Beating the System: Adding To Your Delphi Toolkit

by Dave Jewell

In the light of my renewed tussle with the disk formatting saga (see the panel below), I thought it would be nice to take a month out from "bleeding edge technology" and present the code for one or two straightforward useful Delphi classes which I use in my own software and which I'm sure you'll find useful too. Delphi supports reusability at the component level, at the form level, via units and (in Delphi 3) you can even group a bunch of components together and save them as a single reusable object. If, like me, you hate reinventing the wheel every time you start a new project, Delphi is without a doubt the best development system around!

A Delimited String Class

One particular "wheel" that comes round with monotonous regularity is the need to parse a string of delimited items. Comma-delimited strings are the norm, but you'll also find strings delimited with semicolons, Unix pipe characters (like this |) and virtually anything else you can think of. It's therefore advisable to write yourself a reusable parser class for chopping up a delimited string into a number of consecutive items. The code shown in Listing 1 is my own offering.

Once you've created an instance of a `TDelimitedString` object, you can pass a string to it using the `Text` property. This causes a private internal method, `ParseText` to be called which is responsible for chopping up the string into its constituent parts using the current value of the `Delimiter` property. This character property defaults to a comma since I suspect that this is the most common type of delimited string that you'll encounter. However, you can change the property to any other character that you want. Doing so will call the internal `SetDelimiter` method, causing the string to be re-parsed with the new delimiter character.

The various fields of the delimited string appear in the `Field` array property. If you've never used an array property before, now's a good time to get acquainted with this elegant Delphi feature. Array properties enable you to implement a custom property which appears to users of the class (remember, we're not necessarily talking about components, you can declare public properties of any class) as an array of `Integer`, `Char`, `String` or whatever. To implement an array property, all you have to do is use the special syntax shown for the `Field` property in Listing 1 and provide an internal method which takes a single `Index` parameter specifying which element of the array you want to retrieve. In this case, the `GetField` method validates the passed array index and returns the appropriate string from the internal `TStringList` variable. The only other property I haven't mentioned is `FieldCount`, which merely returns the number of fields in the array.

Another nice feature of Delphi Pascal (there are lots of them if you look!) is the ability to specify a particular property as the default property. This enables you to reference the default property of an object without specifying the property name. In this particular case, it means we can replace this:

```
Caption := PageList.Field[Idx];
```

where the `Field` property is explicitly referenced, with this:

```
Caption := PageList[Idx];
```

where the default property is implicitly referenced.

Of course, making use of this sort of language feature is sheer laziness on my part. I take refuge in the fact that Anders must be a kindred spirit or he wouldn't have put it

How Not To Design An Operating System: Revisited

Last month, I began with an apology. This month, I'd like to do the same, but this time round I'm not apologising for bugs in my own code, I'm apologising on behalf of Microsoft! You may remember that some time ago, I presented a set of 16-bit routines which (amongst other things) allowed you to read the serial number of a disk under Windows 3.1, Windows 95 and NT. I worked quite hard on this code, in order to ensure that it worked correctly on all three target platforms. However, since that time, I've heard from a number of readers who've had problems using the code under Windows for Workgroups (WFW), an operating system which I don't have installed and which I've never used to any great extent. It transpires that if you have 32-bit disk access enabled under WFW, then disk serial numbers aren't available! Sigh. It seems that the only way to get round this is to disable 32-bit disk access. I suppose it's possible that (under program control) one could turn off 32-bit disk access, do your serial number checking, and then restore the status quo. However, as I indicated to the Editor, I don't think I have sufficient strength left to investigate this approach! I simply mention it here so that you'll know not to send me any more bug reports when running the code under WFW. If you do, I may not be responsible for my actions...

into the language in the first place! As further evidence of my terminal sloth, you'll notice that I've added a second constructor to the class. Although not immediately obvious from the language specification, a class can have as many constructors as you want, subject to the obvious restriction that they all have a unique name. This means that we can implement a `CreateAssign` constructor to create an object instance and set the `Text` property at the same time. Sheer indulgence!

When using multiple constructors, it's possible to use one constructor to call another, which is what I've done here. The `CreateAssign` constructor calls the ordinary `Create` constructor to do most of the work and then finishes up by calling the `Assign` method. Although I could have just coded the nested call as `Create` rather than `Self.Create` it's good to use the latter form because it makes it clear that you're not trying to call the inherited constructor.

Maybe this technique of calling one constructor from another causes you to raise an eyebrow? Well, I've looked at the generated object code very closely and I believe there's no problem. Internally, Delphi uses a `Boolean` flag (passed on the stack as a hidden parameter to a constructor) to indicate whether or not a constructor should actually allocate the memory for a new object. When you call one constructor from another, this flag is zero, which effectively just turns the inner constructor into an ordinary subroutine of the outer one.

The code fragment in Listing 2 shows how easy it is to use the `TDelimitedString` class. In this case, the program assumes that Delphi 1 is installed on the system and uses our delimited string class to fill a listbox with the contents of the `System` page of the component palette. You can see the program running in Figure 1.

Yes, I realise that later versions of the `TStrings` class implement a `CommaText` property, which provides support for comma-delimited strings. However, I wanted to provide a class which was portable to 16-bit Delphi applications. Also, the `CommaText` property will (as the name suggests) only work with comma-separated strings and has

➤ *Listing 2*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Idx: Integer;
  IniFile: TIniFile;
  PageList: TDelimitedString;
begin
  IniFile := TIniFile.Create ('C:\WINDOWS\DELPHI.INI');
  try
    PageList := TDelimitedString.CreateAssign(
      IniFile.ReadString('COMPLIB.DCL.Palette', 'System', ''));
    try
      PageList.Delimiter := ';';
      for Idx := 0 to PageList.FieldCount - 1 do
        ListBox1.Items.Add (PageList [Idx]);
    finally
      PageList.Free;
    end;
  finally
    IniFile.Free;
  end;
end;
```

➤ *Listing 1*

```
unit Delimit;
interface
uses SysUtils, Classes;
type
  TDelimitedString = class (TObject)
  private
    fText: String;
    fList: TStringList;
    fDelimiter: Char;
    procedure ParseText;
    procedure SetText (const NewText: String);
    function GetFieldCount: Integer;
    function GetField (Index: Integer): String;
    procedure SetDelimiter (Delim: Char);
  public
    constructor Create;
    constructor CreateAssign (const NewText: String);
    destructor Destroy; override;
    property FieldCount: Integer read GetFieldCount;
    property Text: String read fText write SetText;
    property Delimiter: Char
      read fDelimiter write SetDelimiter;
    property Field [Index: Integer]: String
      read GetField; default;
  end;
implementation
constructor TDelimitedString.Create;
begin
  Inherited Create;
  fDelimiter := ',';
  fList := TStringList.Create;
end;
constructor TDelimitedString.CreateAssign(
  const NewText: String);
begin
  Self.Create;
  Text := NewText;
end;
destructor TDelimitedString.Destroy;
begin
  fList.Free;
  Inherited Destroy;
end;
procedure TDelimitedString.ParseText;
var
  i: Integer;
  buff: String;
begin
  fList.Clear;
  buff := Text;
  while Length (buff) > 0 do begin
    i := Pos (Delimiter, buff);
    if i = 0 then
      i := Length (buff) + 1;
    fList.Add (Copy (buff, 1, i - 1));
    Delete (buff, 1, i);
  end;
end;
procedure TDelimitedString.SetText (const NewText: String);
begin
  fText := NewText;
  ParseText;
end;
function TDelimitedString.GetFieldCount: Integer;
begin
  Result := fList.Count;
end;
function TDelimitedString.GetField (Index: Integer): String;
begin
  Result := '';
  if (Index >= 0) and (Index < fList.Count) then
    Result := fList.Strings [Index];
end;
procedure TDelimitedString.SetDelimiter (Delim: Char);
begin
  if fDelimiter <> Delim then begin
    fDelimiter := Delim;
    ParseText;
  end;
end;
end.
```

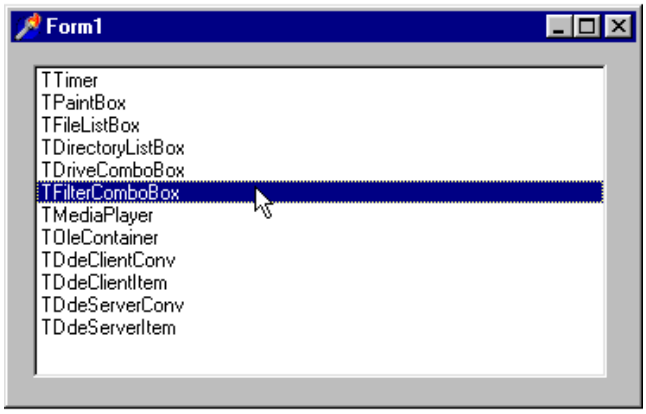
no facility for using other delimiter characters.

1066 And All That

Nowadays, many well-written applications maintain one or more history lists on behalf of the user. A history list enables users of your program to quickly specify a previously used choice, rather than having to type the whole thing in again. History lists are *very* extensively used by all versions of the Delphi IDE. For example, type `Ctrl-F` and you'll see your last few search strings in the `Find Text` dialog. History lists are generally (but not always) employed within the context of a `TComboBox` component which has its `Style` property set to `csDropDown`. This is important because it allows the user to enter new choices into the history list so that they'll appear next time round. History lists are all about remembering the custom word list entered by the user: a fixed list doesn't qualify.

When implementing a history list, we also need to limit the total number of items that are maintained in the list. For example, if the Delphi 1 IDE was to remember every single search string I've ever entered into the `Find Text` dialog, the storage involved might well represent a fair chunk of my hard disk by now! Not only that, it would be very cumbersome to select from such a large number of items whenever the history list appeared. In the implementation presented here, I've limited history lists to a maximum of 10 items, but this can easily be changed. As we shall see, implementation choices are a major factor here.

Another important aspect of history lists is the order in which items appear. If you open up the `Find Text` dialog, search for `MyList.Clear` and then close the dialog, you'll naturally expect to see the same text appear as the first, selected, item in the combobox next time you call up the dialog. What this means is that whatever choice was made last time round has to appear as the first choice next time round. This is true whether or not the previous choice was a new addition to the history list, or was a pre-existing item. From this you can see that maintaining a history list involves "freshening" choices by moving them to the top of the list. This is often referred to as an MRU list, because the items are stored in **M**ost **R**ecently **U**sed order. Similarly, when new items are added to the history list, the oldest item (that is, the item at the end of the list) will be automatically deleted if the list exceeds its maximum permitted size.



➤ *This sample program (Listing 2) uses the delimited string class to list the class names of the components on the System page of Delphi 1's component palette*

With all this in mind, take a look at the code in Listing 3. This code was written so as to be compatible with 16-bit and 32-bit Delphi, courtesy of a bit of unpleasantness with exit procedure handling! The unit relies on the previously mentioned delimited string unit and only exports two routines of its own: `GetHistoryList` and `AddHistoryString`. The idea is that you give a plain-English name to the history string such as `SearchItems`. You can then assign the `Items` property of a combobox as easily as this:

```
ComboBox1.Items:=
  GetHistoryList('SearchItems');
```

Similarly, if you're maintaining a list of previously opened files, you can assign to the `HistoryList` property of your `TOpenDialog` component with a single line of code:

```
OpenDialog.HistoryList :=
  GetHistoryList('OpenedFiles');
```

When you close a modal form, get a value of `True` back from the `Execute` method of your open dialog, or whatever, you can simply update the appropriate history string like this:

```
AddHistoryString('SearchItems',
  ComboBox1.Text);
```

or like this:

```
AddHistoryString('OpenedFiles',
  OpenDialog.FileName);
```

The beauty of this approach is that it's so simple. To add history lists to your application, you just need a couple of calls to `GetHistoryList` and `AddHistoryString` for each history list you implement.

So how does it work? The code is really quite straightforward. It uses a .INI file which it's assumed will reside in the same directory as your application. Additionally, it's assumed that the .INI file will have the same base name as the program itself. Thus, if the application is called WHIZZ.EXE the `History` unit will use a .INI file called WHIZZ.INI. By making these assumptions, the `History` unit can be made completely autonomous of the main application. Incidentally, if you think .INI files are old hat, you can easily replace all occurrences of `TIniFile` with `TRegIniFile` so that the code works with the registry instead. However, in my experience .INI files tend to be considerably more robust than the Windows 95 registry. Oops, did I say that?

The initialisation part of the unit calls `HistoryStartup` and arranges for the cleanup procedure, `HistoryShutdown`, to be called when the unit terminates. The `HistoryStartup` code, in turn, creates the `HLists` global which holds the names of all

the history lists and, in the Objects array, subsidiary TStringList objects which contain the contents of each individual history list. It then reads the History section of the .INI file into memory and parses each comma-delimited string item into a history list using the AddList subroutine. Unsurprisingly, AddList makes use of the delimited string unit to convert each comma-delimited list into a TStringList object.

You'll notice that for performance reasons the entire history list is preloaded into memory when the application starts running and isn't written back out to disk until the History unit is closed down, when the application terminates. The HistoryShutdown code completely erases the History section of the .INI file and takes care to write only those history lists which contain one or more items. That's because a new, empty, history list is created each time GetHistoryList is called for a list name which hasn't been encountered before. If no corresponding call to AddHistoryString takes place, you wind up with an empty history list which is excluded from the .INI file.

This only leaves the two interface routines, GetHistoryList and AddHistoryString. GetHistoryList simply looks up the specified list name in the "list of lists", returning the existing TStringList object if found. If the designated list isn't found, then it creates a new, empty list, adds it to the list of lists, and returns it to the caller. AddHistoryList takes care of "freshening" history list entries as I described earlier. If the required entry exists in the history list it's deleted and moved up to the top of the list, to indicate that it's the most recently used item. If the entry doesn't exist it's added, taking care to delete the least recently used item if the list reaches its maximum capacity.

Limitations

Because I've used .INI files, there are some implementation restrictions to consider. I'm not too sure that .INI files would work properly with line lengths greater than 255, and in any event 16-bit Delphi doesn't implement strings larger than this. Accordingly, I've limited the number of items in a history list to 10 and made the assumption that the total length of all the items in the list (plus commas, the keyname and the = character) doesn't exceed 255. This will probably be fine most of the time, but you will undoubtedly run into trouble if you store ten fully qualified pathnames in a single history list. To get around this, you might consider

➤ *Listing 3*

```
unit History;
interface
uses SysUtils, Classes, WinProcs, IniFiles;
function  GetHistoryList(const ListName: String):
  TStringList;
procedure AddHistoryString(const ListName, Str: String);
implementation
uses Delimit;
const MaxStrings = 10;
var   HLists: TStringList;
function GetHistoryList(const ListName: String): TStringList;
var Idx: Integer;
begin
  Idx := HLists.IndexOf(ListName);
  if Idx = -1 then
    Idx := HLists.AddObject(ListName, TStringList.Create);
  Result := TStringList(HLists.Objects [Idx]);
end;
procedure AddHistoryString(const ListName, Str: String);
var
  Idx: Integer;
  List: TStringList;
begin
  List := GetHistoryList(ListName);
  Idx := List.IndexOf(Str);
  if Idx <> -1 then
    List.Delete(Idx)
  else if List.Count = MaxStrings then
    List.Delete(MaxStrings - 1);
  List.Insert(0, Str);
end;
procedure HistoryShutdown; far;
var
  Str: String;
  List: TStringList;
  IniFile: TIniFile;
  Idx, Num: Integer;
begin
  IniFile :=
    TIniFile.Create(ChangeFileExt (ParamStr (0), '.INI'));
  try
    IniFile.EraseSection ('History');
    for Idx := 0 to HLists.Count - 1 do begin
      List := TStringList (HLists.Objects [Idx]);
      if List.Count > 0 then begin
        Str := '';
        for Num := 0 to List.Count - 1 do begin
          Str := Str + List.Strings [Num];
          if Num < List.Count - 1 then
            Str := Str + ',';
        end;
        IniFile.WriteString('History',
          HLists.Strings[Idx], Str);
      end;
      List.Free;
    end;
    HLists.Free;
  finally
    IniFile.Free;
  end;
end;
procedure HistoryStartup;
var
  Idx: Integer;
  IniFile: TIniFile;
  NamesList: TStringList;
  procedure AddList(const ListName: String);
  var
    Idx: Integer;
    NewList: TStringList;
    ds: TDelimitedString;
  begin
    ds := TDelimitedString.CreateAssign(
      IniFile.ReadString('History', ListName, ''));
    try
      if ds.FieldCount > 0 then begin
        NewList := TStringList.Create;
        for Idx := 0 to ds.FieldCount - 1 do
          NewList.Add(ds [Idx]);
        HLists.AddObject(ListName, NewList);
      end;
    finally
      ds.Free;
    end;
  end;
begin
  HLists := TStringList.Create;
  IniFile :=
    TIniFile.Create(ChangeFileExt(ParamStr (0), '.INI'));
  try
    NamesList := TStringList.Create;
    try
      IniFile.ReadSection('History', NamesList);
      for Idx := 0 to NamesList.Count - 1 do
       AddList(NamesList.Strings [Idx]);
    finally
      NamesList.Free;
    end;
  finally
    IniFile.Free;
  end;
end;
initialization
  HistoryStartup;
{$IFDEF WIN32}
  finalization
    HistoryShutdown;
{$ELSE}
  AddExitProc (HistoryShutdown);
{$ENDIF}
end.
```

```
[History_20]
Count=11
H0=c:\rx\demo\rxdemo\rxdemo.dpr
H1=c:\windows\desktop\rx\rx\demo\rxdemo\rxdemo.dpr
H2=c:\delphi\ektest.dpr
H3=c:\windows\desktop\anim\democurz.dpr
H4=c:\articles\delphi\prev\nemesis.dpr
H5=d:\nusurvey\nu.dpr
H6=c:\aol25i\download\compress\compdemo.dpr
H7=c:\delphi\demos\imagview\imagview.dpr
H8=c:\delphi\demos\scribble\scribble.dpr
H9=d:\nemesis\nemesis.dpr
H10=d:\nusurvey\brand\brand.dpr

[History_24]
Count=12
H0=c:\rx\units\rxtooreg.pas
H1=c:\rx\units\rxdbreg.pas
H2=c:\rx\units\rxctlreg.pas
H3=c:\windows\desktop\rx\rx\units\rxtooreg.pas
H4=c:\windows\desktop\rx\rx\units\rxdbreg.pas
H5=c:\windows\desktop\rx\rx\units\rxctlreg.pas
H6=c:\aol25i\download\compress\componly.dcu
H7=c:\aol25i\download\compress\compctrl.dcu
H8=c:\aol25i\download\compress\compress.dcu
H9=d:\nusurvey\anigif.pas
H10=d:\nusurvey\epcmpreg.pas
H11=d:\nusurvey\expbtn.pas
```

The Delphi 1 IDE uses a more complex two-level arrangement which allows individual items in a history list to be very long; by using TRegIniFile, you can avoid this complication

modifying the `History` unit so that it stores each history item on a different line of the .INI file. This was the approach taken by Borland when they implemented their own history unit for the 16-bit Delphi IDE.

If you look at Figure 2 (a partial dump of the 16-bit DELPHI.INI file), you can see how this works. Rather than referring to a history list by name, Borland refer to a history list using a number. Thus, the `ProjectsOpened` list might be assigned the number 20. This is mapped onto a section name of `History_20` beneath which can be found all the items for that particular list, one per line. I don't quite understand why the Borland code maintains a `Count` key for each list, since the `TIniFile.ReadSection` method will take care of returning all the items in a particular list. However, each to his own. The two-level Borland approach is more complex than my one-level scheme, but it can handle larger strings. In time honoured fashion, if you want to implement the Borland way of doing things, it's left as an exercise for the reader: it's not difficult to do. However, you can easily avoid the extra complexity and cope with very long strings simply by using `TRegIniFile` rather than `TIniFile` (the registry can handle much longer string entries than .INI files).

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave as DaveJewell@msn.com, DSJewell@aol.com or DaveJewell@compuserve.com