Turbocharging Database Applications With TLists

by Jonathan Morgan

By handling data records in memory data structures rather than placing the burden on a disk-based database, the speed of some applications can be significantly improved. Delphi provides us with a superb tool for handling groups of data records in memory: the `TList`. This article describes the basic construction and use of a `TList`. Then we will use some `TLists` along with supporting functions to create a local in-memory database.

TList Basics

The Delphi `TList` is essentially an array of pointers. However, `TLists` differ from normal Pascal arrays. For one thing they are dynamic, ie the number of pointers they hold can increase and decrease. Also, as objects, they come complete with methods for insertion, deletion and sorting.

When I first heard of `TLists` I thought they were going to be an abstract Delphi implementation of linked lists. Don't make the same mistake! As stated above, and as you will see below, `TLists` are far more akin to arrays.

As with other objects, `TLists` must be created and freed. A good place to do this is in the form create destroy methods, see Listing 1. The online help provides a good overview of the available `TList` properties and methods. However, here is a quick summary with a few extra comments for those interested in the inner workings:

- `Add(pointer)` places a pointer on to the end of a list. A pointer can be to an object, a method, a function or a variable. To add objects to a list use `aList.Add(aObject)` instructions. `TList` pointers will often be to instantiated objects and in this article our list of pointers will be to the same object type. `TLists` are not type-safe, but see Jim Cooper's article in Issue 9 if you wish to make them so. There is no technical reason, however, why pointers to different class types shouldn't exist in the same `TList` and sometimes it is very useful.
- `Capacity`: setting this property reserves or releases memory in the `TList`. It's not normally worth worrying about unless clock cycles are really important and you want a list to reserve memory in one hit. Never set the capacity to a value less than the number of items on the list.
- `Clear` takes all the pointers off a `TList` and sets the `Count` to 0. Clear also sets the capacity to 0, thereby releasing *most* of a list's memory. Don't clear the list after an `aList.Capacity := n` instruction!
- `Count` holds the number of items on the list and will always be less than or equal to `Capacity`.
- `Delete(Index)` takes a pointer off a list and moves all the higher indexed pointers down one position with a call to `System.Move` (which proves that `TLists` are arrays, not linked lists).
- `Insert(Index, Pointer)` as you might guess moves all the pointers from `Index` up one position then puts the new pointer in the gap.
- `Items` is the array of pointers itself. It is not often directly referenced, however, since the shorter `aList[n]` instruction is equivalent to `aList.items[n]`.
- `Free` will release the memory used by the list for its pointers. It will not release what the pointers point to, that is your responsibility.
- `IndexOf(pointer)` returns the list index where the pointer resides, or -1 if the pointer is not on the list. `IndexOf` will find the pointer's index with a sequential search through all the pointers in the list until a match is found. That search can be quite time consuming on large lists.
- `Pack` calls `Delete(Index)` on pointers which are equal to `nil`. It does not change the `Capacity` (and thereby reduce the memory used by the list) even though the name implies that it will. Also, pointers will rarely become `nil` unless your program specifically makes them so. Therefore, this function should rarely need to be used.
- `Remove(Pointer)` essentially does an `aList.Delete(IndexOf(apointer))` where you provide the pointer to delete.
- `Sort(TSortCompare)` will sort the list of pointers with a `Quicksort` function. Quicksort is an undocumented function in the `Classes` unit that can also be used for purposes outside of `TLists`. `TSortCompare` is a pointer to the function that you must provide to perform comparisons in the sort.

➤ *Listing 1: Creating and destroying a TList*

```
procedure TForm1.FormCreate(Sender: TObject );
begin
  aList := TList.Create
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  aList.Free
end;
```

Database Applications

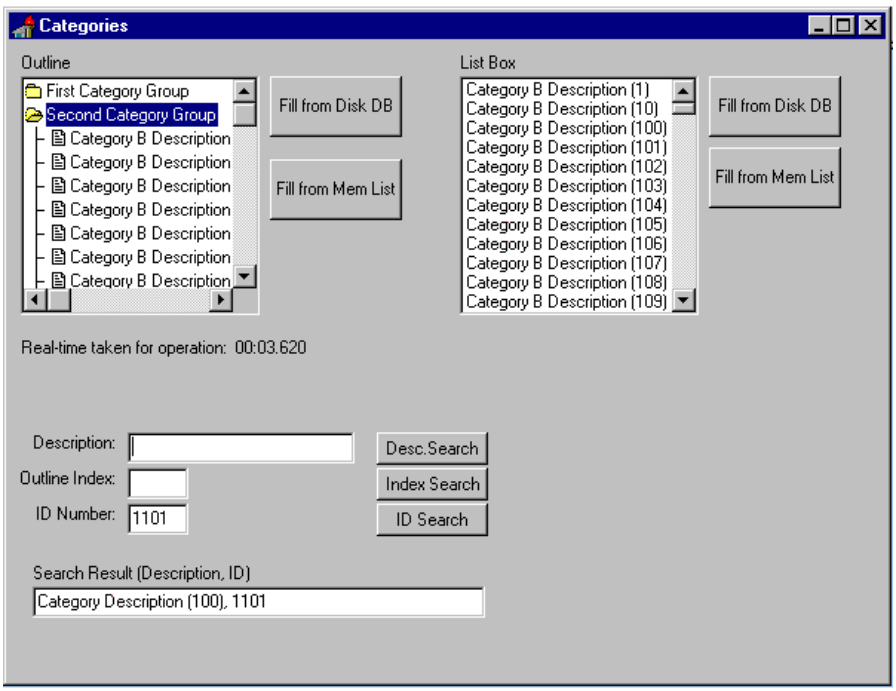
Now we know how TLists are constructed and manipulated, but how can we use them to turbocharge database applications? That depends on the application. The requirements of one project I worked on were to allow 40 operators to enter up to 50,000 contact records a month. Each contact could be assigned to one or more categories. There were 1,000 categories to chose from, presented in a standard outline component. The original approach was to draw the category outline by reading the entire category table from the database on each contact. Not surprisingly, drawing the outline simultaneously on even a few screens took several seconds. However, when it was found that the customer would rarely be updating the category table and that updates would not need to be immediately sent to the operators, we turned to TLists to hold the data in memory with the minimum of resource wastage. So, the category data only had to be read in once a day (on application startup) rather than once per contact. The effect was dramatic. As the project went on more and more "administration" type database tables were read into TLists on the application start-up and the screen response time became very fast.

So, when can and should we spend a little extra time to put a disk based table into a memory based TList? First, remember that once a table is loaded into memory, changes by another user to the original table will not automatically be reflected locally. That point is critical! Only store locally tables that either rarely get updated, or tables where a "flash shot" will suffice (eg in printing). Second, it is only worth doing the extra coding if the table records are required frequently or *en masse*. However, even if your application doesn't meet these requirements you should still find the way the demonstration program uses TLists a very useful insight into Delphi and into memory management.

The demonstration program for this article is loosely based around the categories problem above. In the demo we need to be able to display the categories in an outline in a pre-determined order and be able to put the categories in a listbox, ordered by description. Lastly we want to be able to retrieve records with a user specified ID, outline position, or description.

Name	Type	Comment
CategoryID	Numeric (8)	Unique id number
OutlinePosition	Numeric (8)	Outline index number
OutlineParent	Numeric (8)	Parent's outline index number
Description	VarChar (40)	

➤ *Figure 1: Category table*



➤ *Figure 2*

The first job is to give the category data table definition to the program. Listing 2 defines an object which matches the fields of the database table in Figure 1. The next job is to read the table's contents in. Listing 3 shows how to use SQL to read the database table into a TList upon form creation. All simple and straightforward so far, but how can we use the table now that it is in memory? We need to create a few functions, based on our requirements, to examine the TList. For speed we need at least two TLists, one that orders the records by their outline position and another that orders the records by their descriptions. Secondly we need a search routine to dig out records with a requested index value. The search routine could be very easily achieved (see Listing 4) with a small iterative routine. However, on this month's disk you will find that to make the search process faster I wrote some recursive binary search routines. Those routines required a third TList which ordered the records by their CategoryID values. To keep the three TLists up together I put them in a single container that controls how records are added to the lists. Rather than simply adding each record on to the end of each list the container makes sure that the records get inserted into the correctly ordered position. The screen shot from the demo (Figure 2) shows the program in action.

With very little effort we have created our own in-memory table.

Is It Worth It?

Some people might quite correctly be thinking that if the SQL database is good enough then it should be holding our "lists" of records in memory automatically. That is very true! However, by taking some of the database control into our own program we can achieve the following: minimal network traffic, minimal server loading, minimal memory requirements and maximum control over what is going on. Further, the speed gains can be very dramatic in both multi- and single-user environments. On my machine (120MHz Pentium with 32Mb RAM) with an Interbase database, to fill an outline with 1,500 records took SQL 2.4 seconds. That was with the database table fully loaded into memory. Loading the same outline from a pre-loaded `TList` took 0.33 seconds, one seventh of the time. That ratio could be vastly higher on a busy network. If you can make some parts of an application seven or more times more responsive then it is definitely worth it!

➤ *Listing 2: Category object*

```
{this class matches the fields in the Database table}
TCategory = class
  CategoryID,
  OutLinePosition,
  OutLineParent : Longint;
  Description : string[40];
end;
```

➤ *Listing 3: Reading a database table into a TList*

```
procedure TForm1.FormCreate(Sender: TObject);
var ACategory : TCategory;
begin
  {create the list}
  CategoryList := TList.Create;
  {then fill the list up with the records currently in the table}
  with Query1 do begin
    sql.clear;
    sql.add('SELECT CATEGORYID, OUTLINEPOSITION, OUTLINEPARENT, '+
            'DESCRIPTION FROM CATEGORIES');
    Open;
    while not eof do begin
      ACategory := TCategory.Create;
      with ACategory do begin
        CategoryID := Fields[0].AsInteger;
        OutlinePosition := Fields[1].AsInteger;
        OutlineParent := Fields[2].AsInteger;
        Description := Fields[3].AsString;
      end;
      CategoryList.Add(ACategory);
      Next;
    end;
    Close;
  end;
end;
```

➤ *Listing 4: Sequentially searching a list for a value*

```
function (SearchStr : String) : TCategory;
{assumes items in aList are valid pointers to TCategory objects}
var I: LongInt;
begin
  Result:=nil;
  I:=0;
  while (Result=nil) and (I<aList.Count) do
    if TCategory(aList[I]).Description=SearchStr then
      Result:= TCategory(aList[I])
    else Inc(I);
end;
```

➤ *Listing 5: Using polymorphism to free items*

```
procedure Tform.FormDestroy(Sender: TObject);
var I: LongInt;
begin
  for I:=0 to aList.Count-1 do
     TObject(aList[I]).Free;
  aList.Free
end;
```

A Few Final Hints

One way that we might consider storing records in a list is to set the list capacity to a very high number and then store a record at, for instance, its `CategoryID` position, ie

```
aList[aCategory.CategoryID] :=
  aCategory;
```

Whilst that method would work it is somewhat memory hungry at four bytes for every pointer (whether the pointer is used or not) and it would rarely be a good solution.

Sometimes you know that all the objects you put onto the `TList` need to be freed when the `TList` is freed. Those situations give us a chance to demonstrate a classic piece of polymorphism. The code in Listing 5 will correctly free all the objects in the list so long as all the pointers are to valid `TObject` descendants.

I sometimes hear the question, "Is there an equivalent to `TStringList`, but for integers?" The simple answer to that is no. However, with a little casting we can use `TLists`. The pointers in a `TList` need not point to anything at all so we can use the pointers to store numbers. To add a number use:

```
aList.add(Pointer(aLongInt));
```

To get a number use:

```
LongInt(aList[aIndex]);
```

Jonathan Morgan was last seen heading for Belgium to do some Delphi contract development and can be contacted by email on CompuServe 102247,2027