تدلّ على مؤشرات بلانوع untyped (مثل *void** في لغة س). إذا استخدمت مؤشرات بلا نوع يجب أن تستخدم *GetMem* بدلا من *New*. إجرائية *GetMem* مطلوبة في كل مرة يكون فيها حجم متغير الذاكرة المطلوب حجزه غير محدد .

حقيقة أن المؤشرات pointers نادرة ما تكون ضرورية، هي ميزة مثيرة للإهتمام في هذه البيئة. لا شك، بأن فهم المؤشرات مهمّ للبرمجة المتقدمة و لفهم متكامل لنماذج كائنات دلفي، التي تستخدم المؤشرات "من خلف الكواليس ".

ملاحظة :بالرغم من أنك لاتستعمل المؤشرات غالبا في دلفي، فأنت عادة ما تستعمل بنية شبيهة جدا، ما يعرف بالإشارة reference. كل حضور لكائن object هو في الواقع مؤشر أو إشارة صريحة لبياناته الفعلية. عموما، هذا الأمر يعد مغيّبا بالكامل عن المبرمج، الذي سيستعمل متغيرات كائن تماما مثله مثل أي نوع بيانات آخر .

أنواع ملفّ

مشيّد نوع باسكال آخر هو نوع ملف *file*. أنواع ملف File types تمثل الملفات الفعلية بقرص التخزين، مؤكّدة غرابة لغة باسكال. يمكنك تعريف نوع بيانات ملف جديد كالتالي :

```
type
  IntFile = file of Integer;
```

بعد ذلك تستطيع فتح ملف فعلي مرتبطا بهذه البنية، وتقوم بكتابة قيم بأرقام صحيحة فيه أو قراءة القيم الحالية منه .

ملاحظة أخرى :الأمثلة المتعلقة بالملفات كانت جزءا من الإصدارات القديمة لكتاب Mastering Delphiو أخطّط لإضافتها هنا كذلك .

التعامل مع الملفات في باسكال يسيط جدا، لكن في دلفي توجد أيضا العديد من المكونّات components القادرة على تخزين أو تحميل محتوياتها من و إلى ملف. هناك بعض الدعم للتسلسليةserialization ، و ذلك على هيئة دفقاتstreams ، و يوجد أيضا دعما لقواعد البيانات database.

ملخّص

ناقش هذا الفصل أنواع بيانات محددة بالمستعمل، مكملا تغطيتنا لنظام أنواع باسكال. الآن نحن جاهزون للنظر للتعليمات التي توفرها اللغة للتعامل مع المتغيرات التي حدّدناها .

الفصل التالي :التعليمات

الفصل 5
التعليمات

ماركو كانتو:
أساسي باسكال

إذا كانت أنواع البيانات هي إحدى أساسات البرمجة بباسكال، فإن التعليمات statments هي الأخرى كذلك. تعليمات اللغة البرمجية تُبنى على أساس كلمات مفتاحية keywords و مفردات أخرى تسمح لك بأن توجّه للبرنامج تتابع العمليات المطلوب انجازها. عادة ما يتم تضمين التعليمات داخل إجراءات procedures أو وظائفfunctions ، كما سنرى في الفصل اللاحق. الآن سنركّز فقط على الأنواع الأساسية للأوامر التي يمكنك استخدامها لصنع برنامج .

التعليمات البسيطة والمركبة

تعليمة باسكال تكون بسيطة عندما لا تحتوي على أية تعليمات أخرى. كأمثلة على التعليمات البسيطة نجد تعليمات التخصيص assignment و استدعاءات الاجرائيات procedure calls. التعليمات البسيطة يتم الفصل بينها بفاصلة منقوطة semicolon :

```
X := Y + Z;  // assignment
Randomize;   // procedure call
```

عادة، تكون التعليمات جزءا من تعليمات مركّبة، مؤمّرة بعلامات البداية begin و النهاية end. التعليمة المركبة يمكن أن تظهر في مكان تعليمة باسكال عامة. ها هنا مثال :

```
begin
  A := B;
  C := A * 2;
end;
```

الفاصلة المنقوطة بعد آخر تعليمة قبل end ليست ضرورية. مثل التالي:

```
begin
  A := B;
  C := A * 2
end;
```

كلا النسختين صحيحتين. النسخة الأولى لها فاصلة منقوطة غير مجدية (لكنها لاتؤذي). هذه الفاصلة المنقوطة، في الواقع، هي تعليمة فارغة؛ تعليمة بدون توليف . code لا حظ هذا، أحيانا، التعليمات الفارغة يمكن استخدامها داخل الحلقات loops أو في حالات خاصة.

ملاحظة :بالرغم من أن الفاصلة المنقوطة الأخيرة لا تخدم أي غرض، إلا أنني أميل لإستخدامها مقترحا عليك القيام بنفس الأمر. أحيانا بعد كتابتك لبعض الأسطر ربما ترغب في إضافة تعليمة أخرى. فإذا كانت الفاصلة المنقوطة الأخيرة مفقودة فعليك أن تتذكّر اضافتها، لذا قد يكون من الأفضل اضافتها من المرة الأولى .

تعليمات التخصيص

التخصيصات assignments في باسكال تستخدم رمزي شارحة يساوي، ترميز غريب بالنسبة للمبرمجين الذين اعتادوا لغات اخرى. الرمز = و الذي يستعمل للتخصيص في بعض اللغات الأخرى، يستعمل في باسكال لاختبار المساواة .equality

ملاحظة:باستخدام ترميز مختلف للتفريق بين التخصيص و اختبار المساواة، يستطيع مجمّع باسكال (مثل مجمّع س) أن يترجم التوليف المصدري بصورة أسرع، لأنه لا يحتاج لفحص سياق التعليمة و كيفية استخدام الترميز لإستنتاج معناه. استعمال ترميزات مختلفة تساعد أيضا في جعل قراءة التوليف سهلة على الناس .

التعليمات الشرطية

التعليمة الشرطية conditional statement تستعمل لتنفيذ إحدى التعليمات التي تتضمنها أو عدم تنفيذ و لا واحدة منها. بالاعتماد على شيء من الاختبار. يوجد شكلين من التعليمات الشرطية: تعليمات if و تعليمات .case

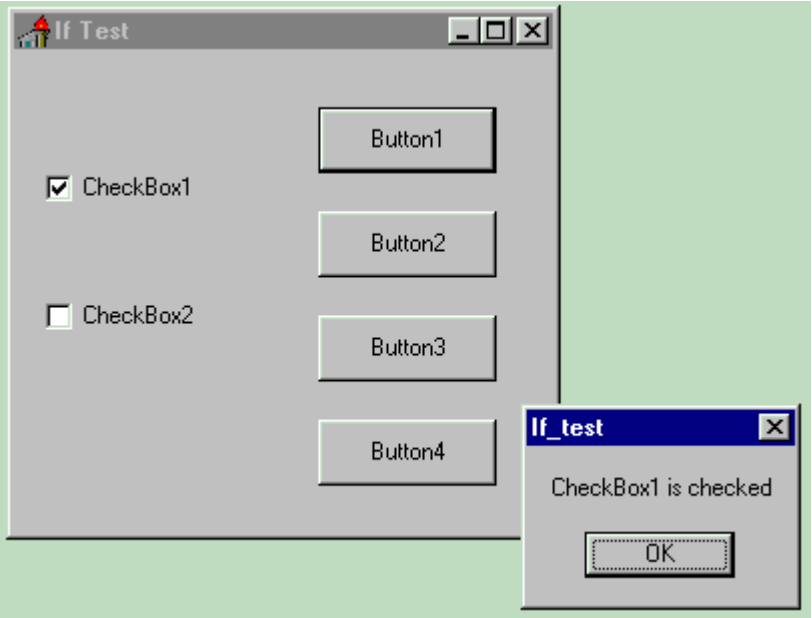
تعليمات If

تعليمة if يمكن استخدامها لتنفيذ تعليمة أخرى فقط إذا تحقق شرط معين(if-then) ، أو للإختيار بين بديلين .(if-then-else) يتم وصف الشرط بتعبير بولي .boolean مثال دلفي بسيط سيوضح كيفية كتابة تعليمات شرطية. أولا قم بإنشاء تطبيق application جديد، ثم ضع على النموذج formخانتي فحص check box و أربعة أزرار . buttons لا تغيّر أسماء الأزرار و خانات الفحص، قم بضغط مزدوج على كل زرّ لإضافة مناول handler للحدث *OnClick* الخاص بكل زرّ، ها هنا تعليمة *if* بسيطة للزر الأول :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // simple if statement
  if CheckBox1.Checked then
    ShowMessage ('CheckBox1 is checked')
end;
```

عندما تضغط على الزرّ، إذا كانت خانة الفحص الأول لديها علامة فحص، سيقوم البرنامج بعرض رسالة بسيطة (انظر الشكل 5.1). لقد استعملت وظيفة ShowMessage لأنها أسهل وظيفة في دلفي يمكن استعمالها لإظهار رسالة قصيرة للمستخدم .

الشكل 5.1: رسالة تعرض بواسطة مثال IfTest عندما تضغط على الزر الأول وتكون خانة الفحص الأول معلما .



إذا ضغطت على الزرّ و لم يحدث شيء، فهذا معناه أن خانة الفحص غير معلّمة. في مثل هذه الحالة، قد يكون من المستحسن جعل الأمر أكثر صراحة، كما هو في التوليف الخاص بالزرّ الثاني، و الذي يستعمل تعليمة if-then-else.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  // if-then-else statement
  if CheckBox2.Checked then
    ShowMessage ('CheckBox2 is checked')
  else
    ShowMessage ('CheckBox2 is NOT checked');
end;
```

لاحظ أنه لايمكنك وضع فاصلة منقوطة بعد التعليمة الأولى و قبل مصطلح*else* ، و إلا فإن المحوّل سيصدر خطأ جملة syntax error. إن تعليمة *if-then-else* في الواقع هي تعليمة واحدة، لذا لا يمكنك وضع فاصلة منقوطة في وسطها .

تعليمة *if* يمكن لها أن تكون أكثر تعقيدا. فالشرط يمكن تحويله إلى سلسلة من الشروط (باستخدام *and* و *or* و *not*)، أو ان تعليمة *if* تتفرع عنها تعليمة *if* أخرى. الزرّين الأخيرين في مثال IfTestيعرضان هاتين الحالتين :

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  // statement with a double condition
  if CheckBox1.Checked and CheckBox2.Checked then
    ShowMessage ('Both check boxes are checked')
end;
```

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  // compound if statement
  if CheckBox1.Checked then
    if CheckBox2.Checked then
      ShowMessage ('CheckBox1 and 2 are checked')
    else
      ShowMessage ('Only CheckBox1 is checked')
  else
    ShowMessage (
      'Checkbox1 is not checked, who cares for Checkbox2?')
end;
```

أنظر إلى التوليف جيدا و شغّل البرنامج لترى ما إذا فهمت كل ما سبق. عندما يكون لديك شكا حول بناء برمجي، فإن كتابة برنامج بسيط جدا مثل هذا يمكن أن يساعدك لتعلّم الكثير. بإمكانك اضافة المزيد من خانات التفحص و زيادة تعقيد هذا البرنامج البسيط، و إجراء أي اختبار ترغب به .

تعليمات Case

إذا ما أضحت تعليمات *if* لديك أكثر تعقيدا، يمكنك استبدالها في أية لحظة بتعليمات *case*. تعليمة *case* عبارة عن تعبير يستخدم لاختيار قيمة، قائمة بقيم محتملة، أو مدى من القيم. هذه القيم هي ثوابت constants ، و يجب أن تكون فريدة ومن نوع تراتبي ordinal. أحيانا، قد يوجد بها تعليمة else و التي يتم تنفيذها إذا لم تتوافق أي من الاحتمالات مع القيمة المعطاة. فيما يلي مثالين بسيطين :

```
case Number of
  1: Text := 'One';
  2: Text := 'Two';
  3: Text := 'Three';
end;

case MyChar of
  '+' : Text := 'Plus sign';
  '-' : Text := 'Minus sign';
  '*', '/': Text := 'Multiplication or division';
  '0'..'9': Text := 'Number';
  'a'..'z': Text := 'Lowercase character';
  'A'..'Z': Text := 'Uppercase character';
else
  Text := 'Unknown character';
end;
```

الحلقات في باسكال

للغة باسكال التعليمات التكرارية النمطية التي لمعظم لغات البرمجة، يتضمن هذا تعليمات *for* ، و*while* ، و . *repeat* معظم ما تفعله هذه الحلقات loops سيكون مألوفا اذا ما سبق وان استخدمت لغات برمجية أخرى، لذا سأغطّي هذه التعليمات بصورة مختصرة .

حلقة For

حلقة for في باسكال مبنية بصورة مقيدة على عدّاد counter ، و الذي يمكن زيادته أو تخفيضه في كل مرة يتم تنفيذ الحلقة. فيما يلي مثال بسيط لحلقة for مستخدمة لإضافة أول عشرة أرقام .

```
var
  K: Integer;
begin
  K := 0;
  for I := 1 to 10 do
    K := K + I;
```

نفس المثال كان يمكن كتابته باستخدام عدّاد معكوس:

```
begin
  K := 0;
  for I := 10 downto 1 do
    K := K + I;
```

حلقة for في باسكال أقل مرونة مقارنة بلغات أخرى (ليس بالإمكان تحديد معدّل زيادة إلا بواحد)، لكنها بسيطة و سهلة الفهم. إذا اردت اختبار شرط بتشعب أكبر، أو أردت ايجاد عدّاد بمواصفات خاصة، فأنت بحاجة إلى استخدام تعليمات *while* أو *repeat* ، عوضا عن حلقة for .

ملاحظة : عدّاد حلقة for ليس بالضرورة أن يكون رقما. يمكن له أن يكون قيمة من نوع تراتبي، مثل حرف أو نوع سردي .

تعليمات While وRepeat

الفرق بين حلقة *while-do* و حلقة *repeat-until* هو أن التوليف code داخل تعليمة repeat ينفّذ دائما، مرّة واحدة على الأقل. تستطيع بسهولة فهم السبب بالإطّلاع على المثال البسيط :

```
while (I < 100) and (J < 100) do
begin
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
end;

repeat
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

إذا كانت القيمة الإبتدائية في *I* أو *J* أكبر من 100، فإن التعليمات داخل حلقة repeat-until سيتم تنفيذها مرة على كل حال .

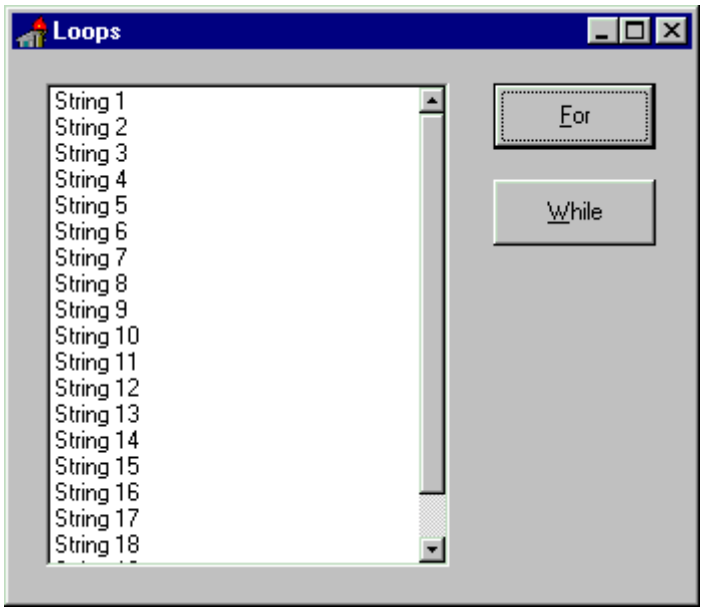
الفرق الرئيسي الآخر بين هذين الحلقتين هو أن حلقة *repeat-until* لديها شرط محجوز *reserved* . الحلقة سيتم تنفيذها حتى اللحظة التي لا يتم فيها تحقيق الشرط. عندما يتحقق الشرط، تتوقف الحلقة. هذا عكس حلقة *while-do* ، التي تنفذ ما دام الشرط موجبا. لهذا السبب عليّ أن أعكس الشرط في التوليف أعلاه للحصول على تعليمة مشابهة .

مثال عن الحلقات

لإستكشاف تفاصيل الحلقات، دعنا نستعرض مثال دلفي بسيط. برنامج Loops يبرز الفرق بين حلقة بعدّاد ثابت و حلقة بعدّاد عشوائي تقريبا. إبدأ بمشروع project جديد، ضع مربّع قائمةlist box و زرّيّن على النموذج form الرئيسي، قم بإعطاء الزرّين إسمين مناسبين (BtnFor و BtnWhile) و ذلك بتوصيف سِمة *Name* فيهما بواسطة معاين الكائنات Object Inspector. يمكن أيضا إزالة كلمة *Btn* من سِمة *Caption* و دائما اضف الحرف & للعنوان لتنشيط الحرف التالي له كمفتاح اختصار shortcut). فيما يلي ملخّص للوصف النصّي للنموذج :

```
object Form1: TForm1
  Caption = 'Loops'
  object ListBox1: TListBox ...
  object BtnFor: TButton
    Caption = '&For'
    OnClick = BtnForClick
  end
  object BtnWhile: TButton
    Caption = '&While'
    OnClick = BtnWhileClick
  end
end
```

الشكل 5.2: في كل مرة تضغط فيها زرّ For في مثالLoops ، يُملأ مربّع القائمة بأرقام متتالية .



الآن يمكنا اضافة بعض التوليف لحدثي OnClick في الزّرين. الزّر الأول بـه حلقة *for* بسيطة لعرض قائمة من الأرقام، كما هو في الشكل 5.2. قبل تنفيذ هذه الحلقة، التي تضيف عددا من الجمل إلى قيمة *Items* في مرّبع القائمة، تحتاج لتصفية محتويات القائمة نفسها .

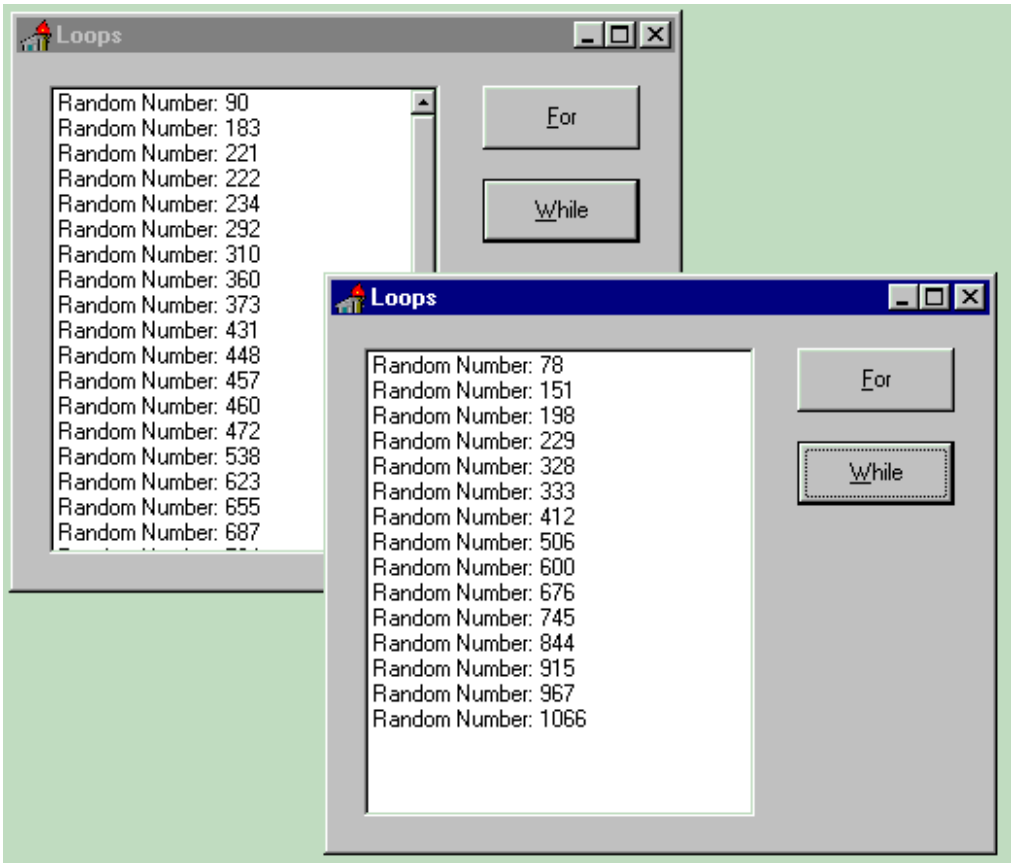
```
procedure TForm1.BtnForClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 1 to 20 do
    Listbox1.Items.Add ('String ' + IntToStr (I));
end;
```

التوليف المربوط بالزّر الثاني أكثر تعقيدا نوعا ما. ففي هذه الحالة، توجد حلقة while قائمة على عدّاد، يتم زيادته عشوائيا. لإنجاز ذلك، قمت باستدعاء الإجرائيةRandomize ، والتي تقوم بإعادة تهيئة مولّد الرقم العشوائي، و وظيفة Random بنطاق مداه 100. نتيجة هذه الوظيفة هو رقم بين 0 و 99، يتم إختيارها عشوائيا. سلسلة الأرقام العشوائية تتحكم في عدد مرّات تنفيذ حلقة while.

```
procedure TForm1.BtnWhileClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  Randomize;
  I := 0;
  while I < 1000 do
  begin
    I := I + Random (100);
    Listbox1.Items.Add ('Random Number: ' + IntToStr (I));
  end;
end;
```

كل مرة تضغط على زرّwhile ، تختلف الأرقام، لأنها تعتمد على مولّد لرقم عشوائي .الشكل 5.3 يعرض نتائج تنفيذين منفصلين لنفس زرّ while. لاحظ أنه ليس فقط الأرقام التي تم توليدها مختلفة، بل أيضا عدد العناصر مختلف. حلقة while هذه تُنتج عددا عشوائيا من العناصر. إذا ضغطت على زرّ while عدة مرّات، سوف ترى أن القائمة لديها عدد أسطر مختلف .

الشكل 5.3: محتويات القائمة في مثال Loops تتغير كل مرة تضغط فيها على زرّ while. لأن عدّاد الحلقة يزداد بقيمة عشوائية، في كل مرة تضغط فيها على الزر، يتمّ تنفيذ الحلقة بعدد مرات مختلف .



ملاحظة :يمكنك تحويل التدفق المعتاد لتنفيذ الحلقة باستخدام اجرائيات النظام *Break* و *Continue*.الأول يستخدم لعرقلة الحلقة؛ أما الثاني فيُستخدم للقفز مباشرة إلى اختبار الحلقة أو إلى مزيد العدّاد، بحيث يعيد الاستمرار مع الدورة التالية للحلقة (ما لم يكن الشرط صفرا أو أن العدّاد قد بلغ حدّه الأعلى). أيضا توجد الإجرائيتان *Exit* و*Halt* ، تسمح لك الأولى بالخروج حالا من الوظيفة أو الإجرائية التي فيها، والثانية تنهي عمل البرنامج .

تعليمة With

أخر نوع من تعليمات باسكال سأقوم بالتركيز عليه هي تعليمة *with* ، و التي تعدّ مميزة في هذه اللغة البرمجية (و تم إدخالها مؤخّرا في فيجوال بيسك) و مفيدة جدا في البرمجة بدلفي .

تعليمة *with* ليس إلا اختصار .shorthand عندما تحتاج إلى الاشارة الى متغير نوع تسجيلة record(و كائن object) فبدلا من تكرار اسمه في كل مرّة، يمكنك استخدام تعليمة *with* . مثال ذلك، بينما أقوم بعرض نوع تسجيلة كتبت التوليف التالي :

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: Date;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

باستعمال تعليمةwith ، بامكاني تحسين الجزء الأخير من التوليف، كالتالي :

```
begin
  with BirthDay do
  begin
    Year := 1995;
    Month := 2;
    Day := 14;
  end;
```

هذا الاسلوب يمكن استخدامه في برامج دلفي للإشارة الى المكوّنات components و أنواع الطبقات class الأخرى. مثلا، يمكننا إعادة كتابة الجزء الأخير من آخر مثال، مثالLoops ، باستخدام تعليمة with لمناولة العناصر items في القائمة :

```
procedure TForm1.WhileButtonClick(Sender: TObject);
var
  I: Integer;
begin
  with ListBox1.Items do
  begin
    Clear; // shortcut
    Randomize;
    I := 0;
    while I < 1000 do
    begin
      I := I + Random (100);
      // shortcut:
      Add ('Random Number: ' + IntToStr (I));
    end;
  end;
end;
```

عندما تتعامل مع المكوّنات components أو الطبقات classes في باسكال عموما، تسمح لك تعليمة with بالاستغناءعن كتابة بعض التوليف، خاصة بالنسبة للحقول المتفرعة .مثلا، لنفترض انك تريد تغيير حجم و لون قلم الرسم لنموذج .form يمكنك كتابة التوليف التالي :

```
Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clRed;
```

و لكنه بالتأكيد سيكون الأمر أسهل لو كتبت التوليف التالي :

```
with Form1.Canvas.Pen do
begin
  Width := 2;
  Color := clRed;
end;
```

عندما تقوم بكتابة توليفا متشعبا، يمكن لتعليمة with أن تكون فعالة و تعفيك من تعريف بعض المتغيرات المؤقتة، و لكنها لا تخلو من العيوب. فبامكانها أن تجعل من التوليف أقل مقروئية، خاصة عندما تتعامل مع كائنات مختلفة لكن لديها سمات متشابه أو متطابقة .

يوجد أيضا عيب آخر، و هو ان استعمال تعليمة *with* قد تسمح بأخطاء منطقية شفافة في التوليف يصعب على المجمّع تحسّسها. مثال ذلك :

```
with Button1 do
begin
  Width := 200;
  Caption := 'New Caption';
  Color := clRed;
end;
```

هذا التوليف يغير من عنوان *Caption* و عرض *With* الزرّ، و لكنه يؤثّر في سِمة اللون *Color* للشكل، و ليس في الزرّ! سبب هذا أن المكوّن *TButton* ليس لديه سِمةColor ، و حيث أن التوليف يتم تنفيذه داخل كائن form نموذج الذي لديه مثل هذه السِمة فإن الافتراض الأول يتجه له مباشرة. بدلا من ذلك إذا كتبنا :

```
Button1.Width := 200;
Button1.Caption := 'New Caption';
Button1.Color := clRed; // error!
```

فإن المجمّع سيصدر خطأ. عموما، يمكننا القول بأنه طالما أن تعليمة *with* استعملت معرّفات identifiersجديدة في نطاق التوليف الحالي، يمكننا اخفاء المعرّفات الموجودة، أو اننا و عن طريق الخطأ سنتعرض لمعرّف آخر في نفس النطاق (كما هو في المحاولة الأولى من

التوليف). مع الأخذ في الإعتبار هذا النوع من العيوب، فأنا أقترح عليك التعوّد على استعمال تعليمة*with* ، لأنه بإمكانها أن تكون مفيدة جدا، و أحيانا كثيرة تجعل من التوليف مقروءا بشكل أفضل.

يجب أن تتجنب الاستخدام المتعدد لتعليمات *with* ، مثل :

```
with ListBox1, Button1 do...
```

فالتوليف الذي سيتبعه سيكون غالبا غير مقروء و صعب التتبع، لأنه مع كل سِمة يتم تحديدها في هذا الحيّز تحتاج إلى استنتاج و معرفة المكوّن المقصود الذي تتبعه هذه السِمة، بالاعتماد على السِمات ذات العلاقة و ترتيب المكونات في تعليمة with .

ملاحظة : بصدد الكلام عن المقروئية، لا تملك باسكال تعليمات مثل *endif* أو *endcase*. إذا كان لتعليمة if حيز *begin-end* ، فإن نهاية هذا الحيز يحدد نهاية التعليمة. بالمقابل، تعليمة *case*تنتهي دائما بكلمة *end*. كل تعليمات *end* هذه، التي غالبا ما تكون واحدة فوق الأخرى، يمكن أن تجعل من التوليف صعب التتبع. فقط من خلال تتبع الهوامش يمكن معرفة تبعية كل *end*و لأية تعليمة. الطريقة العامة المتّبعة لحل هذه المشكلة و لجعل التوليف مفهوما أكثر هو في اضافة ملاحظة بعد تعليمة *end* تشير إلى تبعيتها، كما في :

```
if ... then
 ...
end; // if
```

ملخّص

شرح هذا الفصل كيفية توليف التعليمات الشَرطية و الحلقات. و بدلا من كتابة قائمة طويلة من هذه التعليمات، يتم تقسيم البرامج عادة إلى إجرائياتroutines ، إجراءات أو وظائف. هذا هو موضوع الفصل التالي، الذي سيقدم أيضا بعض الملامح المتقدمة في باسكال .

الفصل التالي :الإجراءات و الوظائف

الفصل 6
الإجراءات والوظائف

ماركو كانتو:
أساسي باسكال

فكرة أخرى مهمة ركّزت عليها باسكال هي مفهوم الإجرائيات routine. مبدئيا هي سلسلة من التعليمات تحت اسم خاص غير مكرّر، و التي يمكن تنشيطها في كل مرّة باستخدام اسمها. بهذه الطريقة تتجنب معاودة كتابة التعليمات مرّة بعد أخرى، فتتحصّل على نسخة واحدة من التوليف يمكنك بسهولة تعديله لصالح كامل البرنامج. من وجهة النظر هذه، يمكنك أن تنظر إلى الإجرائيات كآلية أساسية لتغليف التوليف. سأعود إلى هذا الموضوع لاحقا مع مثال بعد أن أقدم أولا الصيغة النحوية syntax لإجرائيات باسكال .

إجراءات و وظائف باسكال

في باسكال، الإجرائية routine يمكن افتراضها بشكلين: إجراء procedure و وظيفة function. نظريا، الإجراء هو عملية تقوم أنت كمبرمج بسؤال الحاسب كي ينجزها، الوظيفة هي حسبة تردّ قيمة. هذا الفرق يؤكّده حقيقة أن الوظيفة لها نتيجة result ، قيمة مسترجعة، بينما الإجراء ليس كذلك. كلا النوعين من الاجرائيات يكمن أن يكون لهما عدّة محدّدات parameters، من أنواع بيانات تعطى لها .

عمليا، الفرق عموما بين الوظائف و الإجراءات محدود جدا: يمكنك استدعاء وظيفة لإنجاز عمل ما ثم تتخطّى النتيجة (التي قد تكون رمز خطأ اختياري أو ما شابه) كما بإمكانك إستدعاء إجراء يمرّر نتيجته ضمن محدّداته (سيأتي الحديث أكثر عن المحدّدات بالإشارة reference parameters لاحقا في هذا الفصل).

ها هنا تعريفات لإجراء و نسختين من نفس الوظيفة، باستخدام صيغ مختلفة قليلا :

```
procedure Hello;
begin
  ShowMessage ('Hello world!');
end;

function Double (Value: Integer) : Integer;
begin
  Double := Value * 2;
end;

// or, as an alternative
function Double2 (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

استخدام *result* بدلا من اسم الوظيفة من أجل تخصيص قيمة الوظيفة المرتجعة أصبحت شائعة جدا، و تنحى لجعل التوليف أكثر مقروئية، حسب رأيي .

حالما يتم تعريف هذه الإجرائات، يمكنك إستدعائهم مرة أو أكثر. تستدعي الإجراء لجعله ينجز مهمته، و تستدعي الوظيفة لحساب القيمة :

```
procedure TForm1.Button1Click (Sender: TObject);
begin
  Hello;
end;

procedure TForm1.Button2Click (Sender: TObject);
var
  X, Y: Integer;
begin
  X := Double (StrToInt (Edit1.Text));
  Y := Double (X);
  ShowMessage (IntToStr (Y));
end;
```

ملاحظة :في الوقت الراهن لا تهتم كثيرا بصيغة الإجراءات أعلاه، و التي هي حقيقة مسارات methods.ببساطة قم بوضع زرّين على نافذة دلفي، لمسة مزدوجة فوقهما وقت التصميم، و ستقوم بيئة دلفي IDE بتوليد ما يناسب من توليف داعم :الآن و ببساطة عليك ملء الأسطر بين *begin* و*end*. لتجميع التوليف أعلاه تحتاج أيضا إلى إضافة خانة كتابة Edit للنافذة .

الآن يمكننا العودة إلى مفهوم تغليف التوليف الذي أشرت إليه سابقا. عندما تستدعي وظيفة *Double*، أنت لا تحتاج إلى معرفة الخوارزمية التي أُستخدمت لتنفيذها .إذا وجدت لاحقا طريقة أفضل لمضاعفة الأرقام، يمكنك بسهولة تغيير توليف الوظيفة، لكن التوليف الذي قام بالإستدعاء سيبقى ثابتا (بالرغم من أن التنفيذ سيكون أسرع!). نفس المفهوم يمكن تطبيقه على وظيفة *Hello*:يمكننا تعديل مخرجات البرنامج بتغيير التوليف داخل هذه الوظيفة، و بطريقة آلية سيتغير التأثير الذي يحدثه مسار Button2Click بدون أن نغيّر فيه :

```
procedure Hello;
begin
  MessageDlg ('Hello world!', mtInformation, [mbOK]);
end;
```

فائدة :عندما تستدعي إحدى وظائف أو إجراءات دلفي، أو أي مسار لمكوّنVCL ، يجب أن تتذكر عدد و نوع المحدّدات. محرّر دلفي يمكن أن يساعد باقتراحه لقائمة المحدّدات الخاصة بالوظيفة أو الإجراء بواسطة تلميحة محادية حالما تقوم بطباعة اسم الاجرائية و تفتح قوسا. هذه الخاصية تدعى Code Parameters و هي جزءا من تقنية .Code Insight

المحدّدات بالإشارة

تسمح لك إجرائيات باسكال بتمرير المحدّدات parameter بقيمتها by value و بالإشارة by reference.افتراضيا المحدّدات يتم تمريرها بالقيمة: يتم نسخ القيمة في الصفّ stack و تقوم الإجرائيات باستخدام و معالجة النسخة، و ليست القيمة الأصلية .

تمرير المحدّد بالإشارة يعني أن قيمته لايتم نسخها في الصفّ لدي الإجرائية) تجنبّ النسخ دائما يعني أن تنفيذ البرنامج يكون أسرع). بدلا من ذلك، البرنامج يشير إلى القيمة الأصلية، يحدث هذا أيضا في توليف الإجرائية. هذا يسمح للإجراء أو الوظيفة بأن تغيّر في قيمة المحدّد. المحدّد المُرّر بالإشارة يُعبّر عنه بالمصطلح *var* .

هذا الأسلوب موجود في معظم لغات البرمجة. هو ليس موجودا في س، و لكن تم ادخاله في س++، حيث تقوم باستعمال علامة) & تمرير بالإشارة). في فيجوال بيسك كل محّدد لا يكون بصفة *ByVal* يتم تمريره بالإشارة .

هنا مثال لتمرير محدّد بالإشارة باستخدام مصطلح *var* :

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

في هذه الحالة، المحدّد تم استخدامه لغرضين، لتمرير قيمة للإجرائية و لإسترجاع القيمة الجديدة للتوليف الذي قام بالاستدعاء. عندما تكتب :

```
var
  X: Integer;
begin
  X := 10;
  DoubleTheValue (X);
```

فإن قيمة المتغير X تغدو 20، لأن الإجرائية تتعامل مع إشارة لموقع الذاكرة الأصلي ل X ، مؤثّرة في قيمتها الأولى .

تمرير المحدّدات بالإشارة له مايبرّره فيما يتعلّق بالأنواع التراتبيةordinal ، و الجُمل strings بالطريقة التقليدية، و بالتسجيلات records الضخمة. في الواقع إن كائنات objects دلفي دائما يتم تمريرها بالقيمة، لأنها هي نفسها إشارة. لهذا السبب فإن تمرير الكائنات بإشارتها لامعنى له تقريبا (ما عدا بعض الحالات الخاصة جدا)، لأنها كما لوكانت "تمرير إشارة بالإشارة ".

جُمل strings دلفي الضخمة لها سلوك مختلف بعض الشيء: هي تتصرّف و كأنها إشارة، لكنك إذا قمت بتغيير واحدة من متغيرات الجمل التي تشير إلى نفس الجملة في الذاكرة، يتم نسخها قبل تحديثها. الجُمل الطويلة التي تمرر كمحدد بقيمة تتصرف و كأنها إشارة فقط من حيث استخدام الذاكرة و سرعة الشغيل. لكن إذا قمت بتعديل قيمة الجملة ، فإن القيمة الأصلية لاتتأثّر، بالمقابل، إذا مرّرت الجُملة الطويلة بالإشارة، يمكنك تغيير القيمة الأصلية .

أدخلت دلفي 3 نوعا جديدا من المحدّدات، وهي .out محدّد out ليس لديه قيمة ابتدائية و يستخدم فقط لترجيع قيمة. هذه المحدّدات يجب استخدامها فقط لإجرائيات و وظائف COM ؛ عموما، من الأفضل التشبّث بمحدّدات var الأكثر فعالية .محددات out تتصّرف مثل محدّدات var بإستثناء عندما لا يوجد لديها قيمة ابتدائية .

محدّدات الثوابت

كبديل للمحددات بالإشارة، يمكنك استعمال محدد .const بما أنّه لايمكنك تخصيص قيمة لمحدد ثابت داخل الإجرائية، يمكن للمجمّع تحسين كفاءة تمرير المحدّد. المجمّع يمكن أن يختار أسلوبا شبيها بالمحددات بالإشارة (أو الإشارة لثابت const reference حسب مصطلحات س++)، لكن التصرّف سيبقى شبيها بالمحددات بالقيمة، لأن القيمة الأصلية لن تتأثر بالإجرائيات .

في الواقع، إذا حاولت تجميع التوليف (السخيف) التالي، ستقوم دلفي باصدار خطأ :

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;       // compiler error
  Result := Value;
end;
```

محدّدات المصفوفة المفتوحة

عكس لغة س، وظيفة أو إجراء دلفي لديهما دائما عددا ثابتا من المحدّدات, إلا أنه توجد طريقة لتمرير عددا غير ثابت من المحدّدات إلى الإجرائية بإستخدام المصفوفة المفتوحة .open array

التعريف الأساسي لمحدد مصفوفة مفتوحة open array parameter هو مصفوفة مفتوحة ذات نوع. هذا يعني انك تشير إلى نوع المحدّد لكنك لاتعرف كم عنصر من هذا النوع سيكون لدى المصفوفة. هنا مثال لمثل هذا التعريف :

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

بإستخدام High(A) يمكننا الحصول على حجم المصفوفة، لاحظ أيضا استخدام قيمة الترجيع في الوظيفة، Result، لتخزين قيم مؤقتة. يمكنك استدعاء هذه الوظيفة بأن تمرر إليها مصفوفة من التعبيرات ذات نوع صحيح .Integer

```
X := Sum ([10, Y, 27*I]);
```

إذا كان لديك مصفوفة من أي حجم ذات نوع صحيح، تستطيع تمريرها مباشرة لإجرائية تتطلب محدد بمصفوفة مفتوحة، أو بدلا من ذلك، يمكنك استدعاء وظيفة Slice لتمرير جزء فقط من المصفوفة (كما هو مشار اليه في ثاني محدد في الوظيفة). ها هنا مثال، حيث مصفوفة كاملة تمّ تمريرها كمحدّد :

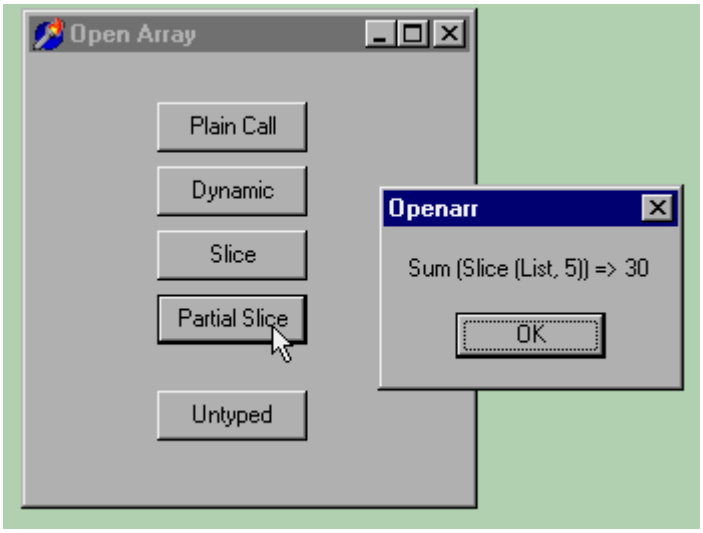
```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);
```

إذا أردت تمرير فقط جزء من المصفوفة إلى وظيفةSum ، ببساطة قم باستدعائها بالطريقة التالية :

```
X := Sum (Slice (List, 5));
```

تستطيع أن تجد كل أجزاء التوليف الذي تم عرضه في هذا القسم في مثال OpenArr (أنظر الشكل 6.1، لاحقا، بالنسبة للنموذج).

الشكل 6.1: مثال OpenArr عندما يتم الضغط على زرّ Partial Slice



المصفوفات المفتوحة النوعية في دلفي 4 متوافقة تماما مع المصفوفات الحيّة dynamic) تم تقديمها في دلفي 4 و مغطّاة في الفصل 8). المصفوفات الحيوية تستخدم نفس الصيغة في المصفوفات المفتوحة، مع اختلاف انه يمكنك استخدام التركيب *array of Integer* لتعريف متغيّر، و ليس فقط لتمرير محدّد .

محدّدات مصفوفة مفتوحة نوع متباين

بجانب هذه المصفوفات المفتوحة النوعية، تسمح لك دلفي بتحديد مصفوفات مفتوحة نوع متباين type-variantأو بلا نوع. هذا النوع الخاص من المصفوفات لديه عدد غير محدود من القيم، و التي يمكن الاستفادة منها لتمرير المحدّدات .

تقنيا، بنية مصفوفة الثوابت تسمح لك بتمرير مصفوفة بعدد غير محدود من العناصر من أنواع مختلفة إلى إجرائية دفعة واحدة. مثال ذلك، ها هنا تعريف لوظيفة Format (سنرى كيف نستخدم هذه الوظيفة في **الفصل 7**، عند الحديث عن الجمل):

```
function Format (const Format: string;
  const Args: array of const): string;
```

المحدّد الثاني هو مصفوفة مفتوحة، تستقبل عددا غير محدود من القيم. في الواقع، يمكنك استدعاء هذه الوظيفة بالطرق التالية :

```
N := 20;
S := 'Total:';
Label1.Caption := Format ('Total: %d', [N]);
Label2.Caption := Format ('Int: %d, Float: %f', [N, 12.4]);
Label3.Caption := Format ('%s %d', [S, N * 2]);
```

لاحظ أنه بإمكانك تمرير المحدّد كقيمة ثابت، أو قيمة متغير، أو كتعبير. تعريف وظيفة من هذا النوع أمر سهل، لكن كيف تقوم بتوليفه؟ كيف تتعرف على نوع المحدّدات؟ ان قيم محددات مصفوفة مفتوحة نوع متباين هي متوافقة مع عناصر نوع *TVarRec* :

ملاحظة :لا تخلط بين تسجيلة *TVarRec* و تسجيلة *TVarData* المستخدمة من قبل نوع Variantنفسه. هاتان البنيتان تخدمان أغراضا مختلفة و ليستا متوافقتين. بالرغم من أن قائمة الأنواع المحتملة مختلفة، لأن *TVarRec* يمكن ان تضم أنواع بيانات دلفي، بينما *TVarData* يمكن أن تحوي أنواع بيانات .OLE

تسجيلة *TVarRec* لها البُنية التالية :

```
type
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
  end;
```

كل تسجيلة محتملة لديها حقل نوع *VTipe* ، بالرغم انه ليس سهلا رؤيته من المرة الأولى لأن تعريفه يتم مرة واحدة فقط .

بواسطة هذه المعلومات يمكننا فعلا كتابة وظيفة قادرة على التعامل مع أنواع بيانات مختفلة. في مثال وظيفة *SumAll* ، أريد أن أكون قادرا على جمع قيم من أنواع مختلفة، تحويل الجمل إلى أعداد صحيحة، الحروف إلى ما يقابلها من قيمة ترتيبية، و إضافة 1 للقيم البولية الموجبة. التوليف يعتمد على تعليمةcase ، و يعدّ سهلا، بالرغم من أنه علينا التعامل مع المؤشرات pointersأكثر من مرّة :

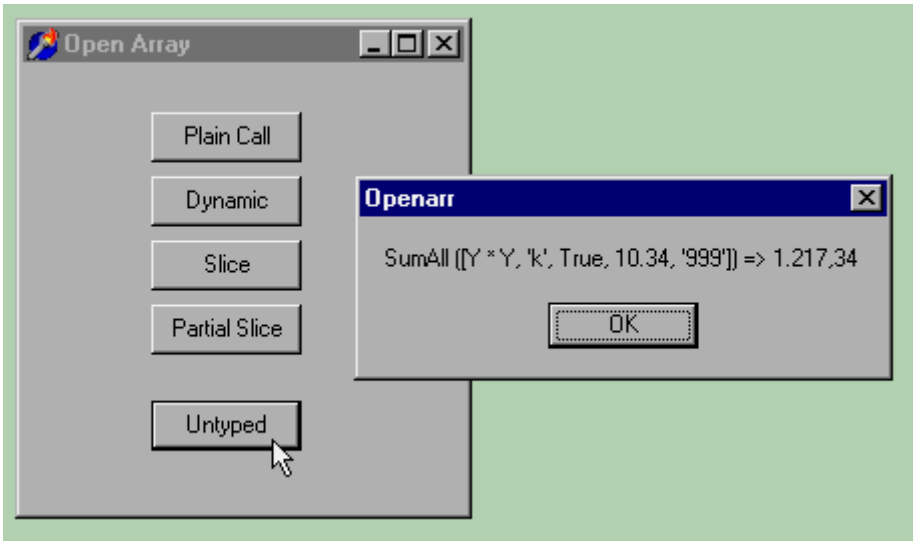
```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger: Result :=
        Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtChar:
        Result := Result + Ord (Args [I].VChar);
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtString, vtAnsiString:
        Result := Result + StrToIntDef ((Args [I].VString^), 0);
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
end;
```

لقد اضفت هذا التوليف إلى مثالOpenArr ، الذي يستدعي وظيفة *SumAll* عند يتم الضغط على زرّ معيّن .

```
procedure TForm1.Button4Click(Sender: TObject);
var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34, '99999']);
  ShowMessage (Format (
    'SumAll ([Y*Y, ''k'', True, 10.34, ''99999'']) =     >  %n',
[X]));
end;
```

يمكن رؤية نتاج هذا الاستدعاء، و النموذج بمثالOpenArr ، في الشكل 6.2 .

الشكل 6.2: النموذج في مثالOpenArr ، مع مربع رسالة تُعرض عند الضغط على زرّ Untyped.



طرق الإستدعاء في دلفي

أدخلت نسخة 32-بت في دلفي مفهوما جديدا لتمرير المحدّدات، تعرف باسم :fastcall فحيثما أمكن، و حتى لثلاث محدّدات يمكن تمريرها في مسجّلات registers المعالج، جاعلة من استدعاء الوظيفة أسرع بكثير. طريقة الاستدعاء السريع (fast calling convention تستخدم افتراضيا في دلفي 3) يشار لها بمصطلح register.

المشكلة أنها الطريقة الإفتراضية، و الوظائف التي تستخدمها ليست متوافقة مع ويندوز: وظائف Win32يجب تعريفها باستخدام طريقة استدعاءsdtcall ، و هي مزيج من الطريقة الأصلية للإستداعاء في باسكال لوظائف Win16 و طريقة استدعاء cdecl في لغة س .

لا يوجد عموما سبب يمنع استعمال طريقة الإستدعاء السريع، إلا إذا كنت تقوم باستدعاءات ويندوز خارجية external أو تقوم بتحديد وظائف callback. سوف نرى مثالا عن استخدام طريقة stdcall قبل نهاية هذا الفصل, يمكنك أن تجد ملخّصا لطرق استدعاءات دلفي تحت موضوع Calling conventions في ملف مساعدة دلفي .

ماهو المسار؟

إذا سبق لك بالفعل العمل بدلفي أو قرأت أدّلة التشغيل، فمن المحتمل انك سمعت بمصطلح methodمسار. المسار هو نوع خاص من الوظائف أو الإجرائيات ذات علاقة بنوع بيانات، الطبقة class. في دلفي، كل مرة نتناول حدثا، نحتاج لتعريف مسار، أو بصفة عامة إجراء. على أية حال، مصطلح مسار يستخدم للإشارة إلى الوظائف و الإجراءات ذات العلاقة بالطبقة class.

سبق لنا أن رأينا بالفعل عددا من المسارات في الأمثلة الواردة في هذا الفصل و في ما سبقه. في ما يلي مسار فارغ أضيف آليا بواسطة دلفي للتوليف المصدري لنموذج :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {here goes your code}
end;
```

التعريف المسبق

عندما تحتاج إلى استخدام معرّف identifier من أي نوع، يجب أن يكون المجمّع قد رأى بالفعل تحديدا ما ليعلم إلى ماذا يشير هذا المعرّف. لهذا السبب، فأنت عادة ما تقدّم تعريفا كاملا قبل استخدام أية إجرائية. على أية حال، توجد حالات لايمكنك فيها ذلك. إذا فرضنا أن الإجراء A يستدعي الإجراءB ، و الإجراء B يستدعي الإجراءA ، عندما تبدأ بكتابة التوليف، فستحتاج إلى مناداة إجرائية لا يزال المجمّع لم ير تعريفها .

إذا أردت تعريف وجود إجراء أو وظيفة بإسم معيّن و محدّدات معطاة، من غير تقديم توليفها الفعلي، يمكنك كتابة الإجراء أو الوظيفة متبوعة بالكلمة المفتاحية forward :

```
procedure Hello; forward;
```

لاحقا، يجب أن يقدم التوليف تعريفا كاملا للإجراء، لكن هذا يمكن استدعاؤه حتى قبل أن يتم تعريفه بالكامل. فيما يلي مثال ساذج، فقط لإعطائك فكرة :

```
procedure DoubleHello; forward;

procedure Hello;
begin
  if MessageDlg ('Do you want a double message?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    DoubleHello
  else
    ShowMessage ('Hello');
end;

procedure DoubleHello;
begin
  Hello;
  Hello;
end;
```

هذه الطريقة تسمح لك بكتابة تواتر recursion متبادل *DoubleHello* :تنادي Hello ، لكن Helloربما تنادي *DoubleHello* ، أيضا. طبعا لابد من وجود شرط لإيقاف التواتر، لتجنب فوران التكدّس stack overflow .

بالرغم من أن تعريف الإجراء المسبق forward procedure ليس شائعا في دلفي، توجد حالة مشابهة و التي تتكر أكثر. عندما تقوم بتعريف إجرائية أو وظيفة في جزء الواجهة interface في الوحدة) unit المزيد عن الوحدات في الفصل القادم)، يتم إعتباره تعريفا مسبقا، حتى إذا كان مصطلح forward غير ظاهر. فعليا لايمكنك أن تكتب جسم الاجرائية في جزء الواجهة. في نفس الوقت، يجب أن تقوم بتقديم التنفيذ الفعلي لكل إجرائية قمت بتعريفها، وذلك في نفس الوحدة .

نفس الشيء ينطبق على تعريف المسار methos داخل نوع طبقة class و الذي يتم توليده آليا بواسطة دلفي (كما يحدث عند اضافة حدث لنموذج أو أحد مكوناته). مناولات الحدث المعرّفة داخل طبقة *TForm* هي تعريفات مسبقة: حيث سيتم تقديم التوليف في جزء التنفيذ implementaionمن الوحدة. فيما يلي مقطع من توليف مصدري لمثال سابق، مع تعريف لمسار *Button1Click* :

```
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;
```

الأنواع الإجرائية

ميزة فريدة أخرى في اوبجكت باسكال وهي وجود الأنواع الإجرائية .procedural types في الواقع يعّد هذا موضوعا برمجيا متقدما، و قلّة من مبرمجي دلفي سوف يستعملونه باستمرار. عموما، ما دمنا سوف نناقش الموضوعات ذات العلاقة في الفصول القادمة) خاصة، مؤشرات المسار، التقنية المستخدمة بكثافة في دلفي)، فإن الأمر يستحق أن نلقي بنظرة سريعة على هذا الموضوع هنا. إذا كنت مبرمجا مبتدئا، يمكنك تخطّي هذا القسم مؤقتا، و أن تعود لاحقا عندما تشعر بأنك مستعدّ لذلك .

في باسكال، يوجد مفهوم النوع الإجرائي) procedural type و الذي يشبه مفهوم مؤشر الوظيفة function pointer في لغة س). تعريف النوع الإجرائي يشير إلى قائمة من المحدّدات و نوع الترجيع في حالة الوظيفة. مثلا، يمكنك تعريف نوع إجراء مع محدد برقم صحيح يتم تمريره بالإشارة مثل :

```
type
  IntProc = procedure (var Num: Integer);
```

النوع الإجرائي هذا متوافق مع أي إجرائية تملك تماما نفس المحددات (أو نفس توقيع الوظيفة function signature، بتعبير لغة س). هنا مثال لإجرائية متوافقة .

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

ملاحظة :في نسخة 16-بت من دلفي، يجب أن يتم تعريف الإجرائيات بإستخدام توجيه far من أجل إستعمالها كقيمة فعلية للنوع الإجرائي .

الأنواع الإجرائية يمكن استخدامها لغرضين مختلفين: يمكنك تعريف متغيرات من نوع إجرائي أو تمرير نوع إجرائي - مؤشّر وظيفة- كمحددات إلى إجرائية أخرى. بوجود النوع السابق و تعريفات الإجراء، يمكنك كتابة هذا التوليف :

```
var
  IP: IntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

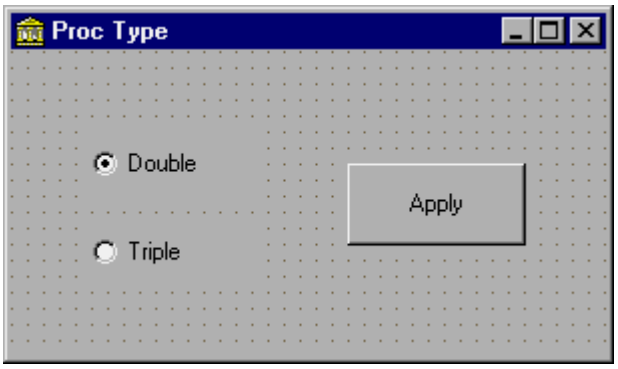
هذا التوليف له نفس التأثير الذي للنسخة الأقصر التالية :

```
var
  X: Integer;
begin
  X := 5;
  DoubleTheValue (X);
end;
```

واضح أن النسخة الأولى أكثر تعقيدا، لذا لماذا علينا استعمالها؟ في بعض الحالات، القدرة على تقرير أية وظيفة يمكن استدعاؤها و أن يتم أستدعاؤها فعليا لاحقا، هذه الامكانية قد تكون مفيدة. يمكن بناء مثال متشعب يعرض هذا التوجه. عموما، أنا أفضّل أن أجعلك تستكشف مثالا بسيط بما يكفي، اسمه ProcType . هذا المثال أكثر تشعبا من كل ما سبق أن رأيناه حتى الآن، لجعل الأمور أكثر واقعية .

ببساطة قم بانشاء مشروع جديد و قم بوضع زرّي خيارradio buttons ، زرّ ضغط، و ملصقين lables على النافذة. كما هو موضّح في الشكل 6.3. هذا المثال مبني على إجرائين. الإجراء الأول يتم استخدامه لمضاعفة قيمة المحدّد. هذا الإجراء شبيه بذلك الذي قمت بعرضه في هذا القسم. الإجراء الثاني يتم استخدامه لزيادة قيمة المحدد بثلاثة أضعاف، لهذا فإن اسمه TripleTheValue:

الشكل 6.3: نموذج مثال ProcType.




```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
  ShowMessage ('Value tripled: ' + IntToStr (Value));
end;
```

الإجراءان يعرضان ماذا يجري فيهما، لكي يعلمانا بأنهما قد استدعيا. هذه ميزة تعرّف بسيطة يمكنك استخدامها لاختبار اذا ما تم تنفيذ جزء معين من التوليف أو متى تم ذلك، بدلا من اضافة نقاط اعاقة breakpoint فيهما .

في كلّ مرة يقوم فيها المستخدم بالضغط على زرّApply ، يتم تنفيذ أحد الإجرائين، حسب حالة خاناتي الخيار. في الواقع، عندما يكون لديك إثنان من خانات الخيار في النموذج، واحدة منهما فقط يمكن إختيارها في نفس الوقت. يمكن لهذا التوليف ان يُنفّذ باختبار قيمة خانتي الخيار داخل التوليف الخاص بالحدث *OnClick* لزرّ .Apply لكن و من أجل استعراض كيفية استخدام الأنواع الإجرائية، قمت بدلا من ذلك باتّباع توجّه أطول لكن مثير للإهتمام. كلّ مرّة يضغط فيها المستخدم على واحدة من خانات الخيار، أحد الإجرائين يتم تخزينه في متغيّر :

```
procedure TForm1.DoubleRadioButtonClick(Sender: TObject);
begin
  IP := DoubleTheValue;
end;
```

عندما يضغط المستخدم على الزرّ، يتم تنفيذ الإجرائية التي يتم تخزينها :

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  IP (X);
end;
```

من أجل السماح لثلاث وظائف مختلفة بتناول المتغيرين IP و X ، نحتاج لجعلهما مرئيين على مستوى النموذج form بالكامل؛ لايمكن تعريفهما محليا) localy داخل أحد المسارات). الحل لهذه المشكلة هي وضع المتغيرين داخل تعريف النموذج :

```
type
  TForm1 = class(TForm)
    ...
  private
    { Private declarations }
    IP: IntProc;
    X: Integer;
  end;
```

سوف نرى تماما ماذا يعني هذا في الفصل التالي، لكن حاليا، يتطلب منك الأمر تعديل التوليف الذي قامت دلفي بإعداده لنوع الطبقة كما هو موضّح أعلاه. و قم باضافة النوع الإجرائي الذي قمت بعرضه سابقا. من أجل تمهيد هذين المتغيرين بقيم مناسبة، يمكننا مناولة حدث*OnCreate* الخاص بالنموذج (اختر هذا الحدث في معاين الكائنات Object Inspector بعد تفعيل النموذج، أو قم ببساطة بضغط مزدوج على النموذج). اقترح أن تقوم بمراجعة التوليف لدراسة تفاصيله في المثال .

يمكنك مشاهدة مثال عملي لإستخدام الأنواع الإجرائية في الفصل 9، في قسم *وظيفة Callback في ويندوز*.

التحميل المضاف لوظيفة

فكرة التحميل المضاف overloading بسيطة: يسمح لك المجمّع compiler بتحديد وظيفتين أو إجرائين يحملان نفس الإسم، بشرط أن تختلف المحدّدات. و باختبار هذه المحددات، يمكن للمجمّ استنتاج أية نسخة من الإجرائيتين تريد استدعاؤها.

راجع هذه السلسة من الوظائف المستخرجة من وحدة Math في مكتبةVCL :

```
function Min (A,B: Integer): Integer; overload;
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;
```

عندما تنادي(Min (10,20 ، يستنتج المجمّع ببساطة بأنك تقصد استدعاء الوظيفة الأولى من المجموعة، لذا فالقيمة المرتجعة ستكون رقما صحيحا.

القواعد الأساسية اثنان:

- كل نسخة من الإجرائية يجب أن تكون متبوعة بالكلمة المفتاحية .overload
- الاختلاف يجب أن يكون في عدد أو نوع المحدّدات، أو في كلاهما. و ليس في نوع المرتجع، الذي لا يمكن استخدامه للتفريق بين الاجرائيتين .

فيما يلي ثلاث نسخ بحمل مضاف لإجراء ShowMsg قمت باضافتها لمثال OverDef (التطبيق الذي يستعرض الحمل المضاف و المحددات الافتراضية)

```
procedure ShowMsg (str: string); overload;
begin
  MessageDlg (str, mtInformation, [mbOK], 0);
end;

procedure ShowMsg (FormatStr: string;
  Params: array of const); overload;
begin
  MessageDlg (Format (FormatStr, Params),
    mtInformation, [mbOK], 0);
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
  ShowMsg (IntToStr (I) + ' ' + Str);
end;
```

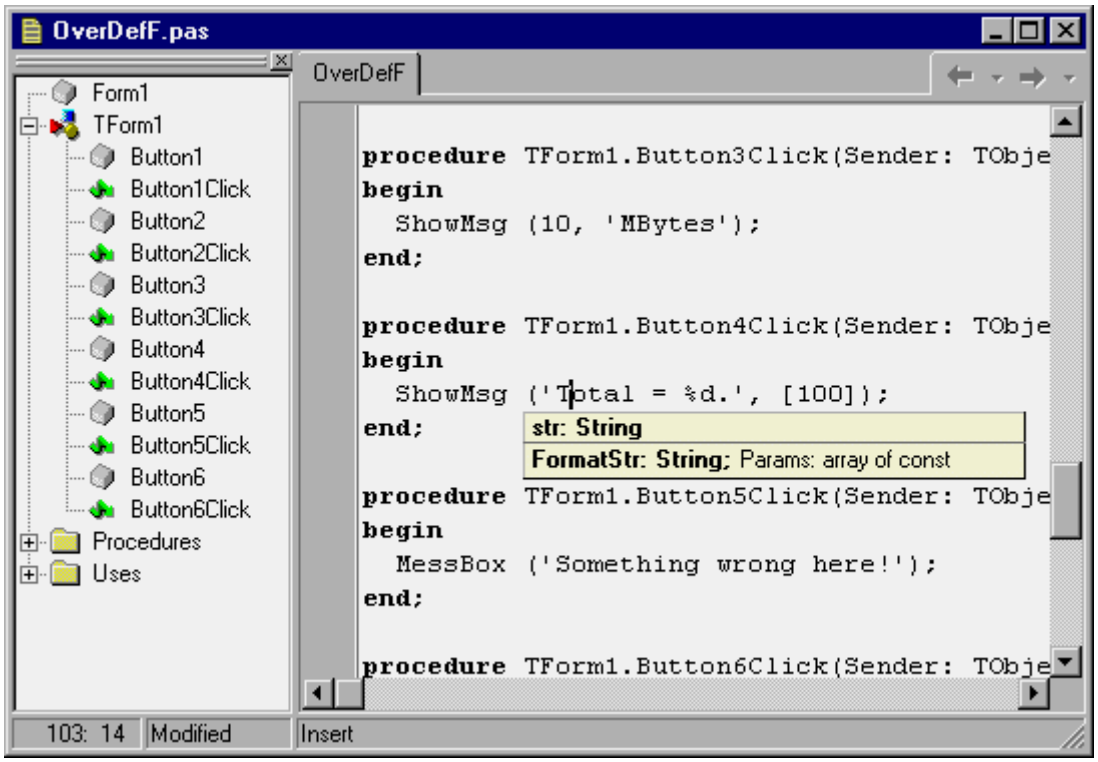
الوظائف الثلاثة تعرض نافذة رسالة مع جملة، بعد صياغة متكررة للجملة بعدة طرق .ها هنا النداءات الثلاث للبرنامج:

```
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

ما ادهشني ايجابيا هو ان تقنية Code Parameters محددات التوليف في دلفي تعمل بصورة رائعة مع محددات الحمل المضاف. فمع بداية فتح قوس في طباعتك بعد اسم الإجرائية، يتم عرض كل البدائل المتوفرة. و مع ادخالك للمحدد، تقوم دلفي باختبار نوعه لتقرير أيا من البدائل

لايزال متوفرا. في الشكل 6.4 يمكنك رؤية هذا بعد البدء بكتابة ثابت جملة تعرض دلفي نسخة متوافقة واحدة (تستثني نسخة إجراء ShowMsg الذي لها نوع صحيح كأول محدد).

الشكل 6.4: البدائل المتعددة التي اقترحتها تقنية Code Parameters لإجرائيات الحمل المضاف، مفروزة بحسب المحددات المتوفرة فعلا .



حقيقة أن كل نسخة من إجرائية حمل مضاف overloaded يجب أن تكون معلّمة بوضوح؛ هذا يعني ضمنا أنك لاتستطيع تحميل إجرائية موجودة في نفس الوحدة unit وليست معلّمة بمصطلح overload. (رسالة الخطأ التي تظهر لك عندما تحاول ذلك هي: تعريف سابق ل< اسم الاجرائية >ليست معلمّة بتوجيه 'overload'.). عموما ، يمكنك اجراء تحميل اضافي لإجرائية تكون قد سبق تعريفها في وحدة مختلفة. هذا لأجل التوافقية مع النسخ السابقة من دلفي، و التي تسمح لعدة وحدات أن تعيد استخدام نفس اسم الإجرائية. لاحظ، على أية حال، بأن هذه الحالة الخاصة ليست ميزة اضافية للتحميل المضاف، لكنها اشارة الى المشكلات التي قد تواجهها.

مثلا، يمكنك اضافة التوليف التالي للوحدة:

```
procedure MessageDlg (str: string); overload;
begin
  Dialogs.MessageDlg (str, mtInformation, [mbOK], 0);
end;
```

هذا التوليف لايقوم فعلا بحمل اضافي لإجرائية MessageDlg الأصلية. في الواقع إذا كتبت:

```
MessageDlg ('Hello');
```

سوف تتحصل على رسالة خطأ لطيفة تشير إلى غياب بعض المحددات. الطريقة الوحيدة لإستدعاء نسخة محلية بدلا من أخرى تابعة لمكتبة VCL هي في أن تشير صراحة إلى الوحدة المحليّة، الأمر الذي يخدش فكرة التحميل المضاف:

```
OverDefF.MessageDlg ('Hello');
```

المحدّدات الافتراضية

خاصّية جديدة أخرى في دلفي 4 و هي أنك تستطيع أن تعطي قيمة افتراضية default لمحدد إجراء أو وظيفة، و يمكنك استدعاء هذه الوظيفة رفق المحدد أو بدونه. دعني أعرض مثالا. يمكننا تحديد التغليف التالي لمسار method MessageBox الخاص بالكائن العام Application، و الذي يستخدم أنواع PChars بدلا من جملstrings ، و سوف نوفّر محددين افتراضيين:

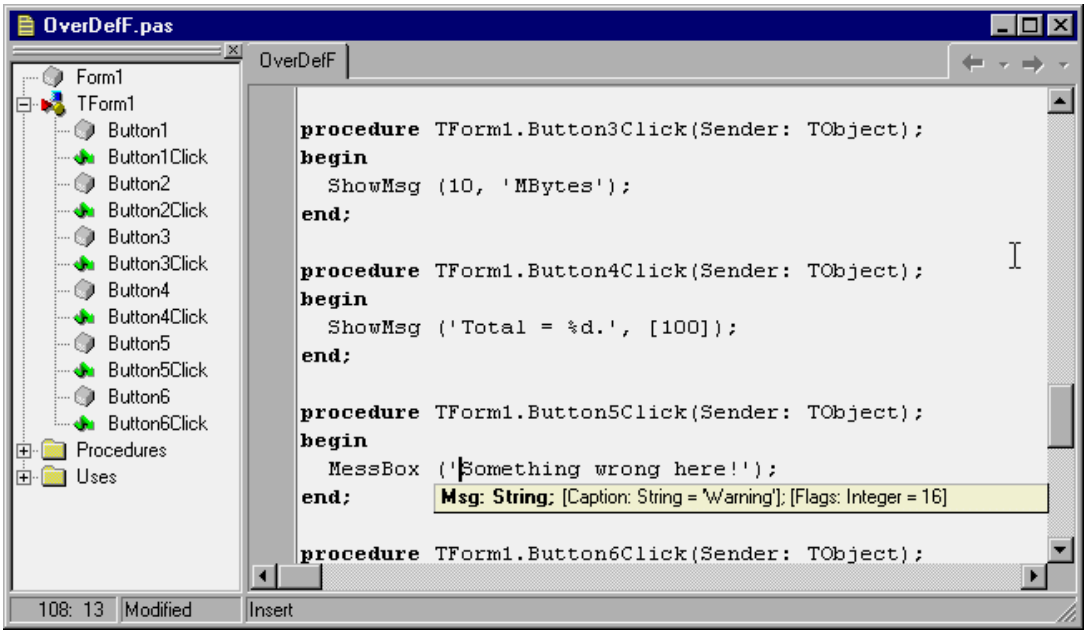
```
procedure MessBox (Msg: string;
  Caption: string = 'Warning';
  Flags: LongInt = mb_OK or mb_IconHand);
begin
  Application.MessageBox (PChar (Msg),
    PChar (Caption), Flags);
end;
```

بهذا التعريف، يمكننا استدعاء الاجرائية بأي من الطرق التالية:

```
MessBox ('Something wrong here!');
MessBox ('Something wrong here!', 'Attention');
MessBox ('Hello', 'Message', mb_OK);
```

في الشكل 6.5 يمكنك رؤية محددات التوليف Code Parameters لدلفي يستخدم نمط مختلف و مناسب ليشير الى المحددات التي لها قيم افتراضية، بحيث تستطيع بسهولة تبيان أي المحددات التي يمكنك استبعادها.

الشكل 6.5: محددات التوليف لدلفي تشير بين أقواس مربعة الى المحددات التي لها قيم افتراضية؛ و التي يمكنك استبعادها في هذا الاستدعاء .



لاحظ ان دلفي لا تقوم بتوليد أي توليف خاص لدعم المحددات الافتراضية؛ كما لا تنشىء نُسخا متعدّدة من الإجرائية. ببساطة، المحددات المستبعدة يتم اضافتها من قبل المجمّع إلى التوليف الذي قام بالإستدعاء.

هناك قيد واحد مهم يؤثّر على استخدام المحدّدت الإفتراضية: لا يمكنك تخطّي المحدّدات. مثلا، لا تستطيع تمرير المحدّد الثالث للوظيفة بعد استبعادك للمحدّد الثاني:

```
MessBox ('Hello', mb_OK); // error
```

هذه هي القاعدة الرئيسية للمحددات الافتراضية: حين الاستدعاء، يمكنك فقط استبعاد المحددات بدءا من المحدد الأخير. بعبارة أخرى، إذا استبعدت محدّدا يجب أيضا استبعاد ما يليه.

هناك أيضا بعض القواعد الأخرى للمحددات الافتراضية:

- المحددات ذات القيمة الافتراضية يجب أن تكون في آخر قائمة المحددات .
- القيم الإفتراضية يجب تكون ثوابت .constants واضح، ان هذا يقيّد الأنواع التي تستطيع استعمالها مع المحددات الافتراضية. مثلا المصفوفة المفتوحة أو نوع واجهة interface type لايمكن أن يكون لها قيمة افتراضية غير لاشيءnil ؛ التسجيلات records لا يمكن استخدامها اطلاقا .
- المحددات الافتراضية يجب أن يتم تمريرها بقيمة أو كثابت. محدد (var) بالاشارة reference لايمكن أن يكون لها قيمة افتراضية .

استخدام محددات افتراضية و حمل مضاف في نفس الوقت يمكن ان يسبب عدّة مشاكل، بسبب امكانية تعارض الميزتين. مثال ذلك، إذا ما اضفت للمثال السابق النسخة الجديدة التالية من إجرائيةShowMsg :

```
procedure ShowMsg (Str: string; I: Integer = 0); overload;
begin
  MessageDlg (Str + ': ' + IntToStr (I),
    mtInformation, [mbOK], 0);
end;
```

عندها المجمّع لن يتذمر إنّه تعريف صحيح. لكن الإستدعاء:

```
ShowMsg ('Hello');
```

يتم اعتباره من قبل المجمّع على أنه استدعاء حمل مضاف ملتبس *Ambiguous overloaded call to 'ShowMsg'*. لاحظ أن هذا الخطأ يظهر في سطر التوليف الذي تم تحويله بطريقة صحيحة قبل تعريف الحمل المضاف الجديد .عمليا، ليس لدينا أية طريقة لاستدعاء أجراء ShowMsgبمحدد جملة واحد، حيث أن المحوّل لا يعرف إذا كنا نريد استدعاء النسخة التي بمحدد جملة واحد فقط أو تلك التي بمحدد جملة و محدد رقم صحيح ذو قيمة افتراضية. عندما يكون لديه مثل هذا الشكّ، المحوّل يتوقّف و يسأل المبرمج أن يبيّن قصده بوضوح أكثر.

ملخّص

كتابة الإجراءات و الوظائف هو العنصر الأساسي في البرمجة، بالرغم من أنك في دلفي سوف تتجه لكتابة المسارات -- methods إجراءات و وظائف مرتبطة بالطبقات classes و الكائنات objects.

بدلا من التنقّل إلى خصائص الإتجاه الكائنيobject-oriented ، الفصول القليلة القادمة تقدّم لك بعض التفاصيل عن عناصر أخرى في برمجة باسكال، مبتدئين بالجُمل strings .

الفصل التالي :مناولة الجُمل

الفصل 7
مناولة الجُمل

ماركو كانتو:
أساسي باسكال

مناولة الجُمل strings في دلفي أمر بسيط، لكن وراء الكواليس؛ الحالة معقدة بعض الشيء. لباسكال طريقتها التقليدية لمناولة الجُمل، ويندوز لها طريقتها الخاصة، المستمدة من لغة س. نسخ دلفي 32-بت تضمنت نوع بيانات قوي لجمل طويلة، و التي تشكل النوع الافتراضي لنوع جملة string في دلفي.

أنواع الجُمل

في تربو باسكال و في دلفي 16-بت من بورلاند، نجد أن النمط العام لنوع جملة هو تتابع من الأحرف مع بايت (حرف) في بدايتها، بشير إلى الحجم الحالي للجملة. و لأن الحجم يعبّر عنه ببايت وحيد، فليس بإمكانه أن يتعدّى 255 حرفا، و هو قيمة منخفضة جدا تخلق عدة مشاكل عند مناولة الجمل. كلّ جملة تُحدّد بحجم ثابت (افتراضيا تكون بحدها الأقصى، 255)، مع انك تستطيع أن تعرّف جملا أقصر للحفاظ على مساحة الذاكرة.

نوع الجملة يشبه نوع مصفوفة. في الواقع، ان الجملة هي تقريبا مصفوفة من أحرف. ما يبيّن هذا؛ حقيقة أنه يمكنك الوصول إلى حرف معيّن في الجملة باستخدام تركيبة.[]

لتجاوز قيود جمل باسكال التقليدية، قامت نسخ 32-بت من دلفي بدعم الجمل الطويلة .يوجد في الواقع ثلاث أنواع جمل:

- نوع ShortString جملة قصيرة و يتوافق مع جمل باسكال التقليدية، كما تم وصفه سابقا، هذه الجمل محدودة ب 255 حرف و تتماشى مع الجمل في نسخة 16-بت من دلفي. كل جزء من جملة قصيرة هي من نوع ANSIChar النوع القياسي للأحرف .(
- نوع ANSIString و يتطابق مع جديد الجمل الطويلة متغيرة الطول. هذه الجمل يتم تخصيصها حيويا، هي معدودة الاشارة reference counted ، و تستخدم تقنية copy-on-write نسخ عند الكتابة. حجم هذه الجمل غير محدود تقريبا (يمكنها التخزين لغاية 2 بليون حرف!). وهي أيضا مبينة على نوع .ANSIChar
- نوع WideString جملة عريضة وهي تشبه نوع ANSIString لكنها مبنية على نوع . WideChar لتخزين أحرف .Unicode

استخدام الجمل الطويلة

اذا استخدمت ببساطة نوع جملة طويلة، فإنك ستتحصّل إما على جمل قصيرة أو جملANSI ، و ذلك حسب قيمة التوجيه $H للمجمّع. قيمة) +$H و هي القيمة الافتراضية) تعني جمل طويلة (نوعANSIString) ، و هو المستخدم من قبل مكوّنات دلفي.

جمل دلفي الطويلة تعتمد على آلية تعداد الإشارةreference counting ، و التي تقوم بتتبّع مقدار المتغيرات نوع جملة التي تشير إلى نفس الجملة في الذاكرة. تعداد الإشارة هذا يُستخدم أيضا لتحرير الذاكرة التي تحتلّها جملة يكون قد توقّف استعمالها، أي عندما يبلغ تعداد الإشارة صفرا.

إذا أردت زيادة حجم جملة في الذاكرة لكن المنطقة المجاورة من الذاكرة تكون محجوزة من قبل شيء آخر، عندها لا يمكن للجملة أن تتوسع في نفس منطقة الذاكرة، بل يجب أخذ نسخة كاملة من الجملة و نقلها لمكان آخر في الذاكرة. عندما تحدث مثل هذه الحالة، فإن وقت التشغيل لدلفي يدعم اعادة توطين الجملة من أجلك و ذلك بأسلوب شفاف تماما و غير محسوس. أنت فقط تقوم بتحديد السعة القصوى للجملة بواسطة إجراءSetLength ، و سيتم تخصيص المقدار المطلوب من الذاكرة بكفاءة.

```
SetLength (String1, 200);
```

إجراء SetLength يقوم بطلب ذاكرة، و ليس بعملية تخصيص فعلية لذاكرة. أنه يحتفظ بمساحة الذاكرة المطلوبة لإستعمالها لاحقا، بدون أن يقوم باستعمالها فعليا. هذه التقنية تعتمد على خاصية في أنظمة تشغيل ويندوز، و تستخدمها دلفي لجميع تخصيصات الذاكرة الحيوية. مثلا عندما أنت تقوم بطلب مصفوفة ضخمة جدا، يتم الحجز في الذاكرة المطلوبة و لكن لا يتم تخصيصها.

نادرا ما تدعو الحاجة إلى تحديد طول لجملة. الحالة الوحيدة التي يجب فيها أن تقوم بتخصيص ذاكرة لجملة طويلة باستخدام SetLenth هي عندما يتطلب الأمر منك تمرير جملة كمحدد الى وظيفة API بعد تلبيس نوع مناسب)، كما سأقوم بعرضه بعد قليل.

النظر إلى الجُمل في الذاكرة

لمساعدتك على فهم أفضل لتفاصيل إدارة الذاكرة للجُمل، قمت بكتابة مثال StrRef البسيط. في هذا البرنامج قمت بتعريف جامع global لجملتين Str1 :و .Str2 عندما يتم الضغط على أوّل الزرّين، يقوم البرنامج بتخصيص ثابت جملة لأوّل المتغيرين ثم بعدها يقوم بتخصيص المتغير الثاني بالأول:

```
Str1 := 'Hello';
Str2 := Str1;
```

بجانب التعامل مع الجمل، يقوم البرنامج بعرض حالتها الداخلية في مربّع قائمةlistbox ، مستعملا وظيفة StringStatus التالية:

```
function StringStatus (const Str: string): string;
begin
  Result := 'Address: ' + IntToStr (Integer (Str)) +
    ', Length: ' + IntToStr (Length (Str)) +
    ', References: ' + IntToStr (PInteger (Integer (Str) - 8)^) +
    ', Value: ' + Str;
end;
```

أمر حيوي في وظيفة StringStatus أن يتم تمرير محدد الجملة كمحدد ثابت. ان تمرير هذا المحدد بواسطة النسخ سينتج عنه آثارا جانبية لوجود إشارة أخرى اضافية للجملة في نفس وقت تنفيذ الوظيفة. بالمقابل، ان تمرير المحدد بطريق الإشارة (var) أو ثابت (const) لا يسبب في خلق إشارة إضافية للجملة. لقد استعملت في هذه الحالة محددconst ، حيث ليس من المفترض أن تقوم الوظيفة بتعديل الجملة.

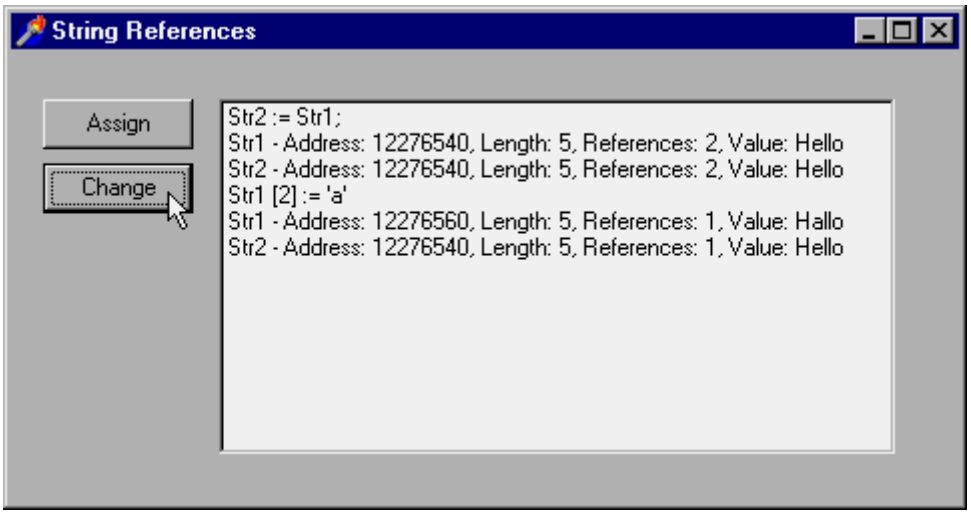
للحصول على عنوان موقع الجملة في الذاكرة (مفيد لمعرفة صفتها الفعلية و لرؤية متى تقوم جملتان بالإشارة إلى نفس منطقة الذاكرة)، قمت ببساطة و في عمق التوليف بتلبيس نوع typecastمن نوع جملة إلى نوع صحيح. الجمل عمليا هي إشارات، هي مؤشّرات :قيمتها تحوي موقع الذاكرة الفعلي للجملة .

من أجل إستخراج عدد الإشارات، جعلت التوليف يعتمد على حقيقة غير معروفة عند الكثيرين و هي أن الطول و عدد الإشارات يتم تخزينها فعليا في الجملة، قبل النصّ الفعلي وقبل الموقع الذي يشير اليه متغيّر الجملة. الموقع (بالسالب) هو 4- و فيه طول الجملة (قيمة يمكنك استخراجها بسهولة أكبر باستعمال وظيفة Length) و 8- و فيه عدد الإشارات.

تذكّر بأن هذه المعلومات الداخلية المتعلقة بمواقع الذاكرة offsets يمكن أن تتغيّر في النسخ المستقبلية من دلفي؛ أيضا لا يوجد أية ضمانة بأن خصائص مشابهة و غير موثّقة سيتم الاحتفاظ بها مستقبلا .

بتشغيل البرنامج، يجب أن تحصل عل جملتين بنفس المحتوى، نفس موقع الذاكرة، و عدد 2 من الاشارات، كما هو ظاهر في الجزء العلوي من القائمة في الشكل 7.1. الآن إذا قمت بتغيير قيمة احدى هاتين الجملتين (لا يهم أية واحدة منهما)، فإن موقع الذاكرة للجملة المعدّلة سوف يتغيّر. هذا هو تأثير تقنية النسخ عند الكتابة.

الشكل 7.1: مثال StrRef يعرض الحالة الداخلية لجملتين، بما في ذلك العدد الحالي للإشارة



يمكننا فعليا توليد هذا التأثير، المبيّن في القسم الثاني من القائمة في الشكل7.1 ، من خلال كتابة التوليف التالي للحدث OnClick للزرّ الثاني:

```
procedure TFormStrRef.BtnChangeClick(Sender: TObject);
begin
  Str1 [2] := 'a';
  ListBox1.Items.Add ('Str1 [2] := ''a''');
  ListBox1.Items.Add ('Str1 - ' + StringStatus (Str1));
  ListBox1.Items.Add ('Str2 - ' + StringStatus (Str2));
end;
```

لاحظ ان التوليف الخاص بمسار *BtnChangeClick* لا يمكن تنفيذه إلا بعد مسار *BtnAssignClick*.و لضمان هذا، يبدأ البرنامج و الزرّ الثاني في حالة خمود disabled (سمة Enabledقيمتها سالبة False)؛ ثم يعيد البرنامج تمكين الزرّ مع نهاية المسار الأول. يمكنك التوسع بحرية في هذا المثال و استخدام وظيفة StringStatus لاستكشاف سلوك الجمل الطويلة في ظروف اخرى متعددة.

جُمل دلفي و PChars في ويندوز

نقطة أخرى مهمة فيما يتعلّق باستخدام الجمل الطويلة و هي: أن هذه الجمل منتهية بصفر-null terminated. هذا يعني أنها متوافقة بالكامل مع الجمل المنتهية بصفر في لغة س و المستخدمة في ويندوز. الجمل المنتهية بصفر هي عبارة عن تتابع لأحرف يلحقها حرف بايت قيمته صفر (أو لاشيء). يمكن التعبير عن هذا في دلفي باستخدام مصفوفة أحرف تبدأ بصفر، و هي نوع البيانات المتبع عادة لبناء الجمل في لغة س. هذا ما يشرح سبب أن مصفوفات الأحرف المنتهية بصفر شائعة الإستخدام في وظائف API في ويندوز (و المبنية على لغة س). فحيث أن جمل باسكال الطويلة متوافقة بالكامل مع الجمل المنتهية بصفر في لغة س، يمكنك ببساطة استخدام الجمل الطويلة و تلبيسها لنوع PChar عندما تحتاج إلى تمرير جملة وظيفة API في ويندوز.

مثال ذلك، لنسخ عنوان نموذج و وضعها في جملة) PChar باستخدام وظيفة API: وهي GetWindowTextثم نسخها لعنوان زرّ، يمكنك كتابة التوليف التالي:

```
procedure TForm1.Button1Click (Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  Button1.Caption := S1;
end;
```

يمكنك ايجاد هذا التوليف في مثال .LongStr لاحظ انك إذا كتبت هذا التوليف و اغفلت عن تخصيص ذاكرة للجملة بواسطةSetLength ، فان البرنامج سينهار غالبا. إذا قمت باستخدام PCharمن أجل تمرير قيمة (و ليس لإستقبال قيمة كما في التوليف أعلاه)، سيكون التوليف أكثر سهولة، لأنه لا توجد حاجة لتعريف جملة مؤقتة و تمهيدها. سطر التوليف التالي يقوم بتمرير سمة Caption الخاصة بملصق Label كمحدد لوظيفةAPI ، ببساطة بتلبيس نوعها لنوع :PChar

```
SetWindowText (Handle, PChar (Label1.Caption));
```

عندما تحتاج لتلبيس جملة عريضة WideString الى نوع يتوافق مع ويندوز، عليك استخدام PWideCharبدلا من PChar لغرض التحويل. الجمل العريضة غالبا ما تستخدم في برامج تستخدم تقنيات OLE وCOM .

بعد أن أبرزت الصورة المشرقة، الآن أريد أن أركّز على الشراك المنصوبة. هناك بعض المشاكل التي يمكن ان تبرز عندما تقوم بتحويل جملة طويلة إلى نوع .PCharبصورة أساسية، المشكلة هي أنه بعد هذا التحويل، ستكون مسؤولا عن الجملة و عن محتواها، و لن تساعدك دلفي بشيء. لاحظ التغيير المحدود التالي لجزء توليف البرنامج الأول أعلاه، Button1Click:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := S1 + ' is the title'; // this won't work
  Button1.Caption := S1;
end;
```

هذا البرنامج سيتم تحويله، لكنك عند تشغيله، فستكون أمام مفاجأة: سمة Caption للزرّ سيكون لها النص الأصلي لعنوان النافذة، بدون نص ثابت الجملة الذي اضفته له .المشكلة هي أن

ويندوز عندما قامت بكتابة الجملة (من خلال استدعاءGetWindowText ، لم تقم بتوصيف طول جملة باسكال الطويلة بطريقة سليمة. لا يزال بمقدور دلفي استخدام هذه الجملة كمخرجات و يمكنها معرفة متى تنتهي هذه الجملة من خلال البحث عن الحرف الصفري المُنهي للجملة، لكنك اذا اتبعتها بأحرف أخري بعد الحرف الصفري، فسوف يتم تخطي هذه الأحرف و اغفالها.

كيف يمكننا تجاوز هذه المشكلة؟ الحلّ هو في اخبار النظام بأن يقوم باعادة تحويل الجملة المرتجعة من استدعاء GetWindowText الى جملة باسكال. عموما، إذا كتبت التوليف التالي:

```
S1 := String (S1);
```

فإن النظام سيتجاهله، لأن تحويل نوع بيانات إلى نفسه عملية غير مجدية. للحصول على جملة باسكال طويلة سليمة، تحتاج إلى إعادة تلبيس الجملة إلى نوع PChar ثم دع دلفي تقوم بالتحويل المناسب ثانية إلى جملة.

```
S1 := String (PChar (S1));
```

حقيقة، يمكن تخطّي عملية تحويل الجملة، لأن التحويلات من PChar إلى جملة تتم آليا في دلفي. ها هنا التوليف النهائي:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := String (PChar (S1));
  S1 := S1 + ' is the title';
  Button3.Caption := S1;
end;
```

بديل أخر و هو إعادة توصيف طول جملة دلفي، باستخدام طول جملةPChar ، بكتابة:

```
SetLength (S1, StrLen (PChar (S1)));
```

سوف تجد ثلاث نسخ من هذا التوليف في مثالLongStr ، الذي له ثلاثة أزرار لتنفيذها. عموما، إذا كنت تريد فقط الوصول إلى عنوان النموذج، يمكنك ببساطة استعمال سمة Caption الخاصة بكائن النموذج نفسه. فلا توجد حاجة لكتابة كل هذا التوليف المربك، و الذي كان فقط لأغراض بيان مشاكل تحويل الجملة. هناك حالات عملية تحتاج فيها للإستعانة بوظائفAPI ، و عندها سيكون عليك الأخذ في الإعتبار مثل هذه الحالة المعقدة.

تشكيل الجُمل

باستخدام علامة الموجب (+) و بعض وظائف التحويل (مثل IntToStr) يمكنك بالتأكيد بناء جمل مركبة من القيم الموجودة. عموما توجد عدة وجهات لتشكيل الأرقام، قيم العملة، و جمل أخرى إلى جملة النهائية. يمكنك استخدام وظيفة Format القوية أو واحدة من الوظائف المرافقة لها.

وظيفة Format تتطلب كمحددات: النصّ الأساسي مع حواجز مكان placeholders (عادة ما تعلّم برمز %) و مصفوفة من القيم، كلّ قيمة خاصة بحاجز مكان. مثلا، لتشكيل رقمين في جملة يمكنك كتابة:

```
Format ('First %d, Second %d', [n1, n2]);
```

حيث n1 و n2 قيمتا عدد صحيح. الحاجز الأول يُستبدل بالقيمة الأولى، الثاني يوافق القيمة الثانية، و هكذا. إذا كان نوع المخرجات لحاجز (يشار إليه بحرف بعد رمز %) لا يوافق نوع المحدد ذو العلاقة، سيظهر خطأ وقت تشغيل. إن عدم وجود تفحص للنوع في وقت التجميع يشكل فعلا أكبر عيب في استخدام وظيفة Format .

وظيفة Format تستخدم محدّد مصفوفة مفتوحة (محدد يمكنه أن يحوي أي عدد من القيم(، الأمر الذي سأناقشه مع نهاية هذا الفصل. حاليا، لاحظ فقط الصيغة الشبيهة بالمصفوفة لقائمة القيم التي يتم تمريرها كمحدد ثان.

بجانب استخدامd% ، يمكنك استخدام حواجز أخرى معرّفة في هذه الوظيفة و التي تم سردها بايجاز في الجدول 7.1. توفّر هذه الحواجز مخرجات افتراضية حسب نوع البيانات. عموما يمكنك استخدام معيّنات تشكيل أخرى لتغيير المخرجات الافتراضية. معيّن العرض، مثلا، يحدد عددا ثابتا من الأحرف في المخرجات، بينما معين الدقّة يشير إلى عدد الخانات العشرية. مثلا:

```
Format ('%8d', [n1]);
```

يحوّل رقم n1 إلى جملة بثمانية أحرف، مع صفّ النص على اليمين (استخدم رمز سالب (-) لصف النصّ لليسار) مالئا الباقي بفراغات.

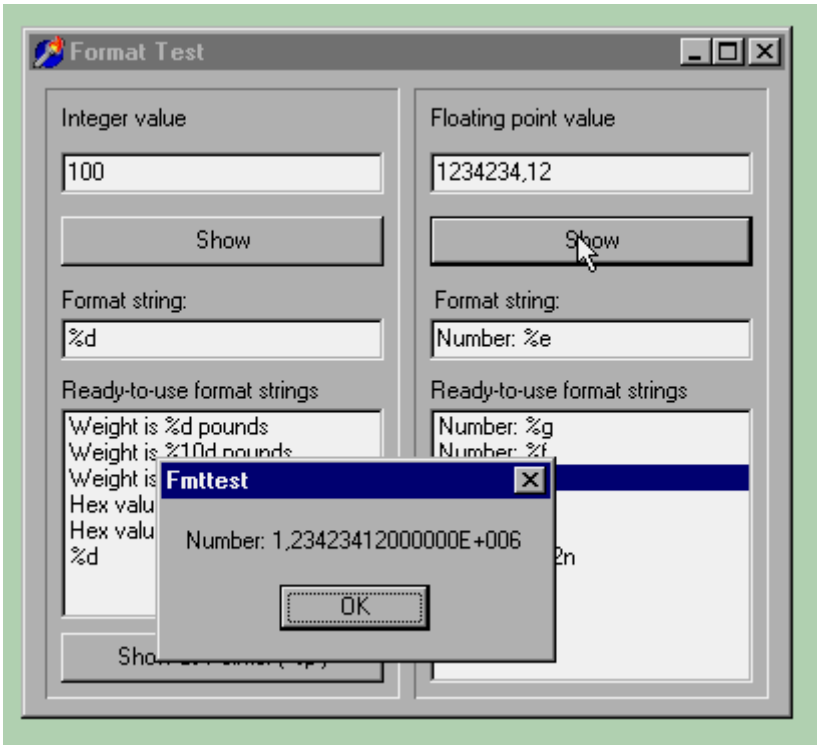
جدول 7.1: معينات النوع لوظيفة Format

معيّن النوع	الوصف
d (decimal)	عشري، قيمة العدد الصحيح يحوّل إلى جملة من الخانات العشرية.
x (hexadecimal)	ستعشري، قيمة الرقم الصحيح تُحوّل إلى جملة من الخانات الستعشرية.
p (pointer)	مؤشّر، قيمة المؤشر يحوّل إلى جملة معبّر عنها بأعداد ستعشرية.
s (string)	قيمة الجملة، الحرف، أو نوع PChar يتم نسخها في مخرجات الجملة.
e (exponential)	مرفوع القوة، قيمة النقطة العائمة تحوّل إلى جملة مبنية على ترميز مرفوع القوة.
f (floating point)	نقطة عائمة، قيمة النقطة العائمة تحوّل إلى جملة مبنية على ترميز النقطة العائمة.
g (general)	عام، قيمة النقطة العائمة تحوّل إلى جملة عشرية بأقرب ما يمكن مستخدمة إما ترميز النقطة العائمة أو مرفوع القوة.
n (number)	رقم، قيمة النقطة العائمة تحوّل إلى جملة نقطة عائمة لكنها أيضا تستخدم فواصل الآلاف.
m (money)	نقود، قيمة النقطة العائمة تحوّل إلى جملة تمثل مقدار العملة. التحويل يعتمد على التوصيف الإقليمي لبيئة التشغيل- انظر ملف مساعدة دلفي تحت موضوع : Currency and date/time formatting variables.

أفضل طريقة لرؤية أمثلة عن هذه التحويلات هي في أن تقوم باختبار تشكيل الجمل بنفسك. لجعل هذا الأمر سهلا قمت بكتابة برنامجFmtTest ، و الذي يسمح للمستخدم بتقديم جمل التشكيل للأرقام الصحيحة و أرقام النقطة العائمة. كما تراه في الشكل7.2 ، هذا البرنامج يعرض نموذجا مقسما إلى جزئين، الجزء الأيسر للأرقام الصحيحة، و الجزء الأيمن لأرقام النقطة العائمة .

كل جزء لدية خانة كتابة أولى مع القيمة الرقمية المراد تشكيلها لجملة. تحت خانة الكتابة الأولى يوجد زرّ لإنجاز عملية التشكيل عارضا النتيجة في نافذة رسالة. ثم تأتي خانة كتابة أخرى، حيث يمكن كتابة جملة التشكيل. كبديل يمكنك ببساطة لمس أحد أسطر مكوّن القائمة، في الأسفل، لإختيار تشكيل جملة محدد سابقا. في كلّ مرّة تكتب تشكيلا جديدا لجملة، يتم اضافته كعنصر جديد للقائمة ذات العلاقة (لاحظ أنه عند غلق البرنامج تُفقد هذه العناصر الجديدة).

الشكل 7.2: مخرجات قيمة نقطة عائمة من برنامج FMTTest



يستخدم توليف هذا المثال نصوص متحكمات مختلفة لتوليد مخرجاته. هذا واحد من ثلاثة مسارات مرتبطة بزرّ Show :

```
procedure TFormFmtTest.BtnIntClick(Sender: TObject);
begin
  ShowMessage (Format (EditFmtInt.Text,
    [StrToInt (EditInt.Text)]));
  // if the item is not there, add it
  if ListBoxInt.Items.IndexOf (EditFmtInt.Text) < 0 then
    ListBoxInt.Items.Add (EditFmtInt.Text);
end;
```

التوليف أساسا يُجري عمليات تشكيل باستخدام نص خانة كتابة EditFmtInt و قيمة متحكم EditInt. إذا كانت جملة التشكيل غير موجودة في القائمة، يتم اضافتها عندئذ .أما إذا لمس المستخدم بدلا من ذلك بندا في القائمة، فإن التوليف ينقل تلك القيمة إلى خانة الكتابة.

```
procedure TFormFmtTest.ListBoxIntClick(Sender: TObject);
begin
  EditFmtInt.Text := ListBoxInt.Items [
    ListBoxInt.ItemIndex];
end;
```

ملخّص

تعد الجمل بالتأكيد أكثر أنواع البيانات شيوعا. بالرغم من أنك تستطيع استخدامها بآمان معظم الأحوال بدون ما يدعو لفهم كيفية عملها، فهذا الفصل يجب أن يكون قد أوضح بالظبط سلوك الجمل، جاعلا بالإمكان استخدام كامل قوة هذا النوع من البيانات .

يتم مناولة الجمل في الذاكرة بطريقة حيوية خاصة، كما يحدث مع المصفوفات المفتوحة. هذا هو موضوع الفصل القادم .

الفصل التالي :الذاكرة

الفصل 8
الذاكرة

ماركو كانتو:
أساسي باسكال

ملاحظة من المؤلّف :سيغطّي هذا الفصل موضوع مناولة الذاكرة .مناقشا مناطق الذاكرة المختلفة، كما يعرّف المصفوفات الحيوية. مؤقتا فقط الجانب الأخير هو المتوفر .

المصفوفات الحيوية في دلفي 4

تقليديا، كانت لغة باسكال تملك دائما مصفوفات ثابتة الحجم. عندما تقوم بتعريف نوع بيانات مستخدما بنية مصفوفة، يجب عليك تحديد عدد عناصر المصفوفة. كما قد يعلم المبرمجون المتمرسون، كان يوجد عدد من التقنيات يمكنك استخدامها لتنفيذ المصفوفات الحيوية، أهمها استخدام المؤشرات و تخصيص الذاكرة المطلوبة و تحريرها يدويا.

ادخلت دلفي 4 طريقة تنفيذ بسيطة جدا للمصفوفات الحيوية، انتظمت بعد نوع الجمل الطويلة الحيوية التي كنت قد ناقشتها منذ قليل. مثلها مثل الجمل الطويلة، المصفوفات الحيوية يتم تخصيصها و تعدادا اشاراتها آنيا، لكنها تخلو من تقنية النسخ عند الكتابة .copy-on-write هذا لا يشكّل مشكلة كبيرة، حيث يمكنك نزع تخصيص deallocate المصفوفة بجعل متغيرها خاليا nil.

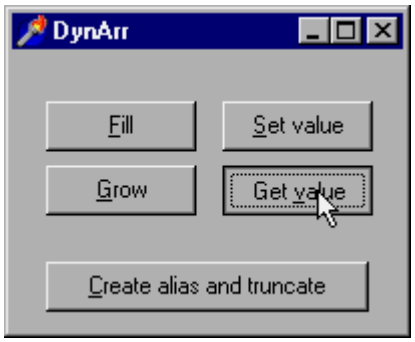
يمكنك الآن ببساطة تعريف مصفوفة بدون الحاجة لتحديد عدد عناصرها ثم تقوم بتخصيصها بحجم معين باستخدام إجراء .SetLength يمكن إستخدام نفس الإجراء لتغيير حجم المصفوفة دون أن تفقد محتوياتها. توجد أيضا إجرائيات أخري لها علاقة بالجمل، مثل وظيفة *Copy* ، و التي يمكنك استخدامها مع المصفوفات.

فيما يلي مقطع من توليف بسيط، يبرز حقيقة انك يجب أن تقوم بتعريف و تخصيص الذاكرة الخاصة بالمصفوفة قبل أن تبدأ باستعمالها:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Array1: array of Integer;
begin
  Array1 [1] := 100; // error
  SetLength (Array1, 100);
  Array1 [99] := 100; // OK
  ...
end;
```

حالما تحدد فقط عدد العناصر في المصفوفة، يبدأ فهرس المصفوفة دائما من صفر .المصفوفات عامة في باسكال معروفة بإمكانية أن يكون حدّها الأدني غير الصفر و أن تكون فهارسها ليست أعداد صحيحة، خاصيّتان لا تدعمهما المصفوفات الحيوية. لمعرفة حالة المصفوفة الحيوية، يمكنك استخدام وظائف Length و High وLow ، كما في أي مصفوفة أخرى. بالنسبة للمصفوفات الحيوية وظيفة Low ترجع دائما قيمة 0، و High ترجع دائما الطول ناقص 1. هذا يعني أنه بالنسبة للمصفوفة الفارغة فإن High ترجع -1 (الأمر الذي عندما تتأمل فيه، تجده رقما غريبا، فهو أقل من ذلك المرتجع منLow).

شكل 8.1: نموذج مثال DynArr



بعد هذا التقديم القصير يمكنني أن أريك مثالا بسيطا يدعى DynArr و يظهر في الشكل 8.1. هو بالتأكيد بسيط لأنه لا يوجد أمرا معقّدا فيما يخصّ المصفوفات الحيوية. أنا سأستخدم المثال أيضا لعرض بعض الأخطاء التي قد يقع فيها المبرمجون. يعرّف البرنامج مصفوفتين جامعتين globalو يقوم بتمهيد الأول في مناول حدث *OnCreate* :

```
var
  Array1, Array2: array of Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // allocate
  SetLength (Array1, 100);
end;
```

هذا يجعل كل القيم صفرا. توليف التمهيد هذا يجعل من الممكن البدء حالا بقراءة و كتابة قيم المصفوفة، دون أي خوف من أخطاء الذاكرة. (طبعا، بافتراض أنك لن تحاول الوصول إلى عناصر خارج الحدّ العلوي للمصفوفة.) و من أجل تمهيد أفضل، لدى البرنامج زرا يقوم بالكتابة داخل كل خلية في المصفوفة:

```
procedure TForm1.btnFillClick(Sender: TObject);
var
  I: Integer;
begin
  for I := Low (Array1) to High (Array1) do
    Array1 [I] := I;
end;
```

زرّ Grow يسمح لك بتعديل حجم المصفوفة بدون أن تفقد محتوياتها. يمكنك إختبار هذا باستعمال قيمة زرّ Get بعد الضغط على زرّ:Grow

```
procedure TForm1.btnGrowClick(Sender: TObject);
begin
  // grow keeping existing values
  SetLength (Array1, 200);
end;

procedure TForm1.btnGetClick(Sender: TObject);
begin
  // extract
  Caption := IntToStr (Array1 [99]);
end;
```

التوليف الوحيد المعقّد قليلا هو في حدث *OnClick* لزرّ .Alias البرنامج ينسخ مصفوفة داخل الأخرى بواسطة العامل=: ، منشئا بكفاءة رديفا) Alias متغير جديد يشير إلى نفس المصفوفة في

الذاكرة). عند هذه النقطة، على أي حال، إذا قمت بتعديل أحد المصفوفتين، ستتأثر الأخرى كذلك، بالنظر إلى أن كلتاهما تشيران إلى نفس منطقة الذاكرة:

```
procedure TForm1.btnAliasClick(Sender: TObject);
begin
  // alias
  Array2 := Array1;
  // change one (both change)
  Array2 [99] := 1000;
  // show the other
  Caption := IntToStr (Array1 [99]);
```

مسار *btnAliasClick* يقوم بعمليتين أخريين. الأولى هي إختبار تساوي على المصفوفتين. هذه العملية لا تختبر العناصر الفعلية للبنيتين و لكن تختبر مناطق الذاكرة التي تشير لها المصفوفتان، فتتفحّص إذا ما كانا المتغيران هما رديفان لنفس المصفوفة في الذاكرة.

```
procedure TForm1.btnAliasClick(Sender: TObject);
begin
  ...
  if Array1 = Array2 then
    Beep;
  // truncate first array
  Array1 := Copy (Array2, 0, 10);
end;
```

العملية الثانية هي إستدعاء لوظيفة*Copy* ، و التي ليست فقط تنقل البيانات من مصفوفة للأخرى، لكنها أيضا تستبدل بالمصفوفة الأولى المصفوفة الجديدة التي تم إنشاؤها بواسطة الوظيفة. التأثير هو أن متغير *Array1* الآن يشير إلى مصفوفة تحوي 11 عنصرا، لذا فإن الضغط على زرّ Get value أو زرّ Set value سوف يولّد خطأ ذاكرة و يبرز اعتراضا exception) إلا إذا كنت قد أوقفت خيار تفحّص المدىrange-checking ، في هذه الحالة يظلّ الخطأ لكن الاعتراض لا يتم اظهاره). توليف زرّ Fill يستمر في العمل جيدا حتى بعد هذا التغيير، حيث أن تحديد عناصر المصفوفة التي سيتم تعديلها يتم بمعرفة حدّيها الحاليين.

ملخّص

يغطّي هذا الفصل مؤقتا المصفوفات الحيوية، و هو بالتأكيد عنصرا مهمّا في إدارة الذاكرة، لكنه جزءا فقط من كامل الصورة. المزيد من الموضوعات ستتبع لاحقا .

بنية الذاكرة التي تم وصفها في هذا الفصل هي من صميم برمجة ويندوز، الموضوع الذي سأتولى تقديمه في الفصل التالي (بدون الخوض في كامل موضوع استخدامVCL ، مع ذلك).

الفصل التالي :برمجة ويندوز

الفصل 9
برمجة ويندوز

ماركو كانتو:
أساسي باسكال

توفّر دلفي تغليفا كاملا للمستويات الدنيا لوظائف API في ويندوز و ذلك باستخدام اوبجكت باسكال و مكتبة المكونات المرئية(VCL) ، لذا فإن الحاجة نادرة لبناء تطبيقات ويندوز باستخدام لغة باسكال صافية و إستدعاءات مباشرة لوظائف API. المبرمجون الذين يحتاجون لإستخدام بعض التقنيات الخاصة غير المدعومة من قبل VCL لا يزال لديهم هذا الخيار في دلفي. سوف ترغب في هذا التوجه في حالات خاصة جدا، مثل بناء مكوّنات دلفي جديدة تعتمد على إستدعاءات API غير معتادة، و أنا لا أريد الخوض في تفاصيل هذا الأمر. بدلا من ذلك، سوف ننظر إلى بعض عناصر تفاعل دلفي مع نظام التشغيل و مجموعة من التقنيات التي قد يسفيد منها مبرمجوا دلفي .

مماسك ويندوز

من بين أنواع البيانات الخاصة بويندوز في دلفي، تعد المماسك handles أكثر المجموعات أهمية. اسم نوع البيانات هذا*Thandle* ، والنوع معرّف في وحدة Windows كالتالي:

```
type
  THandle = LongWord;
```

أنواع بيانات Handle تنفّذ كأرقام، ولكنها لا تستعمل كذلك. في ويندوز، الممسك handle هو إشارة إلى بنية بيانات داخلية للنظام. مثلا، عندما تتعامل مع نافذة) Window أو نموذج Form في دلفي)، يعطيك النظام ممسكا للنافذة. النظام يخبرك بأن النافذة التي تتعامل معها هي نافذة رقم 142 مثلا. بدءا من هذه النقطة، يمكن لتطبيقك أن يطلب من النظام أن يشتغل على النافذة رقم 142 لتحريكها، لتغيير حجمها، لتحويلها إلى أيقونة، و هكذا. العديد من وظائف API في ويندوز، في الواقع، لديها ممسك كأول محدد. هذا لا ينطبق فقط على الوظائف التي تتناول النوافذ؛ بل هناك وظائف API أخرى محددها الأول ممسك رسومي GDI handle ، ممسك لائحة أوامر menu handle ، ممسك صورة bitmap handle ، ممسك لتمثّل instance handle.

بتعبير آخر، الممسك هو توليف داخلي يمكنك استعماله لتشير به إلى عنصر معين يتم مناولته من قبل النظام، يتضمن ذلك نافذة window ، صورة bitmap ، أيقونة icon ، كتلة ذاكرة memory block، مشير cursor ، خط font ، لائحة أوامر menu ، و هكذا. في دلفي، نادرا ما تحتاج إلى استعمال المماسك مباشرة، حيث أنها مخفية داخل النماذج forms ، و الصور، و داخل كائنات دلفي الأخرى. ستكون مفيدة عندما ترغب في استدعاء وظيفة API في ويندوز ليست مدعومة من قبل دلفي .

لتكملة هذا الوصف، فيما يلي مثال بسيط يستعرض مماسك ويندوز. برنامج WHandle لديه نموذج form بسيط، يحتوي فقط على زرّ. في التوليف، قُمت بالاستجابة لحدث OnCreate الخاص بالنموذج، و حدث OnClick الخاص بالزرّ، كما هو واضح في التوصيف النصّي التالي للنموذج الرئيسي :

```
object FormWHandle: TFormWHandle
```

```
  Caption = 'Window Handle'
  OnCreate = FormCreate
  object BtnCallAPI: TButton
    Caption = 'Call API'
    OnClick = BtnCallAPIClick
  end
end
```

حالما يتم خلق النموذج، يقوم البرنامج باستخلاص ممسك النافذة الخاصة بهذا النموذج، من خلال الحصول على سمة Handle الخاصة بالنموذج نفسه. نستدعي IntToStr لتحويل القيمة الرقمية للممسك إلى جملة، ثم نلحقها بعنوان النموذج، كما يمكنك أن تراه في الشكل 9.1 :

```
procedure TFormWHandle.FormCreate(Sender: TObject);
begin
  Caption := Caption + ' ' + IntToStr (Handle);
end;
```

لأن *FormCreate* هي مسار لطبقة النموذج، يمكنها الولوج لسمات و مسارات أخرى تابعة لنفس الطبقة class مباشرة. لهذا، في هذه الإجرائية يمكننا ببساطة الإشارة إلى سمات *Caption* و *Handle* الخاصة بالنموذج مباشرة .

الشكل 9.1: مثال WHandle يعرض ممسك نافذة النموذج. كل مرّة تقوم بتشغيل هذا البرنامج ستتحصّل على قيمة مختلفة .

إذا قمت بتشغيل البرنامج عدّة مرّات تحصل بصفة عامة على قيما مختلفة للممسك. هذه القيمة، في الواقع، يتم تقريرها من قبل ويندوز و يعاد ارسالها إلى التطبيق. (الممسكات لا يتم تقريرها من قبل البرنامج، و لاتملك قيمة محددة مسبقا؛ الممسكات يتم تقريرها من قبل النظام، و التي تقوم بتوليد قيم جديدة في كل مرّة تقوم بتشغيل البرنامج).

عندما يضغط المستخدم على الزرّ، يقوم البرنامج ببساطة باستدعاء وظيفة API و هي SetWindowText، التي تغيّر نص أو عنوان النافذة التي تم تمريرها كمحدد أول. لنكون أكثر دقّة، المحدد الأول لوظيفة API هو ممسك النافذة التي نريد تعديلها :

```
procedure TFormWHandle.BtnCallAPIClick(Sender: TObject);
begin
  SetWindowText (Handle, 'Hi');
end;
```

لهذا التوليف نفس تأثير مناول الحدث السابق، الذي قام بتغيير نصّ النافذة بواسطة إعطاء قيمة جديدة لسمة Caption بالنموذج. في هذه الحالة فإن إستدعاء وظيفة API ليس له معنى، لأنه توجد تقنية لدلفي مشابهة. بعض وظائفAPI ، على أي حال، ليس لها ما يوافقها في دلفي، كما سنرى في أمثلة متقدمة أكثر لاحقا في الكتاب .

التصريحات الخارجية

عنصر مهم آخر في البرمجة لويندوز و هو ما تمثله التصريحات الخارجية external declarations. كانت في الأصل تستخدم لربط توليف باسكال بوظائف خارجية كُتبت بلغة التجميع assembly ، التصريح الخارجي يستخدم عند البرمجة لويندوز لإستدعاء وظيفة من مكتبة DLL (مكتبة الربط الحيوية). في دلفي، يوجد العديد من هذه التصريحات في وحدة Windows unit.

```
// forward declaration
// تعريف مسبق
function LineTo (DC: HDC; X, Y: Integer): BOOL; stdcall;

// external declaration (instead of actual code)
// تعريف خارجي (بدلا من التوليف الفعلي)
function LineTo; external 'gdi32.dll' name 'LineTo';
```

هذا التصريح يعني أن توليف وظيفة LineTo مخزّنة في المكتبة الحيوية GDI32.Dll (أحد أهم مكتبات نظام ويندوز) بنفس الإسم الذي نستخدمه في التوليف. داخل التصريح الخارجي، في الواقع، يمكننا توضيح أن وظيفتنا تشير إلى و ظيفة DLL و التي أصلا لها إسما مختلف .

أنت ناردا ما تحتاج لكتابة تصريح مثل الذي سبق عرضه، ما دامت التصريحات هي بالفعل متضمنة في وحدة Windows و في عدد من وحدات النظام في دلفي. السبب الوحيد الذي قد يدعوك لكتابة توليف لتصريح خارجي هو لإستدعاء وظائف من مكتبات DLL خاصة، أو لإستدعاء وظائف ويندوز غير موثّقة .

ملاحظة : في نسخة دلفي 16-بت، التصريح الخارجي يستعمل اسم المكتبة دون الامتداد extension، و كانت تُتبع بتوجيه name كما في التوليف أعلاه (أو بتوجيه index كبديل، متبوع بترتيب رقم الوظيفة داخل DLL. التغيير قد عكس تبديل النظام لطريقة الولوج للمكتبات: بالرغم من أن WIN32 لا زالت تسمح بالوصول إلى وظائف DLL بواسطة الرقم، إلا أن ميكروسفت أعلنت أن هذه الطريقة لن تدعم مستقبلا. لاحظ أيضا وحدة Windows حلّت محلّ وحدات WinProcs و WinTypes التي في دلفي نسخة 16-بت .

وظيفة نداء عكسي لويندوز

شاهدنا في الفصل 6 أن أوبجكت باسكال تدعم الأنواع الإجرائية procedural types. الإستعمال الشائع للأنواع الإجرائية هي لتوفير وظائف نداء عكسي callback لوظائف API ويندوز .

قبل كلّ شيء، ما هي وظيفة نداء عكسي؟ الفكرة هي أن بعض وظائف API تنجز عمل ما على عدد من العناصر الداخلية في النظام، كما كل النوافذ من نفس النوع. مثل هذه الوظيفة، أيضا تسمّى وظيفة سردية أو تواترية enumerated ، تتطلب كمحدد الفعل الذي ستقوم بانجازه على كل عنصر من العناصر، و الذي يمرر كوظيفة أو إجراء متوافق مع النوع الإجرائي الذي تم اعطاؤه. تستعمل ويندوز وظائف النداء العكسي في ظروف أخرى، لكننا سنحدد دراستنا في هذه الحالة البسيطة .

الآن راقب وظيفة API المسماة EnumWindows ، و التي تملك التوصيف التالي (منسوخة من ملف مساعدة WIN32):

```
BOOL EnumWindows(
  WNDENUMPROC lpEnumFunc,  // address of callback function
  LPARAM lParam // application-defined value
  );
```

بالطبع، هذا تعريف بلغة س. يمكننا أن أن ننظر داخل ملف وحدة Windows لرؤية التعريف الموافق له بلغة باسكال :

```
function EnumWindows (
  lpEnumFunc: TFNWndEnumProc;
  lParam: LPARAM): BOOL; stdcall;
```

باستشارة ملف المساعدة، نجد أن الوظيفة الممررة كمحدد يجب أن تكون بالنوع التالي (مرة أخرى بلغة س):

```
BOOL CALLBACK EnumWindowsProc (
  HWND hwnd, // handle of parent window
  LPARAM lParam // application-defined value
  );
```

هذا يوافق تعريف دلفي التالي للنوع الإجرائي :

```
type
  EnumWindowsProc = function (Hwnd: THandle;
    Param: Pointer): Boolean; stdcall;
```

المحدد الأول هو الممسك handle لكل نافذة رئيسية عليها الدور، بينما الثاني هي القيمة التي نمررها عندما ننادي وظيفة *EnumWindows*. في الواقع في باسكال نوع *TFNWndEnumProc*ليس معرفا بطريقة مناسبة؛ هو ببساطة مؤشر poiter. هذا يعني أنه علينا توفير وظيفة بالمحددات المناسبة ثم نستخدمها كمؤشر، بأخذ عنوان الوظيفة بدلا من استدعائها. لسوء الحظ، هذا يعني أيضا أن المجمّع لن يقدم أي عون في حالة وجود خطأ في نوع أحد المحددات .

> تتطلب ويندوز من المبرمجين اتباع طرقة استدعاء stdcall في كل مرة نقوم فيها باستدعاء وظيفة API أو نمرر فيها وظيفة نداء عكسي للنظام. أما دلفي، افتراضيا ، تستخدم طريقة استدعاء مختلفة و أكثر كفاءة، يشار إليها بالكلمة المفتاحية .register

ها هنا تعريفا مناسبا لوظيفة متوافقة، تقوم بقراءة عنوان النافذة في جملة، ثم تضيفها إلى مربّع قائمة لنموذجم عين :

```
function GetTitle (Hwnd: THandle; Param: Pointer): Boolean;
stdcall;
var
  Text: string;
begin
  SetLength (Text, 100);
  GetWindowText (Hwnd, PChar (Text), 100);
  FormCallBack.ListBox1.Items.Add (
    IntToStr (Hwnd) + ': ' + Text);
  Result := True;
end;
```

النموذج form لديه مربّع قائمة تغطّي تقريبا كامل المنطقة، رفق لوحة panel صغيرة في الأعلى تستضيف زرّا، عندما يُضغط الزرّ، يتم استدعاء وظيفة*EnumWindows* ، و يتم تمرير وظيفة *GetTitle* كمحدد لها :

```
procedure TFormCallback.BtnTitlesClick(Sender: TObject);
var
  EWProc: EnumWindowsProc;
begin
  ListBox1.Items.Clear;
  EWProc := GetTitle;
  EnumWindows (@EWProc, 0);
end;
```

كان بامكاني استدعاء الوظيفة دون تخزين القيمة أولا في متغير مؤقت نوع إجرائي، لكنّي أردت جعل ما يجري في هذا المثال واضحا. تأثير هذا البرنامج مثير للإهتمام بالفعل، كما ترى في الشكل 9.2. مثال Callback يعرض قائمة بكل النوافذ الرئيسية الشغّالة في النظام. معظمها نوافذ مخفية لن تراها عادة (و الكثير منها ليس لها عنوان في الواقع).

شكل 9.2: ناتج مثالCallback ، يسرد النوافذ الرئيسية الحالية(المرئية و المخفية).

برنامج ويندوز محدود

لإكمال تغطية موضوع البرمجة لويندوز و لغة باسكال، أريد أن أعرض لك تطبيقا بسيطا لكنه كاملا و بُني بدون استعمال مكتبة VCL. البرنامج ببساطة يأخذ معطيات سطر الأمر (command-lineمخزّنة من قبل النظام في المتغير العام cmdline) ثم يقوم باستخلاص المعلومات منها بواسطة وظائف باسكال ParamCount و ParamStr. أولى هذه الوظائف تسترجع عدد المعطيات؛ الثاني يرجع المعطى أو المحدد حسب موقعه .

بالرغم من أن المستخدمين نادرا ما يحددون معطيات سطر الأمر في بيئة واجهة رسومية، إلا أن معطيات سطر الأمر في ويندوز تعد مهمة للنظام. مثلا، حالما تقوم بالربط بين امتداد اسم ملف و تطبيق ما، بعدها يمكنك ببساطة تشغيل البرنامج من خلال اختيار الملف المرتط به. عمليا، عندما تقوم بلمسة مزدوجة على الملف، تبدأ ويندوز بتشغيل البرنامج المرتبط وتحيل له الملف المختار كمحدد لأمر سطري .

فيما يلي توليف مصدري كامل للمشروع (ملفDPR ، وليس ملف PAS):

```
program Strparam;

uses
  Windows;

begin
  // show the full string
  MessageBox (0, cmdLine,
    'StrParam Command Line', MB_OK);

  // show the first parameter
  if ParamCount > 0 then
    MessageBox (0, PChar (ParamStr (1)),
      '1st StrParam Parameter', MB_OK)
  else
    MessageBox (0, PChar ('No parameters'),
      '1st StrParam Parameter', MB_OK);
end.
```

التوليف يستخدم وظيفة API و هي *MessageBox* ، ببساطة لتجنب أخذ كامل مكتبة VCL داخل المشروع. برنامج ويندوز صاف كالذي في الأعلى له ميزة ، في الواقع، و هي أن بصمته صغيرة جدا في الذاكرة: حجم الملف التنفيذي للبرنامج حوالي 16 ك.ب .

لتقديم معطيات سطر الأمر لهذا البرنامج، يمكنك استخدام أوامر دلفي Run ثم Parameters. طريقة أخرى و هي أن تفتح مستكشف ويندوز Windows Explorer ، تذهب للدليل الذي يحتوي على الملف التنفيذي للبرنامج، ثم تقوم بجرّ drag الملف الذي المراد تشغيله و اسقاطه فوق الملف التنفيذي. سيقوم مستكشف ويندوز بابتداء البرنامج مستعملا اسم الملف الذي أُسقط كمعطيات لسطر الأمر. الشكل 9.3 يعرض المستكشف و المخرجات المتعلقة به .

الشكل 9.3: يمكنك تقديم محدد سطر الأمر لمثال StrParm بجرّ ملف و اسقاطه فوق الملف التنفيذي في مستكشف ويندوز .

ملخّص

في هذا الفصل شاهدنا تقديما برؤيا منخفضة لبرمجة ويندوز، مناقشين المماسك و برنامج ويندوز بسيط جدا. لأغراض برمجة ويندوز العادية، ستقوم عموما باستخدام دعم التطوير المرئي المقدمة من قبل دلفي و المعتمدة على مكتبة VCL. لكن هذا الأمر خارج نطاق هذا الكتاب، الذي يركّز على لغة باسكال .

الفصل التالي سيغطي المتباينات variants ، اضافة غريبة جدا لنظام أنواع بيانات باسكال، و التي تم ادخالها لتقديم دعم كامل لتقنية OLE .

الفصل 10
المتباينات

ماركو كانتو:
أساسي باسكال

لتوفير دعما كاملا لOLE، تضمنت نسخة 32-بت من دلفي نوع بيانات متباين.Variany هنا أريد أن أناقش نوع البيانات هذا من زاوية عامة. نوع متباين، في الواقع، أصبح له تأثير متزايد على كامل اللغة، لدرجة أن مكتبة مكوّنات دلفي تستخدم هذا النوع بطرق ليس لها علاقة ببرمجة OLE.

المتباينات ليس لها نوع

بصفة عامة، يمكنك استعمال المتباينات لتخزين أي نوع بيانات و لانجاز مختلف العمليات و تحويلات النوع. لاحظ أن هذا ضد التوجّه العام للغة باسكال و ضدّ أعراف البرمجة الجيدة. المتباين يتم فحص نوعه و يتم حسابه في وقت التشغيل. المجمّع compiler لن يحذرك من احتمالات الأخطاء في التوليف، و التي لن يستدلّ عليها إلا بعد اجراء اختبارات مكثفة. بصورة عامة، يمكنك اعتبار أجزاء التوليف التي تستخدم المتباينات هي لتوليف ترجمة فورية interpreted، لأنه، مثل أي توليف لترجمة فورية، العديد من العمليات لا يمكن التقرير بشأنها وحلّها إلى في وقت التشغيل. هذا يوثر بصفة خاصة في سرعة التوليف.

الآن و قد حذرتك من استخدام نوع المتباين، حان الوقت لرؤية ماذا يمكنه أن يفعل .أساسا حالما تقوم بتعريف متغير متباين مثل التالي:

```
var
  V: Variant;
```

يمكنك أن تخصّص له عدة أنواع مختلفة:

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

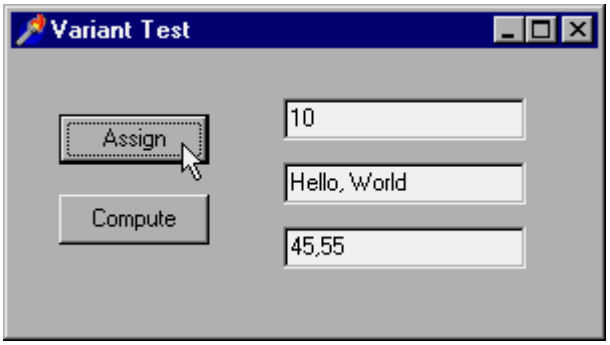
حالما تتحصل على قيمة متباين، يمكنك نسخه إلى أي نوع بيانات آخر متوافق أو غير متوافق. إذا خصصت قيمة لنوع بيانات غير متوافق، تقوم دلفي بإجراء التحويل، إذا استطاعت ذلك. و إلا فإنها ستصدر خطأ وقت التشغيل. في الواقع المتباين يخزن معلومات النوع رفق البيانات، لتسمح بعدد من العمليات في وقت التشغيل؛ هذه العمليات قد تكون مفيدة لكنها بطيئة و غير مأمونة.

أنظر إلى المثال التالي اسمه VariTest ، و الذى هو توسيع للتوليف أعلاه. قمت بوضع ثلاث خانات كتابة على نموذج form جديد، و اضفت بعض الأزرار، ثم كتبت التوليف التالي لحدث OnClick للزرّ الأول:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant;
begin
  V := 10;
  Edit1.Text := V;
  V := 'Hello, World';
  Edit2.Text := V;
  V := 45.55;
  Edit3.Text := V;
end;
```

أمر مضحك، أليس كذلك؟ فبجانب تخصيص متباين يحوي جملة إلى سمة Text في مكون خانة الكتابة، يمكنك تخصيص متباين يحوي رقما صحيحا أو رقم نقطة عائمة إلى نفس سمة Text .

الشكل 10.1: ناتج مثال VariTest بعد الضغط على زرّ Assign



أسوأ من هذا، يمكنك استخدام المتباينات لحساب القيم، كما تري في التوليف المتصل بالزرّ الثاني:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  V: Variant;
  N: Integer;
begin
  V := Edit1.Text;
  N := Integer(V) * 2;
  V := N;
  Edit1.Text := V;
end;
```

كتابة مثل هذا التوليف فيه مخاطرة، أقلّها، إذا احتوت خانة الكتابة الأولى على رقم، فكل شيء سيعمل، غير ذلك، سيبرز اعتراض exception. مرّة أخرى يمكنك كتابة توليف مشابه، لكن في حالة عدم وجود سبب ضاغط، يجب أن تتجنب نوع المتباين؛ تشبث بأنواع بيانات باسكال التقليدية و بأسلوب تفحّص النوع. في دلفي و في مكتبة المكونات المرئية VCL ، تستخدم المتباينات أساسا لدعم تقنية OLE و لمناولة حقول قواعد البيانات.

المتباينات، نظرة أعمق

تتضمّن دلفي نوع تسجيلة متباينة variant record type و هي، *TVarData* ، والذي له نفس توزيع الذاكرة الذي لنوع متباين Variant. يمكنك استعماله للوصول إلى النوع الفعلي للمتباين. بنية *TVarData* تتضمن نوع المتباين هذا، مشار إليه بحقل *VType* ، و بعض الحقول المحجوزة، و القيمة الفعلية.

القيم المحتملة لحقل *VType* تتطابق مع أنواع البيانات التي يمكنك استعمالها في آليةOLE ، و التي غالبا ما تسمّى بأنواع OLE أو أنواع متباينة. فيمل يلي سرد أبجدي كامل لأنواع المتباين المتوفرة :

varArray	varBoolean	varByRef	varCurrency
varDate	varDispatch	varDouble	varEmpty
varError	varInteger	varNull	varOleStr
varSingle	varSmallint	varString	varTypeMask
varUnknown	varVariant		

يمكنك ايجاد أوصاف هذه الأنواع في موضوع VarType function في نظام مساعدة دلفي.

هناك أيضا العديد من الوظائف من أجل العمليات على المتباينات يمكنك استخدامها لصنع تحويلات معينة أو لطلب معلومات عن النوع الذي يحمله المتباين (انظر، مثلا، وظيفة *VarType*). معظم تحويلات النوع و وظائف التخصيص هذه فعليا يتم استدعاؤها بصورة آلية عندما تكتب تعبيرات تستخدم فيها المتباينات. إجرائيات أخرى تدعم نوع متباين (راجع موضوع Variant support routines في ملف المساعدة) تقوم أيضا بعمليات على المصفوفات المتباينة.

المتباينات نوع بطئ!

التوليف الذي يستخدم نوع متباين سيكون بطئ، ليس فقط عندما تقوم بتحويل أنواع البيانات، و لكن أيضا عندما تجمع قيم متباينين يحمل كلاهما عددا صحيحا. هي تقريبا بطيئة مثلها مثل التوليف المترجم لفيجوال بيسك! من أجل مقارنة سرعة خوارزمية مبنية على المتباينات مع أخرى بتوليف مشابه و معتمد على أعداد صحيحة، يمكنك النظر إلى مثال:VSpeed

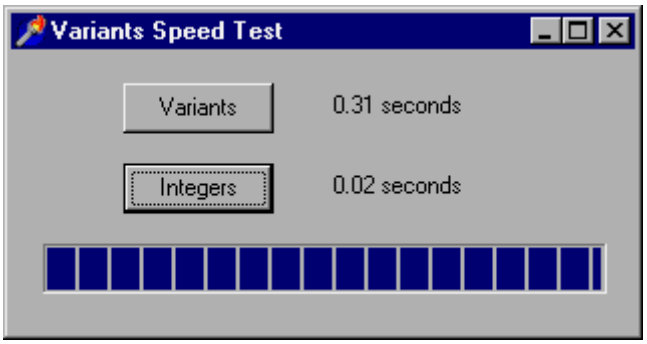
هذا البرنامج ينفذ حلقةloop ، مع قياس سرعتها، عارضا حالته في قضيب انجاز progress bar.فيما يلي أولى الحلقتين المتشابهتين، المبنيتين على الأعداد الصحيحة و المتباينات:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  time1, time2: TDateTime;
  n1, n2: Variant;
begin
  time1 := Now;
  n1 := 0;
  n2 := 0;
  ProgressBar1.Position := 0;
  while n1 < 5000000 do
  begin
    n2 := n2 + n1;
    Inc (n1);
    if (n1 mod 50000) = 0 then
    begin
      ProgressBar1.Position := n1 div 50000;
      Application.ProcessMessages;
    end;
  end;
  // we must use the result
  Total := n2;
  time2 := Now;
  Label1.Caption := FormatDateTime (
    'n:ss', Time2-Time1) + ' seconds';
end;
```

توليف قياس الوقت يستحق النظر، لأنه شيئ يمكنك تطويعه بسهولة لأي نوع من اختبارات الأداء. كما ترى، يستخدم البرنامج وظيفة Now للحصول على قيمة الوقت الحالي و وظيفة FormatDateTimeلعرض الفرق في الوقت، طالبين فقط الدقائق ("n") و الثواني ("ss") في الجملة المصاغة. كبديل، يمكنك استخدام وظيفة API ويندوز *GetTickCount* ، و التي ترجع مؤشر دقيق جدا لمقدار جزء من ألف من الثانية قد مرّت منذ أن بدأ نظام التشغيل.

في هذا المثال فرق السرعة حقيقة كبير جدا لدرجة تلاحظها حتى بدون قياس دقيق للوقت. على كل حال، يمكن رؤية النتائج على حاسوبي الخاص في الشكل 10.2. القيم الفعلية تعتمد على الحاسوب الذي تستخدمه لتشغيل هذا البرنامج، لكن التناسب لن يتغير كثيرا.

الشكل 10.2: فرق السرعات لنفس الخوارزمية، مبنية على أعداد صحيحة و على المتباينات (الوقت الفعلي يتباين من حاسوب لآخر)، كما هو معروض من قبل مثال VSpeed



ملخّص

المتباينات مختلفة جدا عن أنواع بيانات باسكال التقليدية والتي قررت تغطيتها في هذا الفصل القصير و المنفصل. برغم دورها في برمجةOLE ، يمكن أن تكون في متناول اليد لكتابة برامج بسرعة *quick and dirty* بدون الحاجة للتفكير حتى في نوع البيانات. كما سبق و رأينا هذا يؤثر على الأداء كثيرا .

الآن و قد أكملنا تغطية معظم خصائص اللغة، دعني أناقش الهيكلية العامة لبرنامج program و البنيوية التي توفرها الوحدات units .

الفصل التالي :برنامج و وحدات

الفصل 11
برنامج و وحدات

تقوم تطبيقات دلفي باستخدام مكثّف للوحداتunits ، أو بنيات modules البرنامج .الوحدات، في الواقع، كانت الأساس لمفهوم البنيات في اللغة قبل ان يتم ادخال الطبقات .classes في تطبيق دلفي، كل نموذج form له وحدة unit توافقه من خلفه. عندما تضيف نموذج جديد للمشروع project (من خلال الزر الخاص في شريط الأدوات أو الأمر File ثم New ثم Form) تقوم دلفي فعليا باضافة وحدة جديدة، فيها تعريف لطبقة النموذج الجديد .

الوحدات

بالرغم أن كلّ نموذج يكون معرّفا في وحدة، فإن العكس ليس صحيحا. الوحدات لا تحتاج لأن تقوم بتعريف النماذج؛ هي ببساطة تستطيع أن تقوم بتعريف مجموعة من الإجرائيات و جعلها متوفرة. من خلال اختيار الأمر File ثم New ثم أيقونة Unit في صفحة New لمستودع الكائنات Object Repository، تقوم باضافة وحدة جديدة فارغة للمشروع الحالي. هذه الوحدة الفارغة تحتوي على التوليف التالي، الذي يحدد أقسام الوحدة بتجزئتها إلى :

```
unit Unit1;

interface

implementation

end.
```

مفهوم الوحدة بسيط. الوحدة لديها اسما مميزا متوافقا مع اسم الملف. قسم الواجهة interface يعرّف فيه كل ماهو مرئي للوحدات الأخرى، و قسم التنفيذ implementation فيه التوليف الفعلي كذلك التعريفات المخفية الأخرى. أخيرا، يمكن للوحدة أن يكون لها قسم اختياري للتمهيد initialization مع بعض التوليف الخاص بالاستهلالstartup ، و الذي سينفذ حالما يُشحن البرنامج في الذاكرة؛ يمكن للوحدة أيضا أن يكون لها قسما اختياري للإنهاءfinalization ، والذي سينفذ عند وقف البرنامج .

البنيان العام للوحدة، مع كل أقسامها المحتملة، هي كالتالي :

```
unit unitName;

interface

// other units we need to refer to
// وحدات أخرى نحتاج إلى الإشاة إليها //
uses
  A, B, C;

// exported type definition
// تعريفات نوع مصدّرة
type
  newType = TypeDefinition;

// exported constants
// ثوابت مصدّرة
const
  Zero = 0;

// global variables
// متغيرات عامة
var
  Total: Integer;

// list of exported functions and procedures
// سرد بالوظائف والإجراءات المصدّرة
procedure MyProc;

implementation

uses
  D, E;

// hidden global variable
// متغيرات عامة مخفية
var
  PartialTotal: Integer;

// all the exported functions must be coded
// كل الوظائف المصدرة يجب أن يتم توليفها //
procedure MyProc;
begin
  // ... code of procedure MyProc
  //    توليف الإجراء
end;

initialization
  // optional initialization part
// جزء تمهيد اختياري

finalization
  // optional clean-up code
  // توليف اختياري للتنظيف بعد الإستعمال

end.
```

فقرة الاستخدام uses في بداية قسم الواجهة interface تشير إلى أي من الوحدات الأخرى التي نحتاج للوصول إليها في جزء الواجهة من الوحدة. هذا يتضمن الوحدات التي تقوم بتعريف أنواع البيانات التي نشير إليها عند تعريف أنواع بيانات أخرى، مثل المكونات components المستخدمة داخل النموذج الذي نقوم بتحديده .

فقرة uses الثانية، في بداية قسم التنفيذimplementation ، تشير إلى وحدات أخرى نحتاج إلى الوصول إليها فقط في التوليف الخاص بالتنفيذ. عندما تحتاج إلى الإشارة إلى وحدة أخرى من داخل توليف لإجرائيات و مسارات، يجب عليك إضافتها في فقرة uses الثانية بدلا من الأولى. كل الوحدات المشار إليها يجب أن تكون موجودة في دليل directory المشروع أو في دليل مشار إليه في مسار البحث (يمكنك تحديد مسار البحث search path الخاص بالمشروع في صفحة Directories/Conditionals من مربع خيارات المشروع Project Options).

مبرمجو لغة س++ يجب أن يلاحظوا أن تعليمة uses لا تشبه توجيه .include تأثير تعليمة usesهي فقط لجلب جزء الواجهة interface المسبق التحويل precompiled من الوحدات المشار إليها. جزء التنفيذ impementation من الوحدة يتم أخذها في الإعتبار فقط عندما يتم تجميع compile تلك الوحدة. الوحدات التي تقوم بالإشارة إليها يمكن أن تكون إما بشكل توليف مصدري (PAS) أو بشكل ممّع(DCU) ، بشرط أن يكون التحويل قد تم بنفس الإصدارة من دلفي .

واجهة الوحدة يمكنها تعريف عدد مختلف من العناصر، يتضمن ذلك الإجراءات، الوظائف، المتغيرات العامة، و أنواع البيانات. في تطبيقات دلفي، ربما تكون أنواع البيانات هي الأكثر إستعمالا. تقوم دلفي آليا بوضع نوع بيانات طبقة جديدة في الوحدة في كل مرة تقوم فيها بانشاء نموذج. على أي حال، احتواء تعريفات النموذج بالتأكيد ليس الإستعمال الوحيد للوحدات في دلفي. يمكن دائما أن يكون لديك وحدات تقليدية، تحوي وظائف و إجراءات، و يمكن أن يكون لديك وحدات تحوي طبقات لا تشير إلى نماذج أو عناصر مرئية أخرى .

الوحدات ونطاقها

في باسكال، تعدّ الوحدات المفتاح لمفاهيم التغليف encapsulation و المشاهدةvisibility ، و هي تقريبا أكثر أهمية حتى من الكلمات المفتاحية private خاص و public عام في الطبقة. (في الواقع، كما سنرى في الفصل التالي، تأثير الكلمة المفتاحية private له علاقة بنطاق الوحدة التي تحتوي الطبقة.) إن نطاق أو مجال المعرّف) identifier مثل المتغير، الإجراء، الوظيفة، أو نوع البيانات) يشمل جزء التوليف الذي يمكن منه الوصول إلى المعرّف. القاعدة الرئيسية هي أن المعرّف له معنى فقط ضمن نطاقه، فقط ضمن المساحة التي تم فيها تعريفه. فيما يلي بعض الأمثلة .

- المتغيرات المحلية :local variables إذا قمت بتعريف متغير ضمن المساحة التي تحدد إجرائية أو مسار، لا يمكنك استخدام هذا المتغير خارج هذه الإجرائية. نطاق المعرّف يمتد إلى كامل الإجراء، بما في ذلك الإجرائيات التفرعية) nested إلا إذا وُجد معرفّ آخر بنفس الإسم في إجرائية متفرعة تخفي التعريف الخارجي). عندما يقوم البرنامج بتنفيذ الإجرائية يتم تخصيص ذاكرة لهذا المتغير، حالما تنتهي الإجرائية يتم آليا التخلص من هذه الذاكرة .

- المتغيرات الجامعة المخفية :global hidden variables إذا قمت بتحديد معرّف في جزء التنفيذ implementation للوحدة، فلا يمكنك استخدامها خارج هذه الوحدة، لكن باستطاعتك استخدامها في أية منطقة أو إجراء محدد ضمن الوحدة. الذاكرة اللازمة لهذا المعرّف أو المتغيّر يتم تخصيصها حالما يبدأ البرنامج و تظل هذه الذاكرة موجودة لحين انتهائه. يمكنك استخدام قسم التمهيد initialization في الوحدة لتوفير قيم تمهيدية معينة .

- المتغيرات الجامعة :global variables إذا حدّدت معرّفا في جزء الواجهة interface في الوحدة، فإن نطاقه سيمتدّ إلى أي وحدة أخرى تقوم باستعمال نفس الوحدة التي فيها هذا التعريف.

استخدام الذاكرة و مدة بقاء هذا المتغيّر أو المعرّف تشابه المجموعة السابقة؛ الاختلاف الوحيد هو في المدى المنظور منها .

أية تصريحات declarations في جزء الواجهة من الوحدة يمكن الوصول إليها من أي مكان في البرنامج يقوم بتضمين هذه الوحدة في فقرة الاستعمال uses الخاصة به .متغيرات طبقة النموذج معرّفة بنفس الطريقة، و بذلك يمكنك الإشارة إلى النموذج (و ما هو عام من حقوله، مساراته، سماته، و مكوناته) من توليف أي نموذج آخر. بالطبع، تعدّ من العادات البرمجية السيئة أن يتم تعريف كل شيء على أنه جامع global. علاوة على مشاكل الاستهلاك المفرط للذاكرة. فإن استخدام المتغيرات الجامعة يجعل من البرنامج أقلّ سهولة عند الصيانة و التحديث. باختصار، يجب أن تستخدم أقل ما يمكن من المتغيّرات الجامعة .

الوحدات و قواعد التسمية

تعليمة uses هي التقنية المعتادة للوصول إلى مجال أو طاق وحدة أخرى. و بالتالي الوصول إلى تعريفات هذه الوحدة. لكن قد يحدث أن تشير إلى وحدتين تتضمنان نفس المعرّف؛ يمكن أن يكون لديك طبقتان أو إجرائيتان تحملان نفس الإسم .

في هذه الحالة يمكنك ببساطة استغلال اسم الوحدة لتسبق اسم النوع أو الإجراء المحدد في الوحدة. مثلا، يمكنك الإشارة إلى إجراء *ComputeTotal* المعرّف في وحدة *Total* بحيث يكون *Totals.ComputeTotal*. هذا يجب أن لا يتكرر بصورة دائما، حيث أنه ينصح بقوة بعدم استخدام نفس الإسم لشيئين مختلفين في البرنامج .

على كل حال، إذا نظرت داخل ملفات مكتبة VCL و ملفWindows ، ستجد أن بعض وظائف دلفي لديها نفس الإسم (و لكن عموما بمحددات مختلفة) الذي لدى بعض وظائف API لويندوز و المتوفرة في دلفي نفسها. مثال على ذلك إجراء *Beep* البسيط .

إذا أنشأت برنامج دلفي جديد، مع اضافة زرّ، وكتبت التوليف التالي :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Beep;
end;
```

عندها و حالما تضغط على الزرّ ستسمع صوتا قصيراً. انتقل إلى تعليمة uses للوحدة و قم بتغيير التوليف من هذا :

```
uses
  Windows, Messages, SysUtils, Classes, ...
```

إلى النسخة الشبيهة جدا التالية (ببساطة انقل وحدة *SysUtils* إلى ما قبل وحدة *Windows*):

```
uses
  SysUtils, Windows, Messages, Classes, ...
```

إذا حاولت الآن إعادة تحويل recompile التوليف، سيظهر خطأ التحويل "Not enough actual parameters."أي: "المحددات الفعلية غير كافية ."المشكلة هي أن وحدة Windows تقوم بتعريف وظيفة Beep أخرى تحوي محدّدين. بتعبير آخر أعمّ، ما يحدث للتعريفات في الوحدة الأولي المتضمنة في تعليمة uses قد يتم حجبها من قبل تعريفات مشابهة في وحدات لاحقة. الحل الآمن لذلك في الواقع بسيط جدا :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SysUtils.Beep;
end;
```

هذا التوليف سوف يتم تحويله بغض النظر عن ترتيب الوحدات في تعليمات .uses عموما يوجد عدد قليل من تعارض الأسماء في دلفي، سبب هذه القلّة ببساطة أن توليف دلفي عادة يكون مودعا داخل مسارات الطبقات. إن وجود مسارين بنفس الإسم في طبقتين مختلفتين لا يسبّب أية مشكلة. المشكلة دائما تكمن في الإجرائيات ذات النطاق الجامع global .

الوحدات و البرامج

تحتوي تطبيقات دلفي على نوعين من التوليف المصدري: وحدة واحدة أو أكثر و ملف برنامج واحد. يمكن اعتبار الوحدات كملفات ثانوية، يتم الإشارة إليها من قبل جزء التطبيق الرئيسي، ألا و هو البرنامج. نظريا، يُعدّ هذا صحيحا. عمليا، ملف البرنامج هو عادة ملف يتم توليده آليا مع إعطائه دورا محدودا. ببساطة هو مطلوب لبدء البرنامج و تشغيل النموذج الرئيسي. يمكن لتوليف ملف البرنامج، أو ملف مشرع دلفي(DPR) ، أن يتم تعديله يدويا أو باستخدام مدير المشروع Project Manager و ببعض ما هو متعلق بالتطبيق و بالنماذج في خيارات المشروع Project Options.

هيكل ملف البرنامج عادة ما يكون أبسط من هيكل الوحدات. فيما يلي التوليف المصدري لنموذج من ملف برنامج :

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.PAS' {Form1DateForm};

begin
  Application.Initialize;
  Application.CreateForm (TForm1, Form1);
  Application.Run;
end.
```

كما ترى، يوجد فقط قسم الاستخدام uses ، ثم التوليف الرئيسي للتطبيق، محصور بين الكلمتين المفتاحين begin و .end تعليمة uses في البرنامج تعتبر بصفة خاصة مهمة لأنها تُستخدم لإدارة عمليات التحويل و الربط linking للتطبيق .

ملخّص

على الأقل في هذه اللحظة، يعدّ هذا الفصل حول هيكل تطبيق باسكال المكتوب بدلفي أو بأحد نسخ تربو باسكال الأخيرة، هو آخر موضوع في الكتاب. لأية ملاحظات أو طلبات يرجى عدم التردد و مراسلتي الكترونيا .

بعد هذا التقديم للغة باسكال، إذا أردت التعمّق أكثر في عناصر التوجّه الكائني لأوبجكت باسكال في دلفي، يمكنك مراجعة كتابي المنشور . (Sybex, 1999) ***Masterig Delphi 5*** من اجل معلومات أكثر عن هذا الكتاب و عن كتبي الأخرى (و كذلك لمؤلّفين آخرين) يمكنك مراجعة موقعي بالشبكة، **www.marcocantu.com**. نفس هذا الموقع يحتوي النسخ المنقّحة من هذا الكتاب، و الأمثلة الواردة به .