Objetos Auxiliares III

Salvador Jover
s.jover@wanadoo.es

Las cadenas de caracteres van a ser una pieza clave de muchos de nuestros desarrollos... En ese punto, descubrimos la importancia que cobran las clases TStrings y TStringList. Y Delphi empieza a cumplir sus promesas... Lo vemos en el artículo.

¿Qué programador de Delphi, no ha usado, con mayor o menor frecuencia, los métodos que nos facilita la clase TStrings ...? Pregunta ingenua, ¿no, creéis?. Resultan tan básicos para la manipulación y almacenamiento de listas de cadenas que, a menudo, nos pasan inadvertidos. Y estamos tan habituados a ellos, que resulta difícil encontrar interés en describirlos, si no fuera quizás por la importancia de la función que desempeñan. En adelante nos referiremos a ellos como listas de Strings o listas de cadenas y nos centraremos a lo largo de este pequeño artículo en desvelar aquellos aspectos que nos puedan ser de mayor interés.

Como bibliografía, fieles siempre a nombrar aquellos textos sobre los que se documenta el artículo, encontramos nuestra querida y tantas veces necesitada Guía del Desarrollador, las fuentes de la VCL que acompañan a nuestro IDE, y en esta ocasión, incorporaremos algunos conceptos aportados por Steve Teixeira y Xavier Pacheco en su libro "Guía de Desarrollo Delphi 5". Este libro, editado por Prentice Hall, lo encontraréis recomendado en la Web del Grupo Albor y allí podréis obtener mayor información sobre el mismo. Un gran libro, sí señor... ;-)

Reflexionábamos en los dos artículos anteriores sobre las listas de punteros, representadas por la clase TList. Algunos autores las clasifican, junto a éstas que nos ocupan, como "clases de apoyo", - nosotros participábamos de la denominación que le da la misma Borland como Objetos Auxiliares- y el motivo no es otro que su participación en el desarrollo de otras clases. En ese aspecto, convendría ser recuperadas nuestras reflexiones al principio de la serie y que en cada capítulo intentamos justificar. Pero, si la clase TList es necesaria, a la postre nos resultará de vital importancia tener una clase especializada en el manejo de listas de Strings; y en ese punto, Borland, será fiel a las promesas de entregarnos un entorno rápido y sencillo, que nos facilite, como programadores, el desarrollo de nuestros proyectos. Idea ésta, que ya he compartido con vosotros en artículos anteriores.

Observemos:

Nos es necesario almacenar una lista de Strings y disponemos de un método, *SaveToFile( ),* con el que rápidamente serán creado un archivo de texto conteniéndolas... ¿fácil, no?

Nos es necesario cargarlas desde el archivo y la llamada al método *LoadFromFile*( ) lo hace de forma sencilla. ¿más aun?...

Nos es necesario incluir un nueva cadena, un nuevo String a nuestra lista y mira por donde, una simple llamada a *Add( )* producirá tan mágica proeza, de forma silenciosa y sin grandes aspavientos. Y la lista hoy puede estar representada en un TMemo, pero mañana ¿por qué no en una instancia de la clase TListBox?...Y de pronto caemos en la cuenta de algo que a primera vista nos pasa desapercibido y que vale la pena recalcar:

Objetos auxiliares III- Salvador Jover

"Estamos desarrollando una pequeña aplicación y hemos dispuesto sobre nuestro TForm varios componentes. Añadimos un componente TMemo que nos permitirá trabajar con un pequeño editor de texto, un representante de la clase TComboBox con el que cargar las fuentes disponibles por el editor. Sin representación visual, un TQuery y un TDataSource conectándonos a una fuente de datos..."

¿Qué pueden tener en común TMemo, TQuery o TComboBox? Interoperabilidad, entre otras cosas. Ni mas ni menos. El primer componente dispone de una propiedad, *Lines*, de tipo TStrings. El segundo también, pero en este caso su nombre es *SQL*, y nos permitirá ejecutar una instrucción en dicho lenguaje. En el tercero será la propiedad *Items* del objeto, conocida de sobra por todos nosotros. Los tres tienen en común una propiedad cuyo tipo es TStrings y que ha de permitir, interoperar entre ellos mediante dicha propiedad, permitiendo que el contenido de Memo1.Lines ser "asignado" a la lista representada en la propiedad Query1.SQL o que las distintas lineas de un TComboBox puedan devenir de la "asignación" por cualquier otro componente con una propiedad del mismo tipo. Se convierte así, en un interfaz común a todos los objetos que incorporan el uso y manipulación de listas de cadenas.

¿Solo cadenas de caracteres?... No, no... Además, se nos permite asociar a cada una de los Strings que componen dicha lista, una referencia a un objeto, a un TObject. El uso que se le quiera dar a gusto de cada cual. La Guía del Desarrollador, que en varias ocasiones hemos citado, nos comenta uno de los usos habituales y que está relacionado con la asociación de un mapa de bits a un String, pero que puede ser uno de los tantos usos que nos podamos imaginar.

Y ligado al concepto comentado de interoperabilidad entre componentes, un aspecto más sutil: dado que la introducción de la clase TStrings en la VCL se ha efectuado a un nivel de jerarquía muy alto, muy cercano a la raíz TObject, su incorporación como parte de innumerables objetos descendientes, hace compartir idénticos métodos, facilitando intuitivamente al desarrollador el conocimiento y uso de la VCL. En ese sentido, el programador que se acerca por vez primera a nuestro entorno se sorprende de que si quiero añadir un String en un memorandum, como así lo llama Borland, utilice el mismo método que para añadir una cadena de caracteres en un TListBox. Se convierte en algo de por si intuitivo. Nos acabamos por acostumbrar a dicho razonamiento: -No lo conozco pero seguro que si hago...

Y encima se cumple: existe el método y su uso es conocido por nosotros. Es parte del espíritu con el que se ha construido la VCL.

En ese sentido, podemos afirmar que Delphi cumple la promesa de minimizar el esfuerzo en cada uno de nuestros desarrollos, estandarizando métodos y evitando en muchos casos la ingrata tarea de adivinar, cual mago que consulta su mágica bola. Y si no, para muestra, un paseo por el API de Windows, tan distante de ese espíritu con el que está construida la VCL y tan propio de Microsoft.

Sin duda puede ser mejorable, no seré yo quien lo discuta. Sin embargo, sigue viva, evolucionando en cada nueva versión,, adaptándose a nuevas circunstancias y nuevos requerimientos.

TStrings y TStringList.

Volviendo a centrar el tema, manteníamos la necesidad de un objeto que nos permitiera trabajar con listas de cadenas de caracteres, respondiendo a una concreción del TDA lista sobre el tipo String. Pero el ánimo de crear ese interfaz común lleva a los creadores de la VCL a diseñar la clase TStrings como una clase en la que se definen básicamente métodos abstractos y virtuales. Los métodos abstractos no se implementan sino que tan solo se definen, dejando que sean sus descendientes los que lo hagan y puedan así particularizar y concretar dicha implementación. En el otro extremo, la conversión de un método estático en virtual permitirá que se puedan redefinir de ser necesario. Cualquier intento de hacerlo en TStrings hubiera sido contraproducente en aras de obtener la ansiada interoperabilidad. Así pues, podemos comprender la existencia de la clase TStringList, descendiente de TStrings, y que además de las capacidades que

posteriormente estudiaremos, heredadas de TStrings, permitirá la ordenación de las cadenas que componen la lista [¿recuerdan el método Sort( ) que pudimos curiosear en el artículo anterior y que afectaba a nuestro TList? Un buen momento para hacerlo ;-)], prohibir su duplicación o permitir la respuesta a un cambio en el contenido de la lista [evento OnChange y OnChanging].

Veamos la parte púbica en el interfaz de la clase TStrings ya que nos va a dar una idea de los métodos disponibles en la clase que posteriormente veremos:

```
  TStrings = class(TPersistent)
  ...
  public
    destructor Destroy; override;
    function Add(const S: string): Integer; virtual;
    function AddObject(const S: string; AObject: TObject): Integer; virtual;
    procedure Append(const S: string);
    procedure AddStrings(Strings: TStrings); virtual;
    procedure Assign(Source: TPersistent); override;
    procedure BeginUpdate;
    procedure Clear; virtual; abstract;
    procedure Delete(Index: Integer); virtual; abstract;
    procedure EndUpdate;
    function Equals(Strings: TStrings): Boolean;
    procedure Exchange(Index1, Index2: Integer); virtual;
    function GetText: PChar; virtual;
    function IndexOf(const S: string): Integer; virtual;
    function IndexOfName(const Name: string): Integer;
    function IndexOfObject(AObject: TObject): Integer;
    procedure Insert(Index: Integer; const S: string); virtual; abstract;
    procedure InsertObject(Index: Integer; const S: string; AObject: TObject);
    procedure LoadFromFile(const FileName: string); virtual;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure Move(CurIndex, NewIndex: Integer); virtual;
    procedure SaveToFile(const FileName: string); virtual;
    procedure SaveToStream(Stream: TStream); virtual;
    procedure SetText(Text: PChar); virtual;
  end;
```

Siguiendo los argumentos esgrimidos en los párrafos anteriores, métodos públicos como:

```
procedure Delete(Index: Integer); virtual; abstract;
```

, declarados como abstractos en TStrings, deberán obligatoriamente ser redeclarados en el descendiente, para nosotros, en este momento, la clase TStringList. De no hacerlo así, y tras ser creada una instancia de dicha clase, la invocación del método produciría una excepción.

Y aquí llegamos a uno de los aspectos que menos claros quedan tras la lectura de la Guía de Desarrollo de Delphi y que en mi opinión merecen un esfuerzo por comentar: Tras la introducción del concepto de lista de cadenas, nuestra Guía abre un apartado para comentar los aspectos de la creación de lista. Previamente, ya se nos ha comentado parte de lo observado hasta ahora, pero se hace por desgracia de forma parcial. Se nos ha dicho que la clase TStrings llegara a convertirse en un interfaz común a todas las instancias de clases que la utilizan pero en ningún lado se nos dice como... porque se da por sabido o adivinado... ;-)

Seguimos leyendo. ¿Y la creación?. La Guía del Desarrollador distingue dos tipos de creación según la duración del objeto. Para listas de cadenas con ámbito local, es decir cuya vida va a estar ligada a un procedimiento o a una función -para entendernos- se aconseja hacerlo del modo habitual, protegido su uso mediante un *try* y resguardado su liberación mediante un *finally*: Vale la pena verlo:

```
procedure Tform1.MiObjetoOnClick( Sender: TObject);
var
  ListaRapida: TStrings;
begin
```

```
  ListaRapida:= TStringList.Create;
  try
  //usamos la lista
  ListaRapida.Add('Soy la primera');
  finally
  ListaRapida.Free; //la liberamos finalmente
  end;
end;
```

Un momento... Parar ahí. ¿Por que la creación de la lista de cadenas la efectuamos mediante una llamada al constructor de su descendiente?. Aquí es donde está el meollo de la cuestión y sin embargo escasean las palabras. Debemos volver a la idea de que la clase TStrings ha sido creada como clase base, en cuyo interfaz se han declarado métodos abstractos que tendrán su implementación en sus descendientes y esta estrategia, permitida por el lenguaje, es la razón última de la creación del interfaz común y la interoperabilidad.

Para verlo con mayor claridad, necesitamos acudir a la misma VCL y asistir al uso de la clase TStrings como parte de otro componente. Fijémonos en la unidad "Stdctrls.pas". En esta unidad podemos tomar como ejemplo la clase TCustomMemo, antecedente del componente TMemo.

Veamos por un lado como se declara y construye la clase TCustomMemo:

```
TCustomMemo = class(TCustomEdit)
  private
    FLines: TStrings;
    FAlignment: TAlignment;
    FScrollBars: TScrollStyle;
  ...

implementación

constructor TCustomMemo.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Width := 185;
  Height := 89;
  AutoSize := False;
  FWordWrap := True;
  FWantReturns := True;
  FLines := TMemoStrings.Create;
  TMemoStrings(FLines).Memo := Self;
end;
  ...
```

Y por otro lado, el interfaz de la clase TMemoStrings, descendiente de TStrings, de cuya instancia nos hemos valido para crear FLines.

```
  TMemoStrings = class(TStrings)
  private
    Memo: TCustomMemo;  // de vital importancia //
  protected
    function Get(Index: Integer): string; override;
    function GetCount: Integer; override;
    function GetTextStr: string; override;
    procedure Put(Index: Integer; const S: string); override;
    procedure SetTextStr(const Value: string); override;
    procedure SetUpdateState(Updating: Boolean); override;
  public
    procedure Clear; override;
    procedure Delete(Index: Integer); override;
    procedure Insert(Index: Integer; const S: string); override;
  end;
```

Una conclusión por favor...

La conclusión no podía hacerse esperar. La instanciación de un objeto TMemoStrings a través de la invocación de su constructor, nos permite no solo que todos los objetos que manipulan listas de Strings puedan compartir el mismo Tipo (TStrings), sino que, la misma composición de la instancia de TMemoStrings incluirá un objeto de la clase TCustomMemo, cuyo puntero nos permite redefinir y completar la manipulación de las Lista en los métodos de la clase descendiente. No es casual la asignación "*TMemoStrings(FLines).Memo := Self;"* El campo Memo es definido como TCustomMemo y como tal, puede recibir un puntero a la clase que se está creando. Así pues, y por poner un ejemplo que nos resulte sencillo, la invocación del método Clear en la instancia de TMemoStrings, provocará la invocación del método Clear en nuestro TCustomMemo,

```
procedure TMemoStrings.Clear;
begin
  Memo.Clear;
end;
```

heredado de su ancestro TCustomEdit, ejecutando finalmente la rutina del API de Windows que cumple el cometido esperado "*SetWindowText(Handle, '');*"

Observad, por tanto, que la construcción de los objetos que se componen de una de estas Listas, ya sea un TListBox, un TComboBox, o un TMemo, etc..., estará ligada siempre al desarrollo de técnicas semejantes a la vista en el ejemplo anterior.

Estos conceptos se explican con cierto detalle en el libro de Texeira y Pacheco, y tienen cabida en un apartado que los autores dedican a los elementos claves de la VCL, que se agradece sin duda alguna. De hecho sus palabras advierten claramente cual es la situación:

"Debemos aclarar que, aunque la clase TStrings defina sus métodos, no implementa dichos métodos para manipular secuencias. La clase descendiente TStrings es la que realiza la implementación de estos métodos. Esto es importante para un diseñador de componentes ya que éste debe saber como ejecutar esta técnica tal como lo hacen los componentes Delphi. Cuando no se está seguro siempre es conveniente acudir al código fuente de la VCL para ver cómo Borland ejecuta estas técnicas."

Movimientos peligrosos...

¡Quien diría que podemos estar hablando de un nuevo ritmo que sacuda nuestro cuerpo! Nada más lejos de la realidad. Simplemente, nos haremos eco de una advertencia que nos hace tanto la Guía del Desarrollador, como Texeira y Pacheco en su libro. El título resulta llamativo, pero es bueno en este caso, llamar la atención de un pequeño detalle que puede tener consecuencias catastróficas o imprevisibles: olvidar que una referencia a un objeto no deja de ser un puntero.

Así pues podemos empeñarnos en intentar hacer la asignación:

MiListaString1:= MiOtraListaString;

donde ambos identificadores representan instancias de TStringList creadas, pensando que tras la asignación hemos copiado el contenido de MiOtraListaString en la primera... y no es así. Quedémonos con la **figura 1** donde lo observaremos claramente:

Como podemos apreciar, al efectuar la asignación, lo que finalmente obtenemos es que el identificador MiListaString1, como puntero que es y que no deja de ser, pese a ser transparente para nosotros, apunte hacia el nuevo objeto, perdiendo el original y quedando nuestro programa en condiciones inciertas que no garantizan en modo alguno resultados no erróneos en la ejecución.

Nuestra garantía es el uso del método Assign como norma general, que internamente, tras efectuar las comprobaciones de clase sobre el parámetro que recibe, garantiza que el contenido es copiado de forma correcta. Pero no nos quedemos tan solo en palabras y veámoslo con un ejemplo sencillo, que nos permita observar el problema con claridad. Para ello vamos a introducir algo de código en la pulsación de un TButton y reproduzcamos una situación similar:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  lista1, lista2: TStringList;
begin
  lista1:= TStringList.Create; //Creamos la primera lista
  lista2:= TStringList.Create; //Creamos la segunda lista
  try
    lista1.Add('Hola'); //Añadimos una cadena a la primera lista
    lista2.Add(Adiós);  //Añadimos una cadena a la segunda lista
    lista1:= lista2;    //Aquí hemos metido la pata...
    lista2.Add('Hola'); //Añadimos una nueva cadena a la segunda lista
    //Conclusión: la modificación de la segunda lista nos modifica la primera
    ShowMessage(IntToStr(lista1.Count)); //lista1.count nos devuelve el valor 2
  finally
    lista1.Free;
    lista2.Free; // Se genera la excepción... Algo
ha ido mal.
  end;
end;
```

Ha habido un uso incorrecto y la asignación, ha hecho que lista1 ahora apunte hacia el contenido de lista2, perdiendo el acceso a la memoria que había reservado cuando fue añadida la primera cadena. No solo eso, además, cualquier cambio en lista2 le afectará, como demuestra la ventana que nos muestra el valor "lista1.count = 2". Si cambiáis la sentencia de asignación por "*lista1.assign(lista2);*", podréis ver que el resultado es correcto, el proceso se hace con corrección y lista1.count nos devuelve el valor 1. Trasladar este ejemplo a una clase como TMemo, sobre la que hablábamos anteriormente, y encontraréis cuan fácilmente podéis perder el acceso a la memoria a la que apunta el campo FLines, y cuan imprevisibles, inciertos y difíciles de localizar pueden ser los errores posteriores.

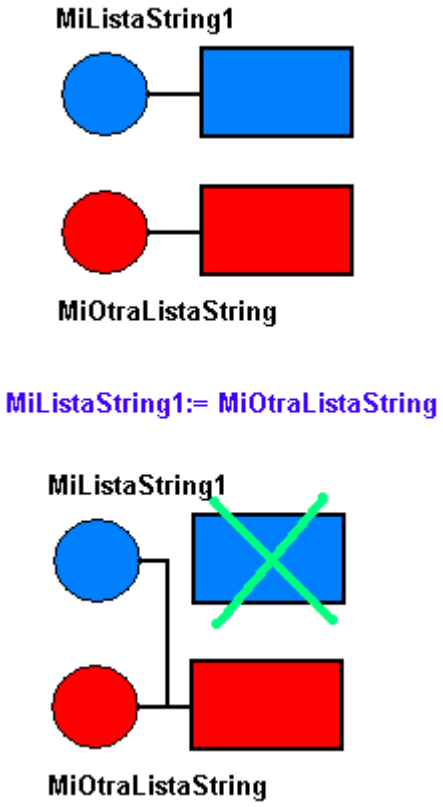

Figura1 Resultado de una asignación entre dos objetos

¿Qué puedo hacer con mi Lista...?

Si analizamos con detenimiento el interfaz que ha declarado TStrings en lineas anteriores, podremos observar que para operaciones de naturaleza similar a las que podíamos ver para la clase TList, se mantiene una nomenclatura similar, imponiendo coherencia y facilitando sin duda el aprendizaje y comprensión de los mismos. Al fin y al cabo, ambas se corresponden con el tipo de dato abstracto Lista. Lógicamente, la declaración de métodos de la clases TStrings y TStringList es más numerosa dado que nos ceñimos ahora a la manipulación de listas de cadenas de caracteres y existe mayor diversidad de operaciones posibles.

Nuestro recorrido se ajustará, básicamente, a la clase TStringList y juntos intentaremos que nuestro viaje sea productivo y ameno. Así que, la primera pregunta que me haría, toda vez que se ha creado con éxito mi lista, es: ¿Y ahora que...?. Veamos, ahora, que métodos nos permitirán añadir elementos a una lista, mientras quedará para una ocasión posterior la modificación o eliminación de dichos elementos:

- **Insertar elementos en nuestra lista.**

La operación mas sencilla que se nos puede ocurrir es añadir una cadena, un String, a nuestra lista, y para hacerlo, disponemos de dos métodos a los que estamos sobradamente acostumbrados: Add( ) e Insert( ) , y la elección de uno o de otro dependerá tan solo de que queramos hacer la inserción sobre el último elemento de la lista o bien en una posición determinada de la misma. Veamos unos ejemplos cualquiera:

```
Var
cadena: String;
begin
Lista.Add('He añadido mi primer cadena');
cadena:= 'Voy a añadir la segunda');
Lista.Add(cadena);
Lista.Insert(0, 'Esta la intercalo...');
    ...
```

Veamos los implementación que se hace en ambas funciones, pero antes comentaremos el significado que tiene declarar el parámetro como constante, ya que es utilizado por prácticamente todos los procedimientos y funciones que veremos. Si seguimos las palabras de la Guía de Object Pascal, llegaremos al concepto de Parámetros constantes, es decir aquellos que se acompañan de la palabra reservada **const**. El uso de esta palabra permitirá al compilador una optimización de código y su uso es similar a la consideración de Parámetro de Valor, salvo que al ser considerados como constantes, su valor no podrán ser modificado. De ahí que pueda ser optimizado por el compilador.

Podemos empezar con la observación de como se implementa el método Add::

```
function TStringList.Add(const S: string): Integer;
begin
  if not Sorted then
    Result := FCount
  else
    if Find(S, Result) then
      case Duplicates of
        dupIgnore: Exit;
        dupError: Error(@SDuplicateString, 0);
      end;
  InsertItem(Result, S);
end;
```

El valor de retorno de Add, de tipo entero, se corresponde a la posición del elemento que se ha añadido dentro de la lista, que empieza a contar su primer elemento desde 0. En este punto resulta imprescindible que se acuda a los razonamientos que hacíamos en los dos artículos anteriores, que nos servirán para entender el significado de dicho indice dentro de una propiedad matricial heredada de TStrings, como Strings. Los puntos comunes aparecerán tan pronto como relacionemos el puntero PStringItemList hacia una estructura matricial cuyo tipo es TStringItem (ver el interfaz), con el puntero PPointerList cuando estudiábamos la clase TList. Nos serviremos de similares razonamientos para entender la relación existente entre la propiedad matricial Strings con el puntero PStringItemList.

La función recibirá como parámetro constante una cadena de caracteres de tipo String. Comentábamos anteriormente que la clase TStringList permitirá la ordenación de las cadenas siguiendo un criterio alfabético, y por lo tanto, cuando es añadida cualquier cadena a la lista lo primero que habrá que comprobar es que el usuario haya decidido mantener activado el ordenamiento. Si *Sorted*, de tipo booleano, devolviera

f*alse*, es decir, si los elementos de la lista no están ordenados, podemos devolver como índice el valor del campo FCount, que representa al igual que en TList, la posición del ultimo elemento mas 1 (la posición en que se deberá insertar la cadena).

En caso contrario, suponemos que la lista esta ordenada y aquí se presentan varios casos según hayamos optado por permitir que se puedan duplicar cadenas o no, o bien que queramos que se genere una excepción en caso de intentar añadir una cadena a la lista.

```
  if Find(S, Result) then
```

El método Find es una función cuyo valor de retorno será si ha tenido éxito o no la búsqueda de la cadena en la lista y de haber tenido éxito, dato que Result es pasado por referencia, tomará como valor la posición del indice de la matriz en la que la búsqueda ha tenido éxito. No obstante, el hecho de tener éxito, obliga a una última evaluación:

```
    case Duplicates of
      dupIgnore: Exit;
      dupError: Error(@SDuplicateString, 0);
    end;
    InsertItem(Result, S);
```

Si debemos ignorar los duplicados (dupIgnore), saldremos de la función sin insertar la cadena, ya que esta ya existe en la lista. Si debemos generar la excepción (dupError) también. Y tan solo en caso contrario: que la lista no este ordenada o que si lo está el valor de la variable enumerada Duplicates sea dupAccept, será invocado el método InsertItem( ).

Por otro lado y de una forma similar:

```
procedure TStringList.Insert(Index: Integer; const S: string);
begin
  if Sorted then Error(@SSortedListError, 0);
  if (Index < 0) or (Index > FCount) then Error(@SListIndexError, Index);
  InsertItem(Index, S);
end;
```

Cuando es invocado este procedimiento es para insertar un string en una determinada posición de la lista, que viene marcada por el parámetro Index de tipo Entero. Hay que tener en cuenta, que este método no puede ser invocado en el caso de estar activo el ordenamiento automático, que nos marca la propiedad booleana Sorted. En dicho caso se generará una excepción.

De no estar activado el orden, la rutina siguiente nos protegerá frente a los errores de rango, evitando que podamos elegir un valor mayor que FCount (recordamos que FCount representaba la posición del último elemento mas 1) y menor que 0, que representa al primer elemento de la lista.

Si todo ha ido correcto, la invocación del procedimiento InsertItem( ) lo da por finalizado.

```
procedure TStringList.InsertItem(Index: Integer; const S: string);
begin
  Changing;
  if FCount = FCapacity then Grow;
  if Index < FCount then
    System.Move(FList^[Index], FList^[Index + 1],
      (FCount - Index) * SizeOf(TStringItem));
  with FList^[Index] do
  begin
    Pointer(FString) := nil;
    FObject := nil;
    FString := S;
  end;
  Inc(FCount);
  Changed;
end;
```

En ambos casos, tanto con la Función Add como con el procedimiento Insert, llamamos finalmente al método InsertItem( ) que será el que realmente hace la faena sucia, digámoslo así. Aunque puedo ser pesado, vuelve a ser necesario releer el primero de los artículos. La mecánica que se seguía para ajustar el tamaño del vector dinámico, las llamadas a Grow y el desplazamiento del vector mediante Sistem.Move son idénticos por lo que no es preciso profundizar en ello y me remito de nuevo a su lectura.

Podemos resaltar varias aspectos que pueden resultar de interés:

Las invocaciones de los métodos Changing y Changed, al principio y al final del procedimiento, llaman a los gestores de eventos respectivos, dándonos la oportunidad como usuarios del componente de implementar código en respuesta de los mismos.

```
procedure TStringList.Changed;
begin
  if (FUpdateCount = 0) and Assigned(FOnChange) then FOnChange(Self);
end;

procedure TStringList.Changing;
begin
  if (FUpdateCount = 0) and Assigned(FOnChanging) then FOnChanging(Self);
end;
```

Una vez, insertado el nuevo elemento a la lista, se procede a incrementar el contador FCount.

¡Y los objetos que...!

Decíamos que una instancia de la clase TStringList nos serviría para asociar un objeto a cada una de las cadenas de nuestra lista, pero aquí surgen detalles que debemos conocer y que a simple vista no se perciben. Es responsabilidad del programador el objeto en sí, asociado a cada una de las cadenas, ya que la responsabilidad de la lista acaba en la referencia al mismo. Si la lista es liberada, deberían previamente destruirse aquellos objetos que carezcan de sentido fuera de ella, tal y como lo hacíamos con la clase TList. Nuestra instancia tan solo almacena una referencia al objeto, un puntero, dentro de la estructura declarada en sección de tipos de la interfaz.

Vale la pena verla y darse cuenta que en FObject podremos guardar un puntero hacia nuestro objeto, lógicamente asociado al string FString, pues son parte del mismo registro, del mismo *Record*:

```
  const
  MaxListSize = Maxint div 16;
  type
  PStringItem = ^TStringItem;

    TStringItem = record
      FString: string;
      FObject: TObject;
    end;

  PStringItemList = ^TStringItemList;
  TStringItemList = array[0..MaxListSize] of TStringItem;
```

Con respecto al tema de la inserción, adelantaremos que nuestros objetos no tendrán vida fuera de la cadena a la que están unidos, por lo que, si estudiamos con detenimiento los métodos que nos permiten añadir objetos, InsertObject( ) y AddObject( ), heredados de TStrings, veremos que precisamente se valen de las cadenas para buscar un posicionamiento dentro de la matriz. Una vez obtenido dicho indice se procede a añadir la referencia al objeto a la lista en dicha posición. (Ver: PutObject(Index: Integer; Aobject:TObject;).

La diferencia entre ambos métodos, la podremos desprender de su misma nomenclatura y que viene a coincidir, lógicamente, con la misma distinción establecida entre Insert y Add, vista en lineas anteriores y que me parece no vale la pena incidir más.

Un respiro, por favor... conclusiones finales.

Nos tomamos un respiro. Es tan solo un alto en el camino que nos permitirá asimilar cuanto hemos compartido. ;-)

Pero no se acaba aquí el camino pues nos queda todavía mucha travesía que recorrer. Nos queda acabar de husmear en lo que es ahora nuestro centro de atención: la clase TStringList, ver aquellos métodos más habituales de manipulación de strings, aquellos que me puedan permitir obtener valores, o eliminarlos, aquellas propiedades que puedo utilizar para recorrer la matriz. ¿Sabíais que las propiedades ***Names*** y ***Values*** de la clase TStrings os permiten trabajar en con el formato ***Nombre=Valor*** propio de los archivos de configuración?. ¿O que la propiedad ***Text*** os devolverá en un ***string*** la concatenación de los strings de la lista mediante #13 y #10?...

Nos queda en definitiva, y si vosotros queréis, otro peldaño que subir y otro camino que andar. Nos vemos en el siguiente número de Síntesis,

Un saludo y hasta pronto.