Contain Yourself!

by David Baer

Invariably, along with the much-heralded enhancements of each new release of Delphi come a number of lesser improvements that sometimes don't rate even a passing mention in the *What's New* help file. One of these, the unit Contnrs.pas, made its first appearance in Delphi 5. In this article we'll take a look at the modest, but useful, collection of goodies to be found in this unit.

We'll also explore a few additional topics related to Delphi container classes. For one thing, the venerable workhorse class, `TList`, got a slight facelift in D5. We'll also briefly look at another class, `TInterfaceList`, introduced in D4 but still unknown to many. Finally, getting into the spirit of the discussion, I'll present a new container class, `TFlexiSortList`, that provides a service that you might find useful on occasion.

Old Dogs, New Tricks

The `TList` class has been with us since the first release of Delphi, and there's probably not another utility class that has been as widely employed. It's simple and flexible, and it hasn't needed much alteration to improve those qualities over time. For instance, the only modification to the class from D3 to D4 was to make one of its methods, `Clear`, dynamic (this got modified to virtual in D5).

But in the D5 version, we find a few unexpected surprises (see Listing 1 for an abbreviated declaration). For one thing, we have a new method function, `Extract`, that removes an item from the list, passing back the pointer value as the function result. Why was this added? Did some Borland VCL engineer simply think 'hey, I bet the developer community would really be grateful for this one'? Probably not. As we'll see, many of the new container enhancements are employed in the VCL. And for those that aren't, we can only determine that no use is made of them in the *shipping* VCL. There's an unknown amount of Borland code that uses the VCL and supports the IDE that we're not privileged to examine.

Another change in `TList` would appear to be related to internationalization capabilities of the VCL. The `Error` method is now an overloaded one with two versions, the newly added method's signature allowing the specification of a `PResStringRec` pointer value to allow error text to be extracted from a resource file. This one is hardly worth special mention, but if you take a closer look at `Error`, you may spot a language feature you've never encountered before (at least, this was something I'd never noticed).

`Error` is a routine called from various places in the `TList` implementation code. It provides a simple and concise way to cause an exception to be raised with a minimum of fuss (not that writing code for raising an exception requires all that many keystrokes to begin with). But take a closer look at Listing 2. Here we see the seldom used `raise... at address` form of the statement.

This is a crafty trick to make the exception appear to be raised at the point it would have been, had the `raise` statement appeared right at the point of failure detection. The single `asm` statement cleverly calculates this address and the compiler does the rest.

If you're really observant, you might also spot one small problem with this new code. The trick worked for `TList` in D4 because all calls to `Error` used the first (and only) version of the now overloaded method (the code of this version is unchanged between D4 and D5). In D5, all the calls to `Error` use the signature of the second method version. This version calls the first version. For all the good intentions in evidence, the result is not quite what was intended, as the exception now appears to occur in the outer `Error` method. But, never mind. Thank you Borland for this intriguing Object Pascal lesson anyway.

Always The Last To Find Out

Although the enhancements discussed so far don't really amount to much, one that could be more than a little useful at times comes from a new notification capability in `TList`. D5 defines a new type:

```
TListNofication = (lnAdded,
  lnExtracted, lnDeleted);
```

```
TList = class(TObject)
  private
    ...
  protected
    ...
    procedure Notify(Ptr: Pointer; Action: TListNotification); virtual;
  public
    ...
    procedure Clear; virtual;
    ...
    class procedure Error(const Msg: string; Data: Integer); overload; virtual;
    class procedure Error(Msg: PResStringRec; Data: Integer); overload;
    ...
    function Extract(Item: Pointer): Pointer;
    ...
  end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
class procedure TList.Error(const Msg: string; Data: Integer);
  function ReturnAddr: Pointer;
  asm
    MOV     EAX,[EBP+4]
  end;
begin
  raise EListError.CreateFmt(Msg, [Data]) at ReturnAddr;
end;
class procedure TList.Error(Msg: PResStringRec; Data: Integer);
begin
  TList.Error(LoadResString(Msg), Data);
end;
```

This can be used to implement a notification scheme in classes derived from TList. A new method, Notify (Listing 1 again), is present in TList. It may be overridden to react to inserts and deletes (including extraction operations via the new Extract function). Notification always occurs after the fact. For an insert operation, the item has already been placed in the list. For a delete or extract, it's been removed.

A replace gives both an lnDeleted and lnAdded notification, again, after the fact. But there seems to be a bug in Borland's code. Look at Listing 3, which shows the code of TList.Put, and see if you can spot the problem.

The Notify method in TList itself is empty, so only classes derived from TList can use this new capability. However, Notify is called at all the appropriate places in other TList methods, and derived classes supplying a Notify override will have that method called faithfully.

Personally, I think it would have been a little more useful had Borland added an OnChange event to the class. Users of the class would then be able to directly make use of notifications without the need of a derived class. Although I'll spend no more time on this subject here, you can find a derived TList class that does just this on the companion disk. TNotifyList is a straightforward derivation of TList that extends its capabilities with an OnChange event.

Before we leave the subject of *New Tricks*, let me point out one that the other main workhorse utility list, TStringList, now knows. In Listing 4, you can see the declaration of the new type TStringListSortCompare. TStringList sports a new method, CustomSort, which can be passed a function to be used in making the 'less-than', 'equal' and 'greater-than' determinations. The old Sort method is still there and works as it always has. But now you can easily introduce an alternative ordering of your own.

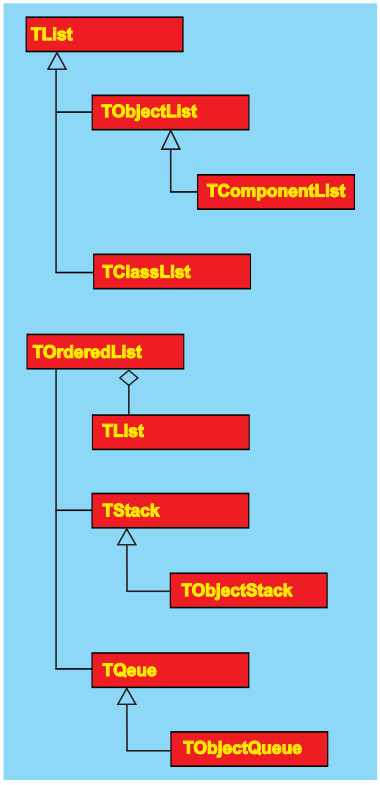
Finally, I want to mention the class TInterfaceList, found in Classes.pas. Based upon the occasional question that appears in the Delphi newsgroups, the existence of this class is unknown to many. It solves one common problem for those needing to keep track of reference counted interfaces.

Not only does it obviate the need for type casting interface references upon insertion, reference, and deletion, its use ensures that the reference counting necessary behind the scenes will be accommodated correctly. Furthermore, the list manipulation is thread safe. TInterfaceList does not derive from TList, but instead contains an internal TThreadList to manage the contained items.

But the most curious aspect of TInterfaceList is that it implements an interface itself: IInterfaceList. This offers most of the same methods and properties that its associated TInterfaceList class does as public methods and properties. When would you use this capability? That's not an easy question to answer. We can see it put to use in only a handful of fairly exotic VCL locales: several uses are made in the internet VCL, and one use is made in Dsgnintf.pas.

What can be safely stated is that if you need to manage a collection of interfaces, TInterfaceList will do the job nicely, even if you make no use of the IInterfaceList at all.

➤ *Figure 1*




```
procedure TList.Put(Index: Integer; Item: Pointer);
var Temp: Pointer;
begin
  if (Index < 0) or (Index >= FCount) then
    Error(@SListIndexError, Index);
  Temp := FList^[Index];
  FList^[Index] := Item;
  if Temp <> nil then
    Notify(Item, lnDeleted);
  if Item <> nil then
    Notify(Item, lnAdded);
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
TStringListSortCompare =
  function(List: TStringList; Index1, Index2: Integer): Integer;
TStringList = class(TStrings)
private
  ...
protected
  ...
public
  ...
  procedure CustomSort(Compare: TStringListSortCompare); virtual;
  ...
end;
```

More Goodies

Let's move on to the new unit, Contnrs.pas. A number of new classes appear in this unit. Figure 1 shows two class hierarchies. The first offers several list types for managing components and classes. The second provides a basic stack and queue capability.

Starting with the first three, TClassList, TObjectList and its derived TComponentList, you can see their declarations in Listing 5. What can we say about these? Well, for one thing, we can be pretty sure they weren't written by Danny Thorpe. Danny, who is one of Borland's pre-eminent VCL gurus, had this to say in his book

Delphi Component Design about using TList in this fashion:

'You'll almost always wind up using a TList in a component, but you'll never create descendants of TList. TList is a worker drone, not a promiscuous ancestor class... You should create a simple wrapper class (derived from TObject, not TList) that exposes only the property and function equivalents of TList that you need ...'

It seems probable that Borland had their own needs in mind when providing these classes. They are used throughout the VCL, in both design-time functionality and in general runtime facilities.

A useful quality of TObjectList (and, by virtue of inheritance, TComponentList) is the property OwnsObjects, which will cause freeing of contained objects upon deletion. Furthermore, an overloaded constructor is provided which allows setting the property value in the constructor call. Also interesting is that TObjectList makes use of the new notification capability of TList discussed earlier.

I don't know about you, but I've been on the verge of writing a TComponent list for several years now. Like a lot of you, I suspect, I *did* write a TIntegerList early on, because I used this type of object frequently and got thoroughly bored with typing requisite casts between Integer and Pointer types. And, OK, I'll confess I didn't follow Danny Thorpe's advice either (but the book hadn't even been published at that time, so give me a little slack).

Although I have occasionally needed a TComponentList, the need never seemed compelling enough to justify getting sidetracked from the work at hand. Looking back, I could have saved myself a good deal of typing had I actually taken the time to write a class like this. At least now we all have a ready-made capability out of the box.

Bucknall On A Budget

The other main class hierarchy in Contnrs.pas offers a simple stack and a simple queue. When I say simple, I mean that the machinery for supplying these capabilities isn't very elaborate. If you were expecting the elegant and highly optimized kinds of access and storage management that you've come to expect of container classes in Julian Bucknall's *Algorithms Alfresco* column, you will be disappointed.

But that's not to say they are inelegant. Rather, they're implemented with concise code that's to the point. Listing 6 shows the declaration of these classes. Both the stack and queue classes are derived from the TOrderedList class, which is itself abstract.

The principal operations supported by both the stack and queue are Push (add item to the container), Pop (remove an item) and Peek (return what would be removed if a Pop were executed,

```
TObjectList = class(TList)
  private
    FOwnsObjects: Boolean;
  protected
    ...
  public
    constructor Create; overload;
    constructor Create(AOwnsObjects: Boolean); overload;
    function Add(AObject: TObject): Integer;
    function Remove(AObject: TObject): Integer;
    function IndexOf(AObject: TObject): Integer;
    function FindInstanceOf(AClass: TClass; AExact: Boolean = True;
      AStartAt: Integer = 0): Integer;
    procedure Insert(Index: Integer; AObject: TObject);
    property OwnsObjects: Boolean read FOwnsObjects write FOwnsObjects;
    property Items[Index: Integer]: TObject read GetItem write SetItem; default;
  end;
  TComponentList = class(TObjectList)
  private
    ...
  protected
    ...
  public
    destructor Destroy; override;
    function Add(AComponent: TComponent): Integer;
    function Remove(AComponent: TComponent): Integer;
    function IndexOf(AComponent: TComponent): Integer;
    procedure Insert(Index: Integer; AComponent: TComponent);
    property Items[Index: Integer]: TComponent read GetItems write SetItems;
      default;
  end;
  TClassList = class(TList)
  protected
    ...
  public
    function Add(aClass: TClass): Integer;
    function Remove(aClass: TClass): Integer;
    function IndexOf(aClass: TClass): Integer;
    procedure Insert(Index: Integer; aClass: TClass);
    property Items[Index: Integer]: TClass read GetItems write SetItems; default;
  end;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
TOrderedList = class(TObject)
private
  ...
protected
  procedure PushItem(AItem: Pointer); virtual; abstract;
  ...
public
  constructor Create;
  destructor Destroy; override;
  function Count: Integer;
  function AtLeast(ACount: Integer): Boolean;
  procedure Push(AItem: Pointer);
  function Pop: Pointer;
  function Peek: Pointer;
end;
TStack = class(TOrderedList)
protected
  procedure PushItem(AItem: Pointer); override;
end;
TObjectStack = class(TStack)
public
  procedure Push(AObject: TObject);
  function Pop: TObject;
  function Peek: TObject;
end;
TQueue = class(TOrderedList)
protected
  procedure PushItem(AItem: Pointer); override;
end;
TObjectQueue = class(TQueue)
public
  procedure Push(AObject: TObject);
  function Pop: TObject;
  function Peek: TObject;
end;
```

```
const
  QUEUE_LIST_CAPACITY = 1024;
constructor TBigQueue.Create;
begin
  inherited Create;
  LL := TList.Create; // the list of lists
  CreateNewItemList;
  PopList := PushList;
  NextPushIndex := 0;
  NextPopIndex := 0;
end;
procedure TBigQueue.CreateNewItemList;
begin
  PushList := TList.Create;
  PushList.Count := QUEUE_LIST_CAPACITY;
  NextPushIndex := 0;
  LL.Add(PushList);
end;
function TBigQueue.Peek: Pointer;
begin
  if (PopList<>PushList) or
   (NextPopIndex<NextPushIndex) then
    Result := PopList[NextPopIndex]
  else
    raise EBigQueueException.Create(
      'Pop or Peek invoked when no item available');
end;
function TBigQueue.Pop: Pointer;
begin
  Result := Peek;
  Inc(NextPopIndex);
  if (PopList = PushList) then begin
    if (NextPopIndex = NextPushIndex) then begin
      NextPopIndex := 0;
      NextPushIndex := 0;
    end;
  end else begin
    if (NextPopIndex = QUEUE_LIST_CAPACITY) then begin
      LL.Delete(0);
      PopList := TList(LL[0]);
      NextPopIndex := 0;
    end;
  end;
end;
procedure TBigQueue.Push(AItem: Pointer);
begin
  if NextPushIndex = QUEUE_LIST_CAPACITY then
    CreateNewItemList;
  PushList[NextPushIndex] := AItem;
  Inc(NextPushIndex);
end;
```

➤ *Listing 7*

```
EFlexiSortListError = class(Exception);
TSortItem = record
  PPI: PPropInfo;
  Descending: Boolean;
  Kind: TTypeKind;
end;
TSortItems = array of TSortItem;
TFlexiSortList = class(TObject)
private
  ContainedClassType: TClass;
  List: TList;
  SortItems: TSortItems;
protected
  function CompareItems(Item1, Item2: Pointer): Integer;
  function Get(Index: Integer): TObject;
  function GetCapacity: Integer;
  function GetCount: Integer;
  procedure InitializeSortItems(const SortFields: array of String);
  procedure Put(Index: Integer; Item: TObject);
  procedure QuickFlexiSort(SortList: PPointerList; L, R: Integer);
  procedure SetCapacity(NewCapacity: Integer);
  procedure SetCount(NewCount: Integer);
public
  constructor Create(ClassType: TClass);
  destructor Destroy; override;
  function Add(Item: TObject): Integer;
  procedure Clear;
  procedure Delete(Index: Integer);
  procedure Pack;
  function Remove(Item: TObject): Integer;
  procedure Sort(const SortFields: array of String);
  property Capacity: Integer read GetCapacity write SetCapacity;
  property Count: Integer read GetCount write SetCount;
  property Items[Index: Integer]: TObject read Get write Put; default;
end;
```

➤ *Listing 8*

but don't remove the item). The only difference between the stack and the queue is that the stack offers last-in-first-out item management and the queue offers first-in-first-out management. In fact, in the implementations of `TStack` and `TQueue` classes, both override the `PushItem` method. The `TStack` version adds an item to the end of the contained `TList`, and the `TQueue` version inserts it at the beginning of the list. Otherwise, they're identical, but we'll examine the performance implications in a moment.

Both `TStack` and `TList` have derived classes, intended for use in managing `TObject` references. There's nothing particularly noteworthy about `TObjectStack` and `TObjectQueue`, but these specializations do provide a convenience that may be appreciated in some situations.

One more thing to note about the two derivative classes is that, unlike the `TObjectList` and `TComponentList` discussed earlier, there is no `OwnsObjects` property. This is entirely reasonable, in that one would rarely pop an object from the list only to see it destroyed.

Q&A Time

Although the implementation of `TQueue` is straightforward enough, I had to wonder how it would perform when the number of items became large. Clearly, the insert at the beginning of the `TList` has to get expensive for large numbers of items. The insert operation allocates additional storage if necessary and then moves the existing contents after the insert point over to accommodate the new item.

Not one to resist a small but interesting science project, I decided to test the performance of `TQueue` against a queue class that caters for large numbers of items. The `TBigQueue` class is included on the companion disk. Let me assure you that I'm not trying to be an alarmist in discussing the shortcomings of `TQueue` for large numbers of items. The performance of the queue is probably just fine for the uses to which Borland presumably puts it in the VCL, where large numbers of items are probably not a worry. We can't be completely certain, though, because `TQueue` and `TObjectQueue` are not used anywhere in the shipping (that is, *visible*) VCL code.

I'll not undertake a full explanation of it, but the basic strategy works as follows. We begin with a single internal list which is given a preset set count of 1K items. The class keeps track of the next push index and the next pop index. The processing does not do any physical movement of existing items. If we have just one internal list, and the last item is popped, the next push and pop indexes are reset to reuse the entire space.

If the next push index exceeds the capacity, a new list is allocated (we keep a list of these internal lists to track the process). During the use of multiple lists, if we pop the last item, then that list is simply destroyed. Listing 7 contains an extract of the class implementation code from which you should be able to get an idea of how this is accomplished.

So, how does performance compare? I constructed a rigorous test that involved two loops each executing 2,000 times. In the first loop, each iteration did four pushes and two pops, and in the second, two pushes and four pops. At the high-water mark, there are 4,000 items in the queue. The differences were dramatic, with `TBigQueue` executing roughly 25 times as fast as `TQueue` for this test case.

Having proved this point, I didn't bother to test performance in non-rigorous circumstances. I would suspect, though, that for modest numbers of items, the two classes would offer fairly comparable performance.

Can I Take Your Order?

Let us now turn our attention to a new container class, `TFlexiSort`. Like so many service classes, this one was inspired by a real-life requirement. I had written an analysis program to read Oracle catalog information and other meta data in order to produce a discrepancy listing. The report items included an error code, a data item identifier, and a severity level (error, warning or hint). I was requested to have the report results sorted by any of the above three pieces of information per the user's preference.

Now, it didn't take all that long to devise a solution. I ended up using a `TList` that contained references to the report items (which were instances of a simple, data-only class), but I had to write three different `Compare` procedures to pass to the `TList.Sort` method depending on which ordering was desired. It seemed that a ready-made class to do this kind of thing would be nicely reusable.

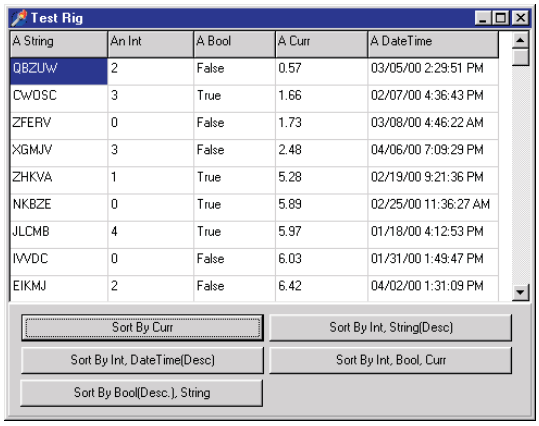
But how can one devise a sort capability for a container class when one doesn't know what data members its contained class will have in the first place? Never fear, it's RTTI to the rescue!

Let's briefly review some basic facts about RTTI (runtime type information). RTTI's most notable commission is that it enables Delphi's component property streaming machinery. It's central to the power of Delphi's RAD-ness.

Classes which are defined with the `$TYPEINFO` (alias `$M`) compiler directive turned on will have RTTI information generated, as will any class which descends from such a class. All classes that derive directly or indirectly from `TPersistent` will have RTTI produced. RTTI isn't produced for everything in a class, but is restricted to published properties.

➤ *Listing 9*

```
function TFlexiSortList.CompareItems(Item1, Item2: Pointer):
  Integer;
var
  I: Integer;
  PPI: PPropInfo;
  function CompareOrd(I1, I2: LongInt): Integer;
  begin
    if I1 > I2 then
      Result := 1
    else if I1 = I2 then
      Result := 0
    else
      Result := -1;
    end;
  function CompareFloat(I1, I2: Extended): Integer;
  begin
    if I1 > I2 then
      Result := 1
    else if I1 = I2 then
      Result := 0
    else
      Result := -1;
    end;
  function CompareInt64(I1, I2: Int64): Integer;
  begin
    if I1 > I2 then
      Result := 1
    else if I1 = I2 then
      Result := 0
    else
      Result := -1;
    end;
begin
  Result := 0;
  I := 0;
  while ((Result = 0) and (I < Length(SortItems))) do begin
    PPI := SortItems[I].PPI;
    if PPI <> nil then begin
      case SortItems[I].Kind of
        tkInteger, tkChar, tkEnumeration:
          Result := CompareInt64(GetOrdProp(Item1, PPI),
            GetOrdProp(Item2, PPI));
        tkFloat:
          Result := CompareFloat(GetFloatProp(Item1, PPI),
            GetFloatProp(Item2, PPI));
        tkString, tkLString, tkWString:
          Result := AnsiCompareStr(GetStrProp(Item1, PPI),
            GetStrProp(Item2, PPI));
        tkInt64:
          Result := CompareInt64(GetInt64Prop(Item1, PPI),
            GetInt64Prop(Item2, PPI));
        end;
      end;
    if Result = 0 then
      Inc(I)
    else if SortItems[I].Descending then
      Result := -Result;
  end;
end;
procedure TFlexiSortList.InitializeSortItems(
  const SortFields: array of String);
var
  I: Integer;
  S: String;
  PPI: PPropInfo;
  PTI: PTypeInfo;
  TK: TTypeKind;
begin
  SetLength(SortItems, High(SortFields) + 1);
  for I := 0 to High(SortFields) do begin
    SortItems[I].PPI := nil;
    SortItems[I].Descending := False;
    if Copy(SortFields[I], 1, 2) = 'D:' then begin
      SortItems[I].Descending := True;
      S := Copy(SortFields[I], 3, $7FFF);
    end else if Copy(SortFields[I], 1, 2) = 'A:' then
      S := Copy(SortFields[I], 3, $7FFF)
    else
      S := SortFields[I];
    PPI := GetPropInfo(ContainedClassType.ClassInfo, S);
    if PPI = nil then
      raise EFlexiSortListError.Create('Sort item ' + S +
        ' is not a published property for ' +
        ContainedClassType.ClassName);
    SortItems[I].PPI := PPI;
    PTI := PPI.PropType^;
    TK := PTI.Kind;
    if not (TK in [tkInteger,tkChar,tkEnumeration,
      tkFloat,tkString,tkLString,tkWString,tkInt64]) then
      raise EFlexiSortListError.Create('Sort item ' + S +
        ' is not a valid type for sorting in class ' +
        ContainedClassType.ClassName);
    SortItems[I].Kind := TK;
  end;
end;
procedure TFlexiSortList.QuickFlexiSort(
  SortList: PPointerList; L, R: Integer);
...
begin
  // standard quicksort logic that calls CompareItems
end;
```



➤ *Figure 2*

```
procedure TfrmTestRig.bSort1Click(
  Sender: TObject);
begin
  FSL.Sort(['A:PInt',
   'D:PDateTime']);
  LoadGrid;
```

➤ *Listing 10*

RTTI not only allows retrieval of data type information of published properties to be accessed via a property name, RTTI-based services exist which can be used for property value access (again, via a property name) at runtime.

Now we're getting somewhere. If we impose the simple requirement that class data members upon which we want to sort must be published properties, we're on our way. Of course, the other requirement is that the RTTI will be produced for the class in the first place, as described in the preceding paragraph.

Listing 8 shows the declaration of the `TFlexiSortList` class. The first thing to note is that the constructor requires a parameter specifying the class type of the contained class. The `Add` method will force compliance We cannot sort a list of heterogeneous object types (unless all derive from a common ancestor class, and the sort items are limited to properties of that base class).

Most of the public methods are ones that will be familiar to users of `TList`. A few (like `Exchange`) are omitted because they are probably of limited utility in a class like this. The one major departure from `TList` is found in the `Sort` method.

The `Sort` method takes an open array of strings as a parameter. Each element in the array must specify the name of a published property. The property name may be prefixed by `A:` or `D:` to denote ascending or descending. If the prefix is omitted, ascending is assumed.

Not all property types are allowed. Sorting on a set property doesn't seem to make much sense, for example, and sorting on an event procedure makes even less. The types allowed include all cardinal, integer, float and string types. You could make a case for also supporting `Variant` types as candidate sort items, but this implementation forgoes that option.

Listing 9 contains the implementation code involved in servicing a sort request. There are several places where RTTI comes into the picture. In the method `InitializeSortItems`, the sort 'columns' are checked for compliance to the sort types supported. We can provide some optimization here by retrieving and storing the `PPropInfo` pointer for each 'column', rather than doing this later for every item comparison. We can also make things run a bit faster (and make the comparison routine coding easier) by looking up and saving the `TTypeKind` value for the property.

Although space doesn't permit a detailed discussion of what's going on here with respect to RTTI access, good accounts of this are available elsewhere. For my own education in developing this code, I referred back to my indispensable copy of *Secrets of Delphi 2* by Ray Lischner. Unfortunately, this book has long been out of print, but I highly recommend that you seek out Ray's new work, *Delphi in a Nutshell*, which should be on the shelves around the time of the publication of this issue. I haven't seen it myself yet, but I fully expect the new volume to become a much referenced member of my Delphi library.

Getting back to the business at hand, it remains only to explain the actual sorting process. Like the `TList` sort, `TFlexiSortList` uses an internal quick sort to run the show. Unlike `TList`, the developer need not supply a compare function. Item comparisons are done by the protected method `CompareItems`. In it, you can see that the values for comparison are extracted using the RTTI support routines, `GetOrdProp`, `GetFloatProp`, etc.

There is obviously a performance overhead in using the `GetXxxProp` in every item comparison. Unfortunately, these functions are not called just once for each property of an item, they are called every time an item is compared to another. Such is the price for the flexibility we've gained. However, for a great many applications where maximum optimization is unnecessary, or when relatively small sets of items need be sorted, the performance will be completely acceptable.

Finally, a demo program shows all this in action. It draws upon services of `TFlexiSortList` to manage a collection of objects of a class that sports a `string`, an `integer`, a `Boolean`, a `TDateTime` and a `Currency` property. Figure 2 shows a form

that contains a `TStringGrid` displaying these values and a number of buttons by which the rows can be sorted in different orderings. Listing 10 shows the code for one of the button's `OnClick` handlers in which you can see how easy it is to invoke the class's sorting services.

Putting A Lid On It

Before closing, I would like to suggest that the `TOrderedList` family of classes in Contnrs.pas offers a nicely compact lesson in the power of inheritance. If you are new to the study of the object-oriented approach to software design, you might learn much from a brief examination of these classes.

Oh yes, remember the bug in `TList.Put`? Look at the first of the two `Notify` calls in Listing 3. It seems to me the first parameter ought to be `Temp`, not `Item`. What do you think?

David Baer is Chief Software Architect at Spear Technologies in San Francisco. He's had that job title printed on his business card for over two years now, unlike a certain individual at Microsoft! Contact him at dbaer@speartechnologies.com