Salvador Jover – mailto:s.jover@wanadoo.es

Objetos Auxiliares IV

Vamos a iniciar el penúltimo capítulo de esta pequeña serie. Andamos pues, muy cerca de su conclusión...

Creo que es un buen momento para volver la vista atrás, y lo vamos a hacer brevemente, en el inicio de este artículo, para todos aquellos amigos que se puedan haber incorporado en estos momentos y a los que, recomendaría iniciaran esta lectura con cautela, y con la penitencia de volver al primer número de la serie:

El primer artículo, artículo de introducción, nos sirvió para presentar una vista global de todos aquellos objetos que iban a ser considerados por nosotros. Era el tiempo de las primeras reflexiones sobre el concepto de Objeto Auxiliar, que iba a dar nombre a la serie. Era el numero 3 de la revista Síntesis.

Ha llovido mucho desde aquel momento, ¿no os parece?. Intentábamos explicar los motivos, las razones, que podían justificar iniciarla y nos extendíamos sobre las relaciones jerárquicas que se establecían entre dichos objetos; y allá quedaron las figuras de las primeras páginas, donde se describían dichas relaciones. Fueron páginas que nos sirvieron para romper el hielo y conocernos un poco más. Inmediatamente, nos pusimos el buzo de trabajo y comenzamos nuestra andadura sobre la clase TList, y en ello se nos fue un interminable artículo. A la conclusión del mismo, prácticamente habíamos abordado dicha clase, pero aun nos restaría otro más para finalizarla. Habíamos dejado de lado en un primer momento, cómo se había implementado el ordenamiento en las listas de punteros, algo menos conocido y que precisaba más detenimiento. Era el método *Sort(*), motivo del segundo capitulo de la serie.

Además, queríamos ir un poco más lejos, y buscamos nuevos métodos incorporados en Delphi 6 para la clase TList, *Assign( )*, que nos ayudaría a relacionar dos listas de punteros. Y como complemento a todo esto, se acompañó dicho número de un par de ejemplos de código fuente, que servirían para culminar, por fin, la clase TList.

Con la llegada del artículo que precede a éste, el número 3 de la serie, observamos que todo el contenido era realmente básico, y que a menudo, es utilizado por nosotros sin percibirlo. Son estructuras fundamentales dentro de la programación en general, como pueden ser una lista de punteros, representadas en la clase TList, o especializaciones de ésta, bien en el manejo de cadenas de caracteres, como puede ser la clase TStrings o TStringList, o de otros objetos, como TObjectList o TComponentList. La serie no lo era todo en si misma y nos apoyábamos en los artículos de otros compañeros, con su permiso y colaboración, para abordar tan extenso surtido de objetos. Además, y será motivo para el último capitulo que cierra esta pequeña serie, abordaremos el análisis de las clases que representan estructuras tan básicas como las "pilas" o las "colas".

Así pues, paralelo a nuestro avance, en el número 4 de Síntesis, **Tavo Ibaceta**, se sumergía dentro de las listas especializadas en componentes, como TComponentsList, dentro de su magnifica serie de artículos sobre creación de ventanas: *"Ventanas – Reflexiones desordenadas III"*

De igual forma sucedía con los objetos auxiliares relacionados con la introducción, modificación y borrado de datos en el registro de Windows y ficheros Ini, tan conocidos por nosotros. Nuestro compañero, **Carlos Conca**, se hacia eco de ello en su artículo: *"¿Ficheros de configuración o Base de Registros? III",* cuya lectura no puede ser menos interesante. Hablamos también del número 4 de Síntesis.

Ambos artículos, por su calidad, van a enriquecer el contenido de esta serie, que por razones de extensión, no podría abarcar por si sola tan numerosas clases sin evitar el bostezo del lector.

También es un buen momento para los que quieran introducirse en C++, avanzar en paralelo con la serie iniciada por **Mario Rodríguez** sobre la STL, y encontrar así puntos de encuentro con los conceptos desarrollados. El número de dos de Síntesis nos dejaba un artículo sobre el Contenedor List. El número tres hacía lo propio con el Contenedor Vector.

Para finalizar, dejamos también un promesa de abordar, en mejor ocasión, aquellos objetos auxiliares descendientes de la clase TStream, relacionados directamente con el almacenamiento, y manipulación de flujos de bits, cuyos métodos tan populares y prácticos, afectaban a una gran cantidad de componentes. Fue postergado para poder ser visto desde un ángulo mayor y que abordara otros conceptos mas generales. De momento queda en el aire, y será retomado por mí o por cualquiera de los que intentamos aportar un pequeño grano de arena en este proyecto que es el grupo Albor. Y ya que hablamos del proyecto, me perdonaréis si desde aquí os animo a participar en él, colaborando, ya desde los foros; ya desde la Web, haciendo uso de ella y ayudándonos a llenarla de contenido; ya desde estas páginas, compartiendo con tantos compañeros vuestra experiencia.

Pero no nos quedemos aquí y veamos en que punto quedamos...

Resolviendo las cuentas pendientes...

Al finalizar el articulo anterior, disponíamos de los métodos necesarios para insertar elementos en nuestra lista, bien al final de la misma, bien en una posición determinada. Hacíamos hincapié, y lo volvemos a hacer de nuevo, en que nos era necesario una lectura del primer artículo de la serie sobre la clase TList, para comprender los mecanismos empleados en la reserva de memoria, en el método seguido para ajustar el tamaño del vector dinámico, que lógicamente, por evitar una innecesaria repetición y por dar mayor dinamismo a la serie se omitían. En aquel artículo se intentaba seguir paso por paso, las lineas principales de las fuentes, comentando aquellos aspectos que debían ser resaltados.

- Manipulemos la lista...

Se nos puede ocurrir, dado que disponemos de cadenas de texto insertadas en nuestro objeto TStringList, decidir en buena hora, que una de ellas no merece gozar de la dignidad de estar allí, con las demás. Es decir, nos vamos a disponer a eliminar una de las entradas hechas y para ello, tan solo nos hace falta saber, el lugar que ocupa la misma. Ese será nuestro parámetro de entrada al método Delete( ).

Veamos la implementación que se hace del método:

```
procedure TStringList.Delete(Index: Integer);
begin
  if (Index < 0) or (Index >= FCount) then Error(@SListIndexError, Index);
  Changing;
  Finalize(FList^[Index]);
  Dec(FCount);
  if Index < FCount then
    System.Move(FList^[Index + 1], FList^[Index],
      (FCount - Index) * SizeOf(TStringItem));
  Changed;
end;
```

El procedimiento, recibe como parámetro el índice que ocupa la cadena en el vector. Recordemos que dicho vector inicia su primer elemento en 0 (cero) y finaliza con el elemento (*Count* – 1). Esto es

importante. Cualquier método empleado para recorrer dicho vector se podrá apoyar en ambos valores numéricos para evitar salir del rango del vector, que es uno de los primeros errores típicos del programador que se inicia. Es lo que se suele llamar por algunos autores como errores de cercado y que nos llevan a una pregunta mas que típica:

En un ARRAY[0..100] of TPersonas, ¿Cuantas personas conviven?...

Si respondisteis 101 habréis sobrevivido a un error de cercado.

Ahora bien, si la pregunta es: Nuestro objeto de la clase TStringList nos devuelve Count = 100... ¿cual es el último elemento de la misma? Ya no está tan claro... cualquier individuo algo impulsivo se habría apresurado a manifestar rápidamente: "-Pues claro, el 100. Es lógico hay cien elementos"...

Nuestro amigo impulsivo, generará al intentar acceder a dicha posición, una hermosa y brillante excepción.

Volviendo al procedimiento, observamos que primero se comprueba que nuestro parámetro está en el rango del vector, para en caso contrario, abortar el procedimiento con una excepción. Justo lo que pretendíamos resaltar:

```
if (Index < 0) or (Index >= FCount) then Error(@SListIndexError, Index);
```

En la llamada al método Changing damos una oportunidad al gestor de evento tal y como comentábamos en el número anterior, facilitándonos, que podamos hacer aquellas acciones que creamos oportunas antes de que cambie la lista.

```
Finalize(FList^[Index]);
```

Esta rutina que se declara en el modulo System, nos ayudará a liberar la memoria asociada a la cadena que deseamos eliminar. Y seguidamente, tan solo nos queda ajustar el vector, disminuyendo en una unidad el recuento de los elementos Dec(FCount) y trasladar cualquier posición posterior a dicho índice una posición menos, reagrupando el vector y eliminando el hueco dejado por la cadena eliminada.

```
System.Move(FList^[Index + 1], FList^[Index],(FCount - Index) * SizeOf(TStringItem));
```

Nos queda tan solo hacer la llamada para que se pueda ejecutar el evento OnChanged, toda vez que nuestra lista ya ha cambiando, volviendo a dar una oportunidad a nuestro usuario para que pueda tomar las medidas que crea necesarias.

Pero si os dais cuenta de una cosa... hablábamos en nuestro número anterior como nuestra lista de cadenas era capaz de asociar un objeto a cada una de las cadenas. Creo que me comprendéis. No se hace referencia a nuestros objetos en el código que hemos visto, por lo que, de haber existido asociado al elemento que se iba a eliminar, hubiéramos debido liberarlo previo a la llamada al método Delete( ).

Es decir, necesitamos tener la seguridad que dicho objeto se ha desligado de nuestra lista, previo a la eliminación de la cadena asociada al mismo. Y se hace necesario, lógicamente y por el mismo motivo, un bucle que recorra cada uno de los elementos, y proceda a dicha operación, de tratarse de un borrado de naturaleza global.

Que se de el error, en caso contrario, es tan solo cuestión de tiempo...

Parecida situación y semejante código es el que se implementa en los métodos Clear( ) y Destroy( ), que llegados al punto actual, estamos obligados a observar.

Empecemos por el método Clear:

Para "limpiar" una lista de elementos, es decir, eliminar todos y cada uno de los elementos, invocaremos al método Clear, que, por lógica, no recibe parámetro alguno:

```
procedure TStringList.Clear;
begin
  if FCount <> 0 then
  begin
    Changing;
    Finalize(FList^[0], FCount);
    FCount := 0;
    SetCapacity(0);
    Changed;
  end;
end;
```

Como en anteriores métodos, no existe método mejor para optimizar un bucle que no recorrerlo y tan solo deberemos ejecutar el bloque de rutinas en el caso de que el contador de elementos, tenga alguno. Por dicha razón, evaluamos previamente la condición, y seguidamente efectuamos parecidas operaciones al método de borrado, con la excepción de que ya no es necesaria la llamada al método *System.Move,* que se ocupaba de desplazar el vector de cadenas.

Finalize(FList^[0], FCount);

Ha de realizar similar operación salvo que en esta ocasión *FCount,*da lugar a que se libere la memoria asociada a todo el vector y no solo a uno de los elementos, fijando la nueva capacidad a 0.

```
SetCapacity(0)
```

Recordemos que el método *SetCapacity( )* era implementado mediante una llamada a la función ReallocMem( ). En este caso, siendo FList^[0] distinto de nil, primer parámetro de la función, y 0 el valor que retornaba *NewCapacity * SizeOf(TStringItem),* segundo parámetro, ReallocMem dispondría del bloque que referencia FList^[0] y fijaría su valor a Nil, eliminando el puntero.

```
procedure TStringList.SetCapacity(NewCapacity: Integer);
begin
  ReallocMem(FList, NewCapacity * SizeOf(TStringItem));
  FCapacity := NewCapacity;
end;
```

También estamos en condiciones de analizar el destructor de nuestra lista de cadenas:

```
destructor TStringList.Destroy;
begin
  FOnChange := nil;
  FOnChanging := nil;
  inherited Destroy;
  if FCount <> 0 then Finalize(FList^[0], FCount);
  FCount := 0;
  SetCapacity(0);
end;
```

Como era de esperar, la invocación de este método será necesaria al destruir definitivamente nuestro objeto TStringList. Lo primero que debemos hacer es romper cualquier relación del objeto con la implementación hecha por el usuario en los eventos a que hace referencias las variables FOnChange o FOnChanging, asignando ambas variables a **nil**. Seguidamente, y aunque rompe el esquema usual de cualquier destructor, es llamado el destructor heredado, que destruirá su parte del objeto, quedando tan solo por destruir el vector de elementos y que se hará en las mismas condiciones a las vistas en el procedimiento anterior.

- Métodos de búsquedas en listas de cadenas

Objetos Auxiliares IV

Podemos necesitar, en algún momento concreto, obtener cierta información de un objeto de la clase TStringList. En muchos casos, estará disponible para nuestro uso, en las propiedades publicadas del objeto, es decir, aquellas que podemos acceder en tiempo de diseño mediante el inspector de objetos, o bien mediante aquellos métodos públicos declarados. Este es el caso dos funciones: Find( ) e IndexOf( ), de las que obtendremos en tiempo de ejecución, bien la existencia y el índice de una cadena a añadir, bien el índice asociado a la cadena en listas ordenadas/no ordenadas. Un poco liado para algo tan sencillo. Veámoslo con mayor detenimiento:

Comentábamos al principio del articulo anterior, que la clase TStringList, era descendiente de TStrings, a la que extendía nuevas funcionalidades, como lo era la capacidad de ordenar los elementos , las cadenas de texto, que la integraban. TStrings ya poseía la capacidad de buscar de forma secuencial, un String dentro de la lista de Strings, pero claro está, mantener dicho método, sobre una lista ya ordenada era poco menos que un derroche de recursos, por cuanto existen métodos que optimizan la búsqueda. Este es el caso de la función Find( ), que deberá ser usada **únicamente** sobre listas ordenadas, bajo responsabilidad del programador el no hacerlo así en el caso de realizar un invocación directa del método. Nos podemos detener un poco en la observación de ambas :

```
function TStringList.IndexOf(const S: string): Integer;
begin
  if not Sorted then Result := inherited IndexOf(S) else
    if not Find(S, Result) then Result := -1;
end;
```

La función IndexOf( ) recibirá como parámetro la cadena de texto a buscar dentro de nuestro objeto TStringList, y nos devolverá un indice que corresponde, si ha tenido éxito la búsqueda, el lugar que ocupa dicha cadena en la lista; si no ha tenido éxito, la función se limitará tan solo a devolver -1.

Así podemos entender que la condición primera a evaluar sea si la lista está ordenada o no. De no estarlo, se invocará el método IndexOf(S) en el ascendiente, que se limitará a recorrer en un bucle f*or* cada una de las cadenas de la lista y evaluar si es igual a la pasada como parámetro, y en el caso de no tener éxito retornar -1 y en caso contrario el índice del elemento encontrado.

En el caso de estar ordenada nuestra lista de cadenas, invocamos al método Find( ):

```
function TStringList.Find(const S: string; var Index: Integer): Boolean;
var
  L, H, I, C: Integer;
begin
  Result := False;
  L := 0;
  H := FCount - 1;
  while L <= H do
  begin
    I := (L + H) shr 1;
    C := AnsiCompareText(FList^[I].FString, S);
    if C < 0 then L := I + 1 else
    begin
      H := I - 1;
      if C = 0 then
      begin
        Result := True;
        if Duplicates <> dupAccept then L := I;
      end;
    end;
  end;
  Index := L;
end;
```

Intentaremos razonar el algoritmo elegido por Borland, que está dentro de los que se pueden considerar como clásicos dentro de la búsquedas en un vector ordenado. Su orden de complejidad, concepto al que aludíamos en el segundo artículo de la serie, es del orden de logaritmo de n, razón de peso para considerarlo, dada la ganancia obtenida en su ejecución.

Consideremos dos variables de tipo entero A y B: La variable A será para nosotros el valor del índice del primer elemento del vector. La variable B representará el índice del último elemento. Establecer un bucle con extremos A y B, definido mediante *For* nos obligaría a recorrer todo el vector, desde el primer elemento hasta el último, en orden creciente o decreciente, manteniendo en su ejecución un orden de complejidad lineal tan solo aliviado por la posibilidad de una salida forzada mediante E*xit*. La estructura *While* nos permitirá modificar los pivotes A y B a medida que se ejecuta el algoritmo.

Veámoslo paso por paso:

```
Result := False;
```

Nos garantiza que al final del recorrido, de no haber sido encontrada la cadena de texto similar al string que entregamos como parámetro, devuelva False.

```
L := 0;
H := FCount - 1;
```

Fijamos el valor del indice asociado al primer elemento de la lista (L) y al último (H).

```
while L <= H do
```

Representa las condiciones que garantizan la salida del algoritmo. Nos podemos hacer una idea que supone que la búsqueda es realizada en orden creciente, hasta que nuestro pivote (L) rebose la posición (H).

```
I := (L + H) shr 1;
```

Dentro del cuerpo del bucle, va a ser una de las piezas fundamentales. Es una variable auxiliar que nos devuelve un "punto medio" entre A y B. Nos va a servir para evaluar mediante la función *AnsiCompareText(FList^[I].FString, S)* si en ese punto (I), la cadena asociada dentro de la matriz, al mismo, con respecto a la cadena pasada parámetro, devuelve valores 0, mayor o menor que cero.

Razonemos:

Si fuera igual a 0, ya habríamos encontrado la cadena y por lógica, nos limitaríamos a devolver dicho índice y el valor True como retorno de la función, dando por finalizado el algoritmo.

En el caso de ser **menor que cero**, significaría, dado que está ordenado, que existe un valor por encima del punto (I) que puede cumplir el algoritmo, puesto que todas las cadenas por debajo de ese punto no lo cumplen. Así, en el siguiente bucle, nos basta dar a la variable (A) el valor de (I+1), disminuyendo el rango del vector, e iniciando un nuevo bucle de análisis con la generación de nuevos puntos medios.

En el caso de ser **mayor que cero**, significaría, dado que está ordenado, que existe un valor por debajo del punto (I) que puede cumplir el algoritmo, puesto que todas las cadenas por encima de ese punto no lo cumplen. Así, en el siguiente bucle, nos basta dar a la variable (B) el valor de (I-1), disminuyendo el rango del vector, e iniciando un nuevo bucle de análisis con la generación de nuevos puntos medios.

Podemos resumir que la ganancia obtenida viene precisamente porque el rango del vector considerado en la búsqueda, es dividido por dos en cada ejecución del bucle, de ahí su complejidad de orden logaritmo de n (n representa el total de elementos que componen la lista)

Nos quedaban, con respecto al algoritmo, un matiz importante que debemos resaltar. Hubiera sido posible, haber considerado una sentencia de salida intercalada en el bloque que cumple la condición (C=0). De haberlo hecho hubiera quedado tal que así:

```
      if C = 0 then
      begin
        Result := True;
        Index:= I;
        L:= I;
      end;
```

Esto hubiera sido posible y recomendable en una lista que no admitiera valores duplicados. Sin embargo, la clase TStringList, permite su existencia y la función, devuelve el indice correspondiente al primero de los duplicados en el caso de aceptarlos. En caso contrario, de no aceptar duplicados, el algoritmo se detiene en el primero de los aciertos. Esto se consigue, avanzando la posición del pivote (A) hacia (I), mediante la rutina, :

```
if Duplicates <> dupAccept then L := I;
```

Resumiendo:

Es posible utilizar Find( ) sobre listas que acepten o no duplicados. Lo único que debemos tener en cuenta es que, necesariamente, los elementos de la lista deben estar ordenados, pues de lo contrario, no hay ninguna garantía de que obtenga éxito la búsqueda iniciada. Si fuera necesario invocarlo directamente nos basta con fijar la propiedad *Sorted* a *true*, pues internamente es llamado el método Sort( );

- Método de ordenación en listas de cadenas

En el artículo 2 de esta serie, nos extendíamos, ampliamente, en el algoritmo utilizado por Borland para efectuar una ordenación sobre listas de punteros (TList). Hablábamos del algoritmo de QuickSort. Su aplicación al ordenamiento de listas de cadenas no lo hace diferente salvo en algún matiz que ahora comentaremos. En el listado 1, podéis ver la implementación de dicho algoritmo.

Dicho esto, se puede entender que nos ciñamos tan solo a esos matices y que os remita al artículo 2 de la serie para ver el desarrollo del mismo.

```
procedure TStringList.QuickSort(L, R: Integer; SCompare: TStringListSortCompare);
var
  I, J, P: Integer;
begin
  repeat
    I := L;
    J := R;
    P := (L + R) shr 1;
    repeat
      while SCompare(Self, I, P) < 0 do Inc(I);
      while SCompare(Self, J, P) > 0 do Dec(J);
      if I <= J then
      begin
        ExchangeItems(I, J);
        if P = I then
          P := J
        else if P = J then
          P := I;
        Inc(I);
        Dec(J);
      end;
    until I > J;
    if L < J then QuickSort(L, J, SCompare);
    L := I;
  until I >= R;
end;
```

Listado 1. Algoritmo de ordenación en listas de cadenas

Nuestro punto de partida en la ejecución de este método puede pasar simplemente por fijar la propiedad booleana *Sorted* a *True*. No puede ser más simple. La modificación de dicha propiedad es hecha a través del método SetSorted( ) que, tras comprobar que ha sido modificado su valor y solo en ese caso, llama al método Sort en el caso de que estemos asignando a *True* dicha propiedad y almacenando el valor internamente dentro de la variable FSorted, sobre la que lee el estado actual de la propiedad.

```
procedure TStringList.SetSorted(Value: Boolean);
begin
  if FSorted <> Value then
  begin
    if Value then Sort;
    FSorted := Value;
  end;
end;
```

La invocación del método Sort inicia nuestro algoritmo de ordenación y lo hace con una llamada al procedimiento CustomSort( ). Ambos, son declarados virtuales en la clase, lo que significa que pueden ser sobrescritos por nosotros y posibilitándonos no solo la elección final del algoritmo mismo sino el criterio de ordenación elegido para hacerlo.

```
procedure TStringList.Sort;
begin
  CustomSort(StringListAnsiCompare);
end;
```

La función StringListAnsiCompare( ) es el criterio elegido por Borland para ordenar las cadenas y el valor de retorno será el resultado de evaluar la función AnsiCompareText declarada en la unidad SysUtils de la VCL. Esta función, recibe como parámetros los índices de los dos elementos a comparar, es decir, el lugar que están ocupando, lo que permite que pueda ser obtenidos ambos Strings y ser comparados. Tomemos S1 y S2 como la cadenas a comparar:

```
-Si S1 < S2 el valor de retorno será menor que 0.
-Si S1 > S2 el valor de retorno será mayor.
-Si S1 = S2 el valor de retorno será 0.

function StringListAnsiCompare(List: TStringList; Index1, Index2: Integer): Integer;
begin
  Result := AnsiCompareText(List.FList^[Index1].FString,
                            List.FList^[Index2].FString);
end;
```

Podemos comentar además, que dicha comparación no será sensible a mayúsculas o minúsculas y se realizará de acuerdo con los parámetros locales fijados por Windows.

Fijémonos en como se ha implementado el método CustomSort( ), responsable en última hora de invocar el algoritmo de ordenación, y que nos dará pie a comentar el último de los matices que nos puede resultar de interés.

```
procedure TStringList.CustomSort(Compare: TStringListSortCompare);
begin
  if not Sorted and (FCount > 1) then
  begin
    Changing;
    QuickSort(0, FCount - 1, Compare);
    Changed;
  end;
end;
```

Cuando hablábamos de la clase TList, dejábamos claro que, siendo como era una lista de punteros, el creador del algoritmo desconocía que criterio podría ser válido para ordenar unos objetos de los que tan solo obtenía un puntero, dejando al programador la responsabilidad de implementar una función correcta. Teníamos un ejemplo claro en las fuentes que se entregaban junto con el artículo. En esta ocasión, si que se conoce la naturaleza de la lista, lo que permite ya establecer un criterio razonable. Y como ya he comentado, al ser declarado virtual, lo hace flexible a poder ser modificado por nosotros para ser adaptado a problemas concretos.

- Método de intercambio de elementos en listas de cadenas

En ocasiones, se puede hacer necesario intercambiar dos elementos en la lista de Strings, para lo cual disponemos del método Exchange( ).

```
procedure TStringList.Exchange(Index1, Index2: Integer);
begin
  if (Index1 < 0) or (Index1 >= FCount) then Error(@SListIndexError, Index1);
  if (Index2 < 0) or (Index2 >= FCount) then Error(@SListIndexError, Index2);
  Changing;
  ExchangeItems(Index1, Index2);
  Changed;
end;
```

Éste, descarga el peso de la implementación en un procedimiento privado de la clase, limitándose tan solo a comprobar que los índices que se obtienen como parámetros, están en el rango del vector, e invocando antes y después de las modificaciones, los eventos de cambio. El corazón del intercambio lo encontramos al analizar el procedimiento ExcangeItems( ).

```
procedure TStringList.ExchangeItems(Index1, Index2: Integer);
var
  Temp: Integer;
  Item1, Item2: PStringItem;
begin
  Item1 := @FList^[Index1];
  Item2 := @FList^[Index2];
  Temp := Integer(Item1^.FString);
  Integer(Item1^.FString) := Integer(Item2^.FString);
  Integer(Item2^.FString) := Temp;
  Temp := Integer(Item1^.FObject);
  Integer(Item1^.FObject) := Integer(Item2^.FObject);
  Integer(Item2^.FObject) := Temp;
end;
```

PStringItem es un puntero a un registro TStringItem. Este registro almacena un puntero al string y otro puntero al objeto asociado. Ambas declaraciones, están al inicio de la declaración de la clase.

```
PStringItem = ^TStringItem;
TStringItem = record
              FString: String;
              FObject: TObject;
              end;
```

Pero pongamos un ejemplo muy sencillo y bastante tonto, antes de intentar explicarlo: Supongamos dos variables del tipo entero A y B y queremos intercambiar sus valores. Podríamos hacer

```
var
  Temp: Integer;
  A, B: Integer;
begin
   A:= 10;
   B:= 5;
   Temp:= A;
   B:= A;
```

```
   A:= Temp;
end;
```

Es exactamente lo mismo, con la diferencia de que en este caso no solo intercambiamos las cadenas sino también los objetos asociados.

Queda quizás un poco más claro que se hace necesario disponer de dos punteros, locales al procedimiento, y una variable auxiliar, que nos permitan intercambiar el contenido. En el proceso se pueden diferenciar 3 momentos distintos:

Momento 1: Obtenemos una referencia a cada elemento de la lista implicado.

```
Item1 := @FList^[Index1];
Item2 := @FList^[Index2];
```

Momento 2: Intercambiamos las cadenas de texto mediante la variable auxiliar. Esta variable auxiliar nos permite almacenar la dirección del primero de los punteros. Hecho esto, primero apunta a segundo, para acabar recogiendo en el segundo elemento un puntero hacia el primero :

```
Temp := Integer(Item1^.FString);
Integer(Item1^.FString) := Integer(Item2^.FString);
Integer(Item2^.FString) := Temp;
```

Momento 3: Intercambiamos los objetos de igual forma:

```
Temp := Integer(Item1^.FObject);
Integer(Item1^.FObject) := Integer(Item2^.FObject);
Integer(Item2^.FObject) := Temp;
```

Creo que no es necesario extendernos.

Básicamente, hemos visto los principales métodos que se declaran como públicos en la clase TStringList, pero, si recordáis como finalizábamos nuestro artículo anterior, como descendiente de la clase TStrings, también nos ofrecía capacidades menos conocidas y que, de alguna forma, estaban disponibles para nuestro uso. Veremos estos aspectos en el apartado que abrimos a continuación y que servirá para cerrar finalmente el artículo actual.

Adelante pues...

Más comentarios interesantes sobre TStrings y TStringList.

Resulta menos conocida, o por lo menos me lo parece a mi, la posibilidad de leer y escribir en cadenas con formato [Nombre = Valor], propia de los ficheros de configuración. No, no es casual que se haya dotado a la clase TStrings de está capacidad, si tenemos en cuenta la amplia polivalencia y compatibilidad que le son inherentes. Era nuestro tema de discusión al finalizar el artículo anterior.

Ahora iniciaba la escritura de todas estás lineas y me preguntaba, al igual que te puedes estar preguntando tú, si esta nueva capacidad iba a ser almacenada con procedimientos similares a la propiedad matricial Items. No lograba ver con claridad, como iba a ser posible que hiciera esto de forma económica y sencilla. Andaba ciertamente desorientado, pues buscaba en la parte privada del componente alguna pista que me afirmase de forma certera aquellos pensamientos. Andaba errado.

En realidad, no hay mayor economía que una buena “plantilla”. Olvidémonos en este momento del segundo miembro de la igualdad y centrémonos en el primero de ellos. Analicemos la propiedad *Names*:

```
property Names[Index: Integer]: string read GetName;
```

Como podéis ver es un propiedad matricial de solo lectura y la hace mediante la función GetName( ). Si a esto añadimos que a través de la propiedad V*alues* pueden ser gestionados los valores en la parte derecha de la igualdad. ¿Cual puede ser entonces la funcionalidad de la propiedad *Names*?

Antes de responder a esa pregunta, tenemos que interesarnos por una pieza que nos falta en este pequeño puzzle, y que es el método protegido Get( ).

```
function TStringList.Get(Index: Integer): string;
begin
  if (Index < 0) or (Index >= FCount) then Error(@SListIndexError, Index);
  Result := FList^[Index].FString;
end;
```

La función Get( ) es un método protegido, declarado como abstracto en la clase TStrings. Esto nos manifiesta que tan solo los descendientes de dicha clase van a poder redefinirlo. En realidad lo único que hace TStrings es decir a sus descendientes: "- Vuestra misión es devolver la cadena de texto asociada al indice. Me da igual donde la almacenéis y como lo hagáis. Yo lo único que quiero es dicha cadena..."

Y en esa tesitura se encuentra nuestro descendiente TStringList. El sí que sabe donde esta la cadena de texto asociada al índice y después de comprobar si esta dentro del rango del vector, resuelve el puntero al vector en el miembro que almacena la cadena (FString).

Ahora ya estamos en condiciones de entender la lectura de la propiedad *Names*:

```
function TStrings.GetName(Index: Integer): string;
var
  P: Integer;
begin
  Result := Get(Index);
  P := AnsiPos('=', Result);
  if P <> 0 then
    SetLength(Result, P-1) else
    SetLength(Result, 0);
end;
```

Cuando se produce la lectura es recibido el parámetro del índice de la cadena de texto. Si decimos:

```
MiVariableString:= MiLista.Names[1];
```

Le estamos diciendo a nuestro objeto MiLista, perteneciente a la clase TStringList, que nos entregue el valor asociado a la segunda cadena del vector. Este objeto, como descendiente de TStrings que es, busca en su método GetName( ) para obtener dicho valor.

Si la función Get( ) ha tenido éxito el valor de retorno ya contiene dicho String. Ahh... Pero a nosotros no nos interesa todo el String sino solo la parte izquierda de la igualdad por lo que:

a/ si en dicha cadena no existiera el carácter ' = ' que es buscado mediante AnsiPos( ), SetLength( ) nos devolvería una cadena vacía.

b/ si existiera dicho carácter, en P estaría la posición que ocupa, por lo que, SetLenght( ) reajusta nuestro String para retornar una subcadena hasta la posición anterior al carácter.

Repetimos la pregunta: ¿Cual puede ser entonces la funcionalidad de la propiedad *Names*?

Ahora podemos dar una respuesta: Como propiedad matricial que es, nos permitirá recorrer el vector y localizar aquellos valores que se hayan a la izquierda de la igualdad, empleando bucles con estructuras habituales para tal efecto, apoyándose en los valores Cero para el primer elemento y Count -1 para el último en el caso de querer recorrer todo el vector (hay que tener en cuenta que el resto de métodos de búsqueda están implementados para localizar todo el String y no solo la parte izquierda de la igualdad).

Y la parte derecha que...!!!!!

Ahora nos resulta mucho más fácil poder comprender esta propiedad, con sus métodos de escritura y lectura respectivos:

```
property Values[const Name: string]: string read GetValue write SetValue;
```

Para verlo con claridad imaginemos que tenemos esta cadena asociada al indice 2:

```
MiIDEPreferido = Delphi
```

Si en el transcurso del programa yo quisiera saber cual es mi entorno de trabajo preferido me bastaría con llamar a la propiedad *Values*:

```
MiString:= MiLista.Values['MiIDEPreferido'];
```

Cuando MiLista recibe la orden nuestra de devolver dicho valor, invocará la función GetValue para poder obtenerlo.

```
function TStrings.GetValue(const Name: string): string;
var
  I: Integer;
begin
  I := IndexOfName(Name);
  if I >= 0 then
    Result := Copy(Get(I), Length(Name) + 2, MaxInt) else
    Result := '';
end;
```

```
function TStrings.IndexOfName(const Name: string): Integer;
var
  P: Integer;
  S: string;
begin
  for Result := 0 to GetCount - 1 do
  begin
    S := Get(Result);
    P := AnsiPos('=', S);
    if (P <> 0) and (AnsiCompareText(Copy(S, 1, P - 1), Name) = 0) then Exit;
  end;
  Result := -1;
end;
```

Listado 2. Función de búsqueda por nombre en el formato [Nombre=Valor]. Como valor de retorno recibimos la posición que ocupa la cadena de texto entregada como parámetro. En el caso de no ser localizada recibiremos el valor -1, habitual en este tipo de funciones.

Si observáis el listado 2, la evaluación de la función IndexOfName( ) depositará sobre la variable I el resultado de la búsqueda para poder ser evaluado:

a/ si ha devuelto -1 retornará una cadena vacía.

b/ cualquier valor mayor o igual a cero, nos devolverá solo la parte derecha de la igualdad ya que de toda la cadena resultante de la invocación de Get( ), tan solo nos vamos a quedar una subcadena que cuyo inicio se fija en la longitud del parámetro (Name) + 2 (es decir, a partir del carácter siguiente al '=').

De forma similar se produce la escritura sobre la propiedad *Names*:

```
procedure TStrings.SetValue(const Name, Value: string);
```

```
var
  I: Integer;
begin
  I := IndexOfName(Name);
  if Value <> '' then
  begin
    if I < 0 then I := Add('');
    Put(I, Name + '=' + Value);
  end else
  begin
    if I >= 0 then Delete(I);
  end;
end;
```

Cuando el método SetValue( ) recibe los parámetros *Nombr*e y *Valor*, lo primero que va a hacer es buscar que posición ocupa en el vector de cadenas, previo al análisis del valor que deba asignar a la misma.

El resto no tiene mucha más dificultad. En el caso de que intentáramos asignar una cadena vacía, toda vez que ha sido localizada en la lista de cadenas, entiende que queremos eliminarlo de nuestra lista.

En el caso contrario, que estemos intentando asignar una cadena no vacía, lo primero que hace es: si no existe, añadir un elemento para obtener el indice correcto, y en cualquier caso, copia en el elemento correspondiente al indice I el formato [Nombre =Valor].

Pero también decías algo de un texto...

Comentábamos en el número anterior, como nuestros objetos descendientes de TStrings eran capaces de devolvernos en un solo String el contenido de todas sus lineas, intercalando entre ellas los retornos de carro. No lo vamos a analizar. Creo que después del análisis hecho a los métodos de escritura y lectura en la propiedad Values, nos facilitan claramente su entendimiento.

```
property Text: string read GetTextStr write SetTextStr;

function TStrings.GetTextStr: string;
var
  I, L, Size, Count: Integer;
  P: PChar;
  S: string;
begin
  Count := GetCount;
  Size := 0;
  for I := 0 to Count - 1 do Inc(Size, Length(Get(I)) + 2);
  SetString(Result, nil, Size);
  P := Pointer(Result);
  for I := 0 to Count - 1 do
  begin
    S := Get(I);
    L := Length(S);
    if L <> 0 then
    begin
      System.Move(Pointer(S)^, P^, L);
      Inc(P, L);
    end;
    P^ := #13;
    Inc(P);
    P^ := #10;
    Inc(P);
  end;
end;

procedure TStrings.SetTextStr(const Value: string);
```

```
var
  P, Start: PChar;
  S: string;
begin
  BeginUpdate;
  try
    Clear;
    P := Pointer(Value);
    if P <> nil then
      while P^ <> #0 do
      begin
        Start := P;
        while not (P^ in [#0, #10, #13]) do Inc(P);
        SetString(S, Start, P - Start);
        Add(S);
        if P^ = #13 then Inc(P);
        if P^ = #10 then Inc(P);
      end;
  finally
    EndUpdate;
  end;
end;
```

Listado 3. Métodos de escritura y lectura sobre la propiedad Text.

Mirad el listado 3. Básicamente es lo mismo. La lectura de la propiedad Text ha de entregar un String que contenga todas las cadenas de la lista y puesto que éstas están almacenadas en el vector, deberá leer del mismo secuencialmente cada una de ellas e ir incrustándolas en dicha cadena. Creo que no es excesivamente importante el modo en el que lo hace para nuestro efectos.

Para la escritura sucede lo mismo. Nuestra lista recibirá en un solo String todas las cadenas que componen el vector de listas y se verá obligado a diseccionar cadena a cadena, tomando como referencia #13 y #10 como fin de cadena, y salvándolos finalmente y en el orden correcto en el vector. Tampoco creo que merezca detenernos demasiado y el único comentario que haré será en resaltar que, previa a la escritura es invocado el método *Clear,* y ya sabéis que significa esto: cuando escribamos sobre Text, el contenido de mi objeto lista desaparecerá si el resultado de la escritura tiene éxito.

Creo que se entiende ahora la gran polivalencia que ofrece la clase Strings, como miembro de otro objeto.

Para el próximo número...

Nuestro próximo número será eminentemente práctico y veremos los dos últimos objetos auxiliares de la serie: las clases TStack y TQueue. Además, intentaremos que todo lo visto hasta el momento se plasme a través de algún pequeño ejemplo.

Vale... no me enrollo más. ;-)

Recibid un saludo y hasta ahora...