tObjectList: A real container for tObject descendants

I wrote this tool because of 3 areas where I felt Delphi was weak.
Delphi's containers are weak in two areas : Iteration and object disposal.
Delphi's stream support is un-documented. It's might be in there, but you have to do detective work. As far as I know, non component streaming is not supported.

A container should give ways to iterate it's content easily. None of Delphi's containers (tStrings, tStringList) have iterator functions.
As for disposal, an object container should free it's objects in it's own destructor. I'm not talking of GUI objects here, but of real world objects. By this I mean non GUI objects. Let's not forget that GUI is only a tool. The real work, the work we get paid for, is behind the GUI, and that is also OOP. If a Form owns a container full of non-GUI objects, the container should take care of disposing of it's content.

tStringList doesn't dispose of it's objects (it does free it's strings). This last problem would have been easy to fix if it had been the only one, but we still have the streaming problem.
As far as I can tell, tStringList's stream support is limited to it's string content, even if it can hold both objects and strings,

For those of you new to Pascal, tCollection was the container that came with BP7. I found tCollection to be very fast and easy to use. It supported easy streaming and it had very good iterator functions.

As to Streams, well if you don't have BP7 manuals you are out of luck because, for some reason, BI decided not to document them, except for what is in the help files, which is not enough to get starting.

I started out by looking at what was done in BP7 to achieve object persistence. You needed to register each object for which you needed stream support. For that registration, you needed to build a complex record giving information on the object and assigning a number. It was, over all, complicated, but it worked. Now with Delphi we have new functions. We can query an object to get its type name (the GetClassName method) and it's size.

This makes thing much easier. All we need to register is a tClass and 2 procedures: one that will write the object to a tstream and one that will read it back in. Here is the format of the RegisterClass procedure:

Procedure RegisterClass(const LoadProc,StoreProc:Pointer;Sender:tClass);

LoadProc & StoreProc are procedures that receive a single tStream parameter. **Note that the LoadProc is not a constructor, as it was in BP7.**

```
Procedure MyClass.Load(S:tStream);virtual;
begin
...
end;
```

Once an Object has been registered and added to your tObjectList container, you can stream it (or them) to disk, to memory or any medium as long as the stream descends from tStream. (With this method, I can store my objects in a BTrieve record!).

As you'll probably notice, most of the code in this file comes from the BP7 RTL and the Delphi RTL, namely the tCollection and tStrings classes. Some of the code was left unchanged, and some was modified to some degree to suit my needs. Also, please forgive my English as it is my

second language...

Send bug reports or suggestions to
Robert Daignault
LPL Soft
Compuserve 70302,1653

Streams

The Load and Store methods (names are examples, you choose the names) of an object should know how to
1- Write each of it's members in such a way that it will be capable of rebuilding itself in the Load method.
2-Rebuild itself by reading the stream in the same order it was written.
If you think that's easy, well sometimes it isn't. You'll often have pointers in there. Don't try and write pointers in a Stream, in won't work when you try and read it back.

Methods

These are the tObjectList methods and properties. If in doubt, check the source code.

Constructor Create;
Will create a standard tObjectList instance. Such an object will automatically grow, as needed with build in increments. It will also dispose of it's objects when destroyed.

Destructor Destroy;
You should not call the Destroy method of a tObjectList instance. Call the Free method instead. That's the standard way of disposing of an object.

function AddObject(Item: tObject): Integer; virtual;
AddObject will add Item to it's list at position Count-1. It will return the new count of items it now contains.

procedure Clear; virtual;
procedure Delete(Index: Integer);
Procedure DeleteAll;
If you call these methods directly, they will not Free the items it contains. They simply delete the pointer and compact the vector. Clear and DeleteAll are identical. If Index is out of bound, an exception will be raised.

Procedure FreeAll;
Procedure FreeAt(Index:Integer);
Procedure FreeObject(Item: tObject);
The Free methods first call the object' s Free method, thus disposing them, and then calls the Delete methods. FreeObject will raise an exception if Item cannot be found, Also if Index if out of bound in FreeAt, an exception will be raised.

Procedure FreeItem(AnItem:Pointer); virtual;
tObjectList.Free methods call FreeItem to dispose an Item. FreeItem calls AnItem.Free to dispose of AnItem. You can override FreeItem for special objects. (See also the DestroyObjects property to modify this behavior).

function IndexOf(Item: tObject): Integer;
Use IndexOf to know the position of Item in the list. The result could be used to retrieve an item by getting Item[Result] with the Item property.

procedure Insert(Index: Integer; Item: tObject); virtual;

Use to insert an object into a specific position.

procedure Move(CurIndex, NewIndex: Integer);
Use to move an object to a new position in the list.

procedure Pack;
Pack will compress the list. It will remove any NILL pointers by calling the Delete method for those positions that contain a nil pointer.

Streaming support

The following 6 methods provide streaming support for the tObjectList class.

Constructor CreateFromStream(const FileName: string);
Use only on a file that was created with the SaveToStream method.

Procedure SaveToStream(const FileName:String);
Use to save a tObjectList to a disk file. Use CreateFromStream to rebuild.

procedure LoadFromStream(const FileName: string);
LoadFromStream will ADD a tObjectList stored on disk to the calling instance.

procedure ReadData(S: TStream); virtual;
procedure WriteData(S: TStream); virtual;
These two methods are called to store/retrieve a tObjectList. You should not have to call these methods directly unless youre using a tMemoryStream. In that case, create the tMemoryStream instance and call ReadData or WriteData with the tMemoryStream as parameter. An exception will be raised if one of the contained objects is not registered.

procedure DefineProperties(Filer: TFiler); override;
Well the way I understand tFiler, tReader and tWriter, I think that's the way it should be done...
This has yet to be tested. Any help here would be appreciated.

Iteration support

For those of you coming from BP7, here are the highly used ForEach style iterators. For others, these will need a bit of explaining.

Procedure ForEach(Action:Pointer);
The Action parameter you pass ForEach must meet 3 conditions:
1- It must be a the adress of a procedure in the form of:
Procedure DoAction(AnObject:tObject); far;
The name is not important, you chose that.
2- It must be an embedded procedure. If you think that's to complicated for nothing, think again, It's far easier that way because you don't need global variables.
3- It must be declared **far** (Forgetting to declare the DoAction procedure as Far will very quickly GPF on you!).

ForEach will call DoAction with each of it's tObject. You can query AnObject to get it's type using it's ClassType method.

Function FirstThat(TestFunction:Pointer):tObject;
Function LastThat(TestFunction:Pointer):tObject;
Both these iterators will call the *TestFunction* with each of it's tObject until *TestFunction* returns **True**. At that point FirstThat and LastThat will exit with the tObject that caused TestFunction to return True. If TestFunction never returns true, FirstThat and LastThat will return NIL. FirstThat

iterates from Items[0] to Count-1. LastThat iterates from Count-1 downto 0;

Function First:tObject; virtual;
Function Last:tObject; virtual;
Function Next(Item:tObject; Forward:Boolean):tObject; virtual;

These are simple iterators. Their use should be pretty obvious. First and Last will return Nil when empty (No exception raised). Next will return the object following Item when Forward is TRUE (or Nil if at end) , and the preceding object when Forward is False (or Nil if at start). Next will raise an Exception if Item is not in it's list.

Properties

Count : Integer;

Use the count property when you need to how many objects are contained in the tObjectLister instance. Note that Count is read-only.

Items:[Index:Integer];

Use Items to access one of the tObject directly. If Index is out of range (that is if higher than Count-1, or <0), the tObjectLister will raise an exception of type EListError. For example, to access the third item in the tObjectList instance, use Items[2]

Capacity:Integer;

This is the current size of the Container, it's different than the Count property. Since tObjectLists will automatically grow, as required, you should seldom use this property, but it's there if you need it. For example, setting Capacity to 100 will grow the instance to 100 positions in one call. This is usefull while constructing a new instance if you know in advance how many objects it will contain.

DestroyObjects:Boolean;

By default, DestroyObjects is TRUE, thus all contained objects are destroyed when the instance is freed. Note that this property affects also all Free type methods. If FALSE FreeItem will not call the object's Free method.