Type-Safe Lists

by Jim Cooper

Looking at all that has been said and written about Delphi, you could be forgiven for thinking that it is only a database development tool. While Delphi is good for these sort of applications, relational databases are not necessarily the ideal way to store data.

If the data is not persistent (that is, it does not last beyond the execution of the program) and will fit into available memory, using temporary database tables is an unnecessary complication. More importantly, object oriented data structures can be very difficult to store in tables. Imagine trying to store a list of components and all their property values in Paradox tables, for instance! While there are techniques that can, and sometimes must, be used to overcome this problem, often a tidier solution is desirable.

Fortunately, help is at hand. Although Delphi doesn't come with a great range of data structures, one of the most useful, the list, is supplied in the form of `TList`. This is not a component, so you won't find it on your palette, but you will find it popping up all over the place. The class `TStringList` uses it for string storage, for example. `TList` is easy to use, but you need to create and destroy it explicitly.

You can add anything you like to a `TList`, because `TList` actually only holds *pointers* to other objects. So, you can have a `double`, a `string` and a `TButton` all in the same list.

There are, however, a couple of restrictions that you should bear in mind. One is that because `TList` is actually implemented as an array of pointers the maximum number of items in a list is 16,380 in Delphi 1.0x (it's something over 13 million in Delphi 2). The other major restriction is that a call to the `Free` method will only destroy the *pointers*, and not *what they are pointing at*. Sometimes this is desirable and sometimes not. I will return to this point later.

Principal Methods:

`Create`	The constructor
`Free`	The destructor
`Add`	Appends an item to the list
`Insert`	Inserts an item at a given position in the list
`First`	Returns the first item in the list
`Last`	Returns the last item in the list
`Delete`	Makes the pointer to the given item nil, the item itself still exists
`Remove`	Removes the pointer to the given item from the list

Principal Properties:

`Count`	A run-time, read only property that returns the number of items in the list
`Items`	An indexed property that allows access to a given item in the list; note that the first item in the list is `Items[0]` and therefore the last is `Items[Count-1]`

➤ *Table 1: TList principal methods and properties*

Using TList

A brief run through of the main properties and methods of `TList` is probably in order, as a couple of them are a bit tricky. The main methods and properties are shown in Table 1. There is also a property called `List` which allows direct access to the array of pointers, ie these two expressions refer to the same item: `MyList.List^[0]` and `MyList.Items[0]`.

I never use `List` as I find the `Items` property is easier and makes code easier to understand. The remaining properties and methods are described in Delphi's on-line help. You won't find any mention in the manuals, however.

Those of you who are used to list structures in other environments will have noticed that a few methods seem to be missing. For a (brief) discussion of how to implement a descendant of `TList` that disposes of all its items when it is destroyed and has iterator methods `ForEach`, `FirstThat` and `LastThat`, see the excellent *Delphi Developer's Guide* by Pacheco and Teixeira, pages 492-493.

Homogeneous Lists

As it stands, `TList` is great for storing heterogeneous data. That is, you can store different types of items in the same list, the aforementioned `double`, `string` and `TButton`, for instance. This is not usually the case, however, and you often want a homogeneous list where all the items are of the same type, or at least, all derived from the same ancestor class (because then you can use polymorphism). While you can do this by exercising a little programming discipline, I prefer to derive type-safe classes that the compiler can type check for me and that do not require typecasting when used. I find that this increases the readability and maintainability of my code. This article will describe how this can be done, and because it is a very mechanical task, I will also present an expert to automate the process. Along the way I'll show how to override properties and non-virtual methods.

A small example may help to clarify a few points. Let's suppose you have a drawing program that draws circles and squares, and that you want to keep a list of what to draw so that you can redraw as required, when doing a screen refresh, say. Listing 1 is the code for a unit that does exactly this. On the disk as file OLDSHAPE.PAS is a unit called `Shapes` which contains the

(very simple) code for the classes of objects to draw. There is a base class TMyShape which encapsulates everything common to all shapes. In this case, that is the size, position and colour of the shape and the fact that all shapes need to know how to draw themselves onto a canvas. The Draw method is overridden in each of the descendant shapes, so that the correct shape is drawn, but each overridden Draw method calls the Draw method inherited from TMyShape. This is because setting the pen and brush colours is common to all shapes.

The unit MainForm contains the code for a simple dialog that maintains a list of shapes to draw (see Figure 1). To use the program, just enter values for the type of shape and its size, top and left positions and colour and click the Add button. Repeat this as often as you need. Every time the Add button is clicked, a new object is added to the variable ShapeList. This is a TList and therefore must be explicitly created and destroyed. This is done in the form's OnCreate and OnDestroy event handlers.

While this is all fairly straightforward, there are a couple of points to note. One is that when adding a new shape to ShapeList no typecasting is necessary and the code is nice and clear. However, there is nothing to stop you adding something that is not a TMyShape. This would result in an exception being raised during the drawing and destroying of the list. This isn't much of a problem in such a small program, but in larger projects, particularly when you are working on someone else's code, it may not be so obvious what you can and can't do. The other point is that every time you access an item in ShapeList it must be typecast. For instance, look at the code in the DrawAllShapes method in Listing 1.

This sort of thing reduces the readability of your code. What would be nice is a list that only holds TMyShape objects. Then the compiler will complain if you try and add some other sort of object to the list and you wouldn't have to do typecasting quite so often. In other words, we want to derive a new list class that has all the methods and properties of TList, but will only hold items of type TMyShape. What we need to do is override all those methods and properties that refer to type Pointer and make them refer to TMyShape instead. The relevant declarations are shown in Listing 2 and are taken from the file DELPHI\SOURCE\VCL\CLASSES.PAS (if you have the VCL source) or DELPHI\DOCS\CLASSES.INT (if you haven't).

At first glance the source seems to suggest that you cannot override any of them because none are declared virtual. This is not actually necessary, as there are two ways to override a method.

Firstly, you can use the override keyword. This is the usual method, but can only be applied to methods which are declared as virtual. The declaration of the new method must also be identical. That is, the number and types of parameters must be identical, it must return the same type if it is a function and it must have the same name. You will usually call the inherited method in the implementation of the new method. A common example is the destructor of any new classes you declare. If you look at Listing 3 you will see an example of a simple class that encapsulates a font. The virtual Destroy method in TObject is overridden in TMyObject

➤ *Listing 1*

```
unit Mainform;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls, Shapes;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    ShapeRadioGroup: TRadioGroup;
    SizeEdit: TEdit;
    PaintBox: TPaintBox;
    AddBtn: TButton;
    Label2: TLabel;
    TopEdit: TEdit;
    Label3: TLabel;
    ColourComboBox: TComboBox;
    Left: TLabel;
    LeftEdit: TEdit;
    procedure AddBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    ShapeList : TList;
    procedure DrawAllShapes;
  public
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.DrawAllShapes;
var i : Integer;
begin
  {Step through the list drawing each shape}
  for i := 0 to ShapeList.Count - 1 do
    TMyShape(ShapeList.Items[i]).Draw(PaintBox.Canvas);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  {Ensure a colour is selected}
  ColourComboBox.ItemIndex := 0;
  {Create the list that will hold all the shapes to draw}
  ShapeList := TList.Create;
end;
procedure TForm1.FormDestroy(Sender: TObject);
var
  i : Integer;
begin
  {Free all the shapes memory}
  for i := 0 to ShapeList.Count - 1 do
    TMyShape(ShapeList.Items[i]).Free;
  {Free the list itself}
  ShapeList.Free;
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
  {Redraw all the shapes}
  DrawAllShapes;
end;
procedure TForm1.AddBtnClick(Sender: TObject);
const Colours : array[0..2] of TColor =
(clRed,clGreen,clBlue);
var   NewShape : TMyShape;
begin
  {Create a new shape object based on the
   radio button selection}
  case ShapeRadioGroup.ItemIndex of
    0 : NewShape := TCircle.Create;
    1 : NewShape := TSquare.Create;
  end;
  {Set the properties of the new shape}
  NewShape.Size   := StrToInt(SizeEdit.Text);
  NewShape.Top    := StrToInt(TopEdit.Text);
  NewShape.Left   := StrToInt(LeftEdit.Text);
  NewShape.Colour := Colours[ColourComboBox.ItemIndex];
  {Add the shape to the list of shapes}
  ShapeList.Add(NewShape);
  {Redraw everything in the list, as the new shape may
   cover old ones}
  DrawAllShapes;
end;
end.
```

so that the font is freed first and then the inherited method is called to free all resources associated with TObject.

The second option is *not* to use the override keyword. Seems obvious really. You can do this to both virtual and non-virtual methods. The Delphi manual tells us that this will replace the inherited method. However, we can still call the inherited method from within the new method. A common example here is the constructor of new classes that you declare. In Listing 3 you will see that I have done this for the Create method. The reason for doing this should hopefully be clear. When a TMyObject is created, I can pass a parameter to initialise it. I believe this flexibility is why the Create method of TObject is not declared virtual.

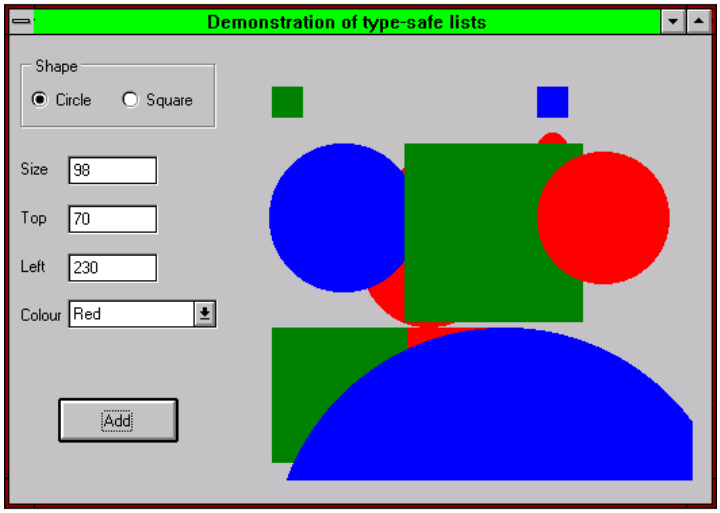
Overriding properties is a bit trickier, but not much. There are again two ways to do it.

If the property has access methods, that is, the read and write parts of a property declaration call methods rather than directly access a field of the class, then you can override those methods. We will do this with the protected methods Get and Put, which are used by the property Items in Listing 2. Incidentally, this shows why it is always a good idea to make accessors protected instead of private. You never know when it might be necessary to get at them.

If the property does not have access methods, or you want it to use different ones, re-declare it. We will have to do this with the List property because it just returns a pointer to the internal array of pointers that is used by TList. We will have to declare a new type here and declare an access method that basically just performs a typecast. This works because an array of TObject is actually an array of Pointer. I don't think this is particularly elegant, though, and in fact I never use the List property, because I think using Items makes for clearer code.

So, all we need to do to make a type-safe list is derive a new class, TShapeList say, from TList, then re-declare all the properties and

➤ *Figure 1*




```
PPointerList = ^TPointerList;
TPointerList = array[0..MaxListSize - 1] of Pointer;
TList = class(TObject)
protected
  function Get(Index: Integer): Pointer;
  procedure Put(Index: Integer; Item: Pointer);
public
  function Add(Item: Pointer): Integer;
  function Expand: TList;
  function First: Pointer;
  function IndexOf(Item: Pointer): Integer;
  procedure Insert(Index: Integer; Item: Pointer);
  function Last: Pointer;
  function Remove(Item: Pointer): Integer;
  property Items[Index: Integer]: Pointer read Get write Put; default;
  property List: PPointerList read FList;
end;
```

➤ *Listing 2*

```
TMyObject = class(TObject)
  fFont : TFont;
  constructor Create(InitialFont : TFont);
  destructor  Destroy; override;
end;
implementation
constructor TMyObject.Create(InitialFont : Tfont);
begin
  inherited Create;  {Calling a nonvirtual inherited method}
  fFont := Tfont.Create;
  fFont.Assign(InitialFont);
end;
destructor TMyObject.Destroy;
begin
  fFont.Free;
  inherited Destroy; {Calling a virtual inherited method}
end;
```

➤ *Listing 3*

methods from Listing 2, replacing all occurrences of Pointer with TMyShape, and those of TList with TShapeList, then finally implement all the overridden properties and methods, performing the appropriate typecasts on the inherited methods.

The result is shown in Listing 4. If we work our way through this listing, the application of the techniques I've just described should become clear. In the interface section, the first declarations are:

```
PMyPointerList =
  ^TMyPointerList;
TMyPointerList =
  array[0..MaxListSize - 1] of
  TMyShape;
```

These declarations are used to override the List property, which we want to return a pointer to an

array of TMyShape objects (in TList it returns a pointer to an array of Pointer types, remember). We just re-declare the List property, as shown, which replaces the one in TList (see Listing 2). It will now return a pointer to an array of TMyShape objects. The access method GetList is similarly re-declared and hence also replaces the GetList method of TList.

In the implementation section, the function GetList must be defined because we have re-declared it. Note that all we need to do is call the inherited property and perform a typecast.

This is the most complicated action and everything else is easy from here. Take the Items property, for example. We now want it to return a TMyShape instead of a Pointer, so we re-declare it as shown. Because it has access methods, we need to ensure they use TMyShape instead of Pointer. They are also not virtual, so we re-declare them (in the protected section this time). Because we have re-declared them, they need new definitions in the implementation section. Note that once again, all we do is call the inherited methods, using typecasts where necessary. All the other methods are overridden in exactly the same way.

This is now a completely type-safe class and does not require typecasting of the list items anywhere in your code. For instance, using the definition for TMyObject in Listing 3, we can do things like this:

```
Label1.Font.Assign(
  MyList.First.fFont);
MyList.Items[3].fFont.Name :=
  'Arial';
```

It also means that we can tidy up the code in the example given in Listing 1. In the MainForm unit we need to change the declaration of ShapeList to:

```
ShapeList : TShapeList;
```

When ShapeList is created in FormCreate it needs to be like this:

```
ShapeList := TShapeList.Create;
```

```
unit NewList;
interface
uses Classes;
type
  PMyPointerList = ^TMyPointerList;
  TMyPointerList = array[0..MaxListSize - 1] of TMyShape;
  TShapeList = class(TList)
  protected
    function Get(Index : Integer) : TMyShape;
    procedure Put(Index : Integer;Item : TMyShape);
    function GetList : PMyPointerList;
  public
    function Add(Item : TMyShape) : Integer;
    function First : TMyShape;
    function Expand : TShapeList;
    function IndexOf(Item : TMyShape): Integer;
    procedure Insert(Index : Integer;Item : TMyShape);
    function Last : TMyShape;
    function Remove(Item : TMyShape) : Integer;
    property Items[Index : Integer] : TMyShape read Get write Put;
    property List : PMyPointerList read GetList;
  end;
implementation
function TShapeList.GetList : PMyPointerList;
begin
  Result := PMyPointerList(inherited List);
end;
function TShapeList.Get(Index : Integer): TMyShape;
begin
  Result := TMyShape(inherited Get(Index));
end;
procedure TShapeList.Put(Index : Integer;Item : TMyShape);
begin
  inherited Put(Index,Item);
end;
function TShapeList.Add(Item: TMyShape) : Integer;
begin
  inherited Add(Item);
end;
function TShapeList.Expand : TShapeList;
begin
  Result := TShapeList(inherited Expand);
end;
function TShapeList.First : TMyShape;
begin
  Result := TMyShape(inherited First);
end;
function TShapeList.IndexOf(Item : TMyShape) : Integer;
begin
  Result := inherited IndexOf(Item);
end;
procedure TShapeList.Insert(Index : Integer;Item : TMyShape);
begin
  inherited Insert(Index,Item);
end;
function TShapeList.Last : TMyShape;
begin
  Result := TMyShape(inherited Last);
end;
function TShapeList.Remove(Item: TMyShape): Integer;
begin
  Result := inherited Remove(Item);
end;
end.
```

➤ *Listing 4*

We can now also remove the typecasting in the routines FormDestroy and DrawAllShapes.

If, in the future, anybody modifies your code and by mistake tries to add something that is not a descendant of TMyShape to ShapeList, they will get an error at compile time, instead of an exception at run time.

I have made one more refinement and added new methods to TShapeList. It is better encapsulation to make operations on the list methods. This way all code that operates on the list is in the one place and this makes later modifications easier. I have overridden the Destroy destructor so that it does the iteration through the list freeing the shape objects and I have introduced a new method called DrawAllShapes that does the drawing. The new Shapes unit which includes the TShapeList class is included on the disk and the final version of the MainForm unit is shown in Listing 5.

TComponentList

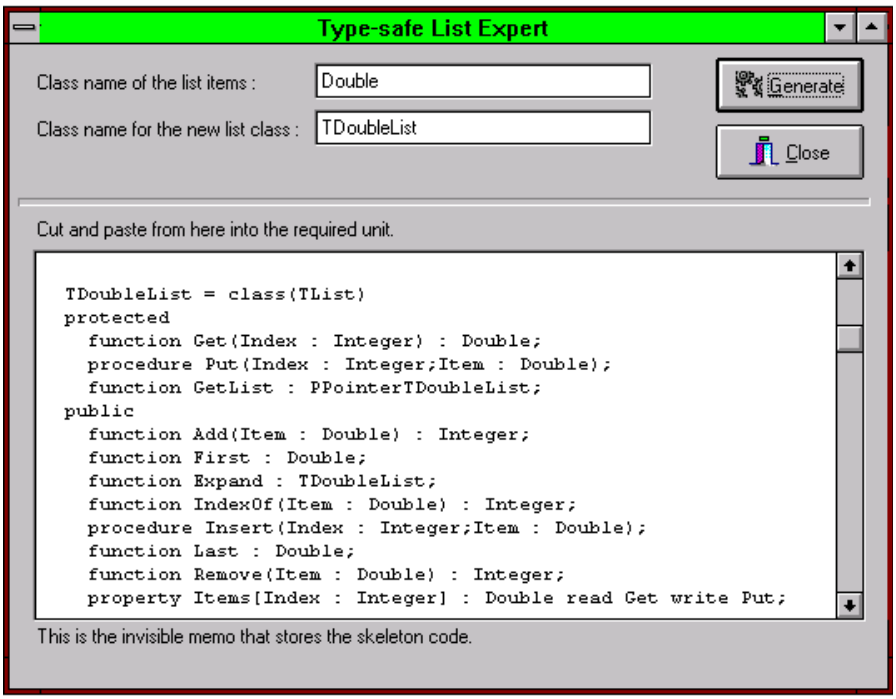
By the way, there is a class called TComponentList in DSGNINTF.PAS in your DOCS directory (it has a very small help file entry) that is a

type-safe list of TComponent objects. Borland used an alternative approach. They encapsulated a TList inside a new class and declared those methods and properties they wanted. It is not a very complete implementation and seems a less object-oriented approach, rather like encapsulating a VBX instead of inheriting from a Delphi control. It does have the advantage that if you want to hide some of the methods or properties of TList you can do it simply by not declaring them. In either case, you can add your own methods and properties to your heart's content.

Experts To The Rescue...

The procedure I have described is obviously very mechanical and is crying out for an expert to do the work. I have included a very simple one on the disk as LISTEXP.PAS, shown in Figure 2.

It has two edit boxes: one for the name of the new list class (eg TShapeList) and one for the type of item to store (eg TMyShape). Note that while the procedure is the same for inbuilt Delphi types like Double or Integer, the expert assumes the item type is a descendant of TObject. There are also two memo fields, one of which is invisible at run-time and stores the template of the code that will get generated. I did it this way to make it easy for you to change the generated code if you don't like my coding style. The other memo will contain the generated code after you press the Generate button. You can cut and paste this into the required unit. A neater solution would be to use the Delphi Tools interface to create a new unit, but (to quote the classics) I leave this as an exercise for the reader...



➤ *Figure 2: The type-safe list expert in action*

Jim Cooper works at Sybiz Software in Newbury, UK (site of the infamous bypass!) and can be contacted on CompuServe as 101641,440

➤ *Listing 5*

```
unit Mainform;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls,
  ExtCtrls, Shapes;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    ShapeRadioGroup: TRadioGroup;
    SizeEdit: TEdit;
    PaintBox: TPaintBox;
    AddBtn: TButton;
    Label2: TLabel;
    TopEdit: TEdit;
    Label3: TLabel;
    ColourComboBox: TComboBox;
    Left: TLabel;
    LeftEdit: TEdit;
    procedure AddBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    ShapeList : TShapeList;
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  {Ensure a colour is selected}
  ColourComboBox.ItemIndex := 0;
  {Create the list that will hold all the shapes to draw}
  ShapeList := TShapeList.Create;
end;
procedure TForm1.FormDestroy(Sender: TObject);
var i : Integer;
begin
  { Free the list and all the shapes it contains,
    because Free calls Destroy if ShapeList was
    successfully created, and TShapeList.Destroy now
    cleans up after itself.}
  ShapeList.Free;
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
  {Redraw all the shapes}
  ShapeList.DrawAllShapes(PaintBox.Canvas);
end;
procedure TForm1.AddBtnClick(Sender: TObject);
const
  Colours : array[0..2] of TColor =
    (clRed,clGreen,clBlue);
var
  NewShape : TMyShape;
begin
  {Create a new shape object based on the radio button
   selection}
  case ShapeRadioGroup.ItemIndex of
    0 : NewShape := TCircle.Create;
    1 : NewShape := TSquare.Create;
  end;
  {Set the properties of the new shape}
  NewShape.Size   := StrToInt(SizeEdit.Text);
  NewShape.Top    := StrToInt(TopEdit.Text);
  NewShape.Left   := StrToInt(LeftEdit.Text);
  NewShape.Colour := Colours[ColourComboBox.ItemIndex];
  {Add the shape to the list of shapes}
  ShapeList.Add(NewShape);
  {Redraw everything in the list, as the new shape may
   cover old ones}
  ShapeList.DrawAllShapes(PaintBox.Canvas);
end;
end.
```