Objetos Auxiliares - Salvador Jover

s.jover@wanadoo.es

Objetos auxiliares I

A medida que se avanza en el conocimiento de Delphi, se descubre un mundo lleno de posibilidades; un mundo invisible en nuestro primer contacto con el entorno; un mundo que se hace necesario conocer.

Pudiéramos pensar, erróneamente, que la Biblioteca de componentes visuales (VCL) se limita a aquellos que nos proporciona la paleta de componentes, es decir, a todos aquellos componentes que se han derivado inicialmente de TComponent, y nada más lejos de la realidad. No es extraño pensarlo, sobretodo si tenemos en cuenta, como el entorno nos va ocultando sucesivamente aquellos detalles de más bajo nivel, facilitándonos el desarrollo de aplicaciones de forma rápida y sencilla. A mi parecer, y ésta es una opinión subjetiva, la primera vez que nos oculta todos estos detalles es cuando tomamos contacto con la VCL: abrimos el entorno, situamos los distintos componentes de la paleta sobre nuestra ficha, implementamos unas líneas de código en respuesta de un evento determinado, y a ejecutar... rápido y sencillo... Tan rápido y sencillo que nos puede hacer olvidar el concepto mismo de VCL, del que, con un ejemplo sencillo, y comparado con un árbol, diré que tan solo vemos sus hojas y acabamos ignorando el resto. Nos podemos quedar al final con la idea, de que nos basta con distribuir un conjunto de componentes sobre nuestro TForm, o más aun, de que siempre vamos a encontrar un componente que resuelva un determinado problema. Es más, Internet, hoy por hoy, es capaz de facilitarnos, ya sea gratuitamente o previo pago, miles de componentes. Pero quizás, nos podemos ir convirtiendo, poco a poco, en buscadores de objetos, sin más afán; con el peligro de ir anquilosándonos en la comodidad del que todo lo encuentra.

Delphi nos ofrece mucho más y está en nosotros descubrirlo día a día. Ya hemos empezado a vislumbrarlo en muchos de los artículos que han compuesto los dos primeros números: desde aquel que nos hablaba de ventanas de Tavo Ibaceta, pasando por el de José Luis Freire acerca de la variable Application, hasta el de Mario Rodríguez sobre la Shell de Windows. En ese sentido, al escribir esta serie sobre los Objetos auxiliares que nos facilita Delphi, se pretende profundizar sobre el funcionamiento de la VCL, ya que dichos objetos forman parte de la definición de numerosas clases derivadas.

Pero vamos por partes. Es preciso, antes que nada, comentar que la VCL es una jerarquía de clases, asociada a Delphi y a C++ Builder, y escrita en Object Pascal. Como jerarquía se entiende que existe una única raíz común, TObject, de la que descienden todos los objetos presentes en la VCL y se establecen, entre todos ellos, una relación reciproca ascendiente/descendiente. Ya tenemos una definición. No hemos especificado la naturaleza de esta relación recíproca entre dos elementos cualquiera, ya que, a nuestro efecto, no sería estrictamente necesario. De hacerlo nos veríamos obligados a introducir los conceptos de herencia y polimorfismo los cuales pueden pertenecer a un artículo específico del tema. Nos basta comprender que existe esa relación. ¿Entonces, los componentes de la paleta? Bueno, los componentes de la paleta descienden todos de la clase abstracta TComponent, descendiente a su vez de la clase TPersistent, descendiente a su vez de la raíz TObject, y tienen unas características que les hacen especiales: como que son manipulables tanto en tiempo de ejecución como de diseño –mediante el Inspector de Objetos – o que dicha clase implementa la capacidad de aparecer en la paleta de componentes o situarse sobre la ventana de diseño. Además, la clase TComponent redefinirá un método dinámico de la clase TPersistent que ha de permitir asignar un propietario a cada componente, teniendo este propietario, la obligación y la capacidad de eliminar la memoria de los componentes asociados a él mediante esta relación. Y quizás os preguntareis el motivo por el que se introducen todos estos conceptos, ajenos en principio al tema que nos ocupa. Vamos a abrir un nuevo apartado e intentaremos justificarlo.

Los objetos auxiliares.

Convendría responder a la duda planteada en el párrafo anterior: El motivo por el que abordamos las particularidades de TComponent, es para remarcar que dichos objetos auxiliares, puesto que no derivan de la clase TComponent, sino que lo hacen directa o indirectamente de TObject, no pueden ni saben liberar la memoria asociada a su creación de forma automática, por lo que hemos de hacerlo nosotros directamente. Lo veremos claramente si observamos el constructor de un componente cualquiera. Vamos a poner como ejemplo TButton:

```
Constructor Create(AOwner: TComponet); override;
```

Objetos auxiliares I

Cuando hacemos la llamada al método *Create( )* pasaremos como parámetro TComponent. Este será el propietario de nuestro botón y dispondrá (el propietario) de una lista con todos los objetos que se han ido creando en la aplicación y que le han aceptado como propietario de los mismos, siendo entonces su responsabilidad el eliminarlos. Ese no será el caso de nuestro TList. TList hereda de TObject su constructor y la llamada a *Create* la realizaremos sin parámetro alguno:

```
var
una_lista: TList;
begin
una_lista:= TList.Create;
...
...
end;
```

Esto, que aparentemente pueda pasar desapercibido en un primer momento, tiene importancia y no poca. En nosotros recaerá la obligación de liberar la memoria del objeto creado. En nosotros, o en cualquier componente que estemos construyendo y que los utilice. Además, por el mismo motivo comentado, tan solo serán manipulables en tiempo de diseño integrados como parte de otro objeto descendiente de TComponent.

Pero llegamos a la parte central de este nuevo apartado. Ya sabemos algo sobre estos objetos pero necesitamos saber que tienen de especiales, o que les ha hecho tan interesantes para dedicarles atención.

Leamos lo que nos dice la Guía del Desarrollador que acompaña a Delphi sobre ellos y nos vamos a quedar con dos párrafos:

"...simplifican las tareas de programación más frecuentes."
ó
"En este apartado se describen varios objetos auxiliares que facilitan las siguientes tareas:
- Creación y gestión de listas.
- Creación y gestión de listas de cadenas.
- Modificación del registro y los archivos INI de Windows.
- Envío de flujos de datos a un disco duro u otro dispositivo de almacenamiento."

A primera vista, se podría desprender la idea de que dichos objetos se han creado, expresamente, dentro de un marco de facilitar al programador una serie de objetos útiles. Un visión menos superficial de estos objetos nos va a hacer pensar que, unos en mayor medida que otros, se harán necesarios en la construcción de la VCL. Pongamos un ejemplo cualquiera: ¿hubiera sido posible el desarrollo de la clase TComponent, tal y como la conocemos, sin el desarrollo anterior de la clase TList o del concepto que ésta representa?.

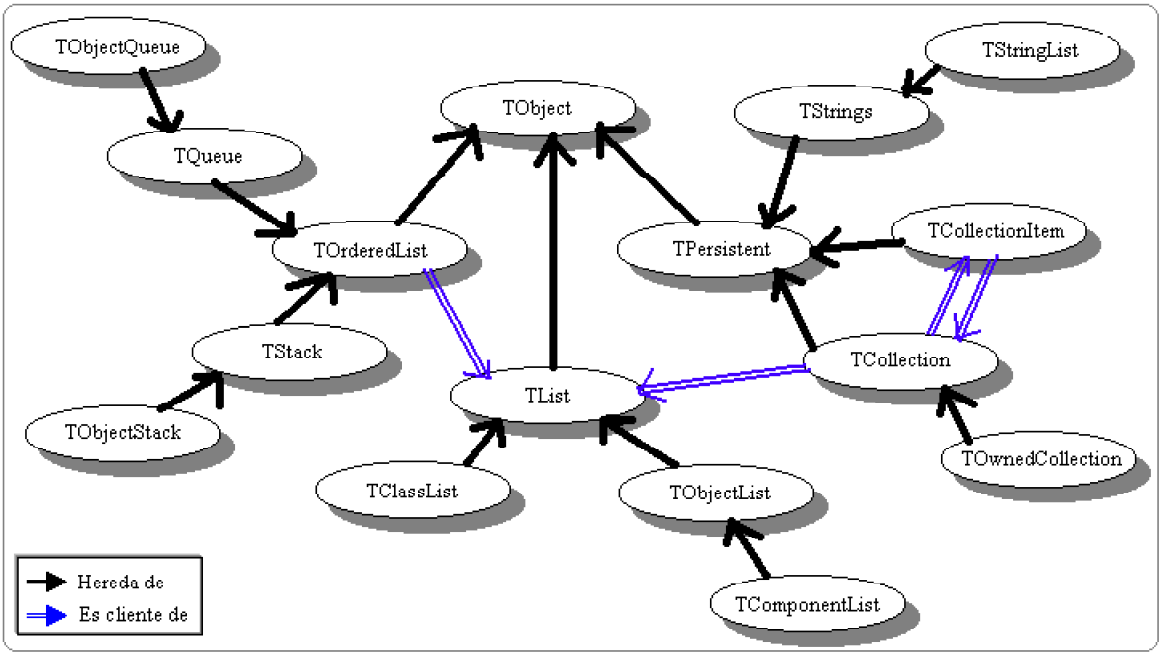

Figura 1. Relación jerárquica de los objetos relacionados con las listas.

El mismo sistema de notificaciones (algo de por si global a todos los componentes) está basado en una instancia de la clase TList, - que no es otra cosa que la implementación de una lista de punteros- , y gracias a la misma, a este sistema de notificaciones, cada componente mantiene una lista de los objetos a los que debe notificar su destrucción. Podríamos seguir poniendo ejemplos, los encontraríamos en cada uno de los módulos que componen la VCL, pero se me antoja necesario, seguir acotando el concepto de objeto auxiliar, tal y como se nos presenta en la Guía. Y la forma de hacerlo, toda vez que comprendemos el sentido más amplio de la palabra utilidad, es comentar o profundizar en las tareas que nos son facilitadas por ellos.

Acerquémonos al tema que nos ocupa de forma introductoria, para tener una visión global de los objetos sobre los que, posteriormente, nos vamos a centrar.

Tareas, tareas, tareas...

La primera de las tareas que se describen en la Guía del Desarrollador es la de creación y gestión de listas. Incluiremos, también en este apartado, la creación y gestión de listas de cadenas. Podéis echar una mirada a la **Figura 1**. Allí están representados, jerárquicamente, aquellos objetos que nos van a ayudar: TList, TObjectList, TComponentList, TQueue, TStack, TObjectQueue, TCLassList, TCollection, TOwnedCollection, TCollectionItem y TStringList. Un breve comentario antes de continuar: estamos tomando como referencia Delphi 5.0. y C++ Builder 5.0. Es preciso hacer esta anotación ya que en versiones anteriores de VCL nos encontraremos que algunas clases nombradas aun no existen. El es caso de TObjectList, TComponentList, TQueue, TStack, TObjectQueue o TCLassList, todas ellas pertenecientes al modulo “contnrs.pas”.

Si observamos con un poco de detenimiento dicha figura, ya podemos empezar a matizar diferencias entre las tareas que desempeñaran los objetos mencionados:

El primer grupo que mantiene unas características comunes es el formado por los descendientes de la clase TList, la clase que gestionará una lista de punteros. Todos tienen en común, en cuanto parten de la misma raíz, la capacidad de gestionar o mantener una colección homogénea de datos, que serán ordenados en una posición determinada. Es decir, responde a lo que podemos considerar el Tipo de dato Abstracto Lista. Abreviadamente, se suele escribir para referirse a los tipos de datos abstractos como TDA o TAD, según otros autores. Así, podemos decir que TObjectList mantendrá una lista de objetos de instancia, mientras que TComponentList hará lo propio con una lista de componentes (en la guía, además, se matiza que dicha lista es gestionada en memoria, lógico por cierto). De igual manera, TClassList tendrá como misión el mantenimiento de una lista de tipos de clases.

El segundo grupo lo forman los descendientes de TPersistent, que incorporan nuevas capacidades: por decirlo de forma sencilla, la de almacenar y recuperar esa información; lo que se llama la Persistencia.. Métodos como *Assign*, *LoadFromFile*, *LoadFromStream*, *SaveToFile* o *SaveToStream*, son característicos de dicha capacidad y son incorporados a los nuevos objetos. En este grupo nos encontramos por un lado tres objetos relacionados entre sí: TCollection, TOwnedCollection y TCollectionItem que mantienen una lista indexada de conjuntos de elementos.

Por otro lado, nos encontramos con TStringList, con capacidad para gestionar una lista de cadenas de caracteres. No hace falta mencionar el papel que juega, junto con su antecesor, TStrings, en la manipulación de cualquier texto en los habituales controles.

Por ultimo, y ya para finalizar este primer grupo de objetos, nos encontramos frente a dos tipos de lista especiales. Estamos hablando de TQueue y TStack (respectivamente con su descendiente). TQueue hace referencia a una estructura de datos que se caracteriza precisamente por mantener una lista de objetos siguiendo el criterio FIFO (First In First Out), el primero que entra es el primero que sale. Visualmente, lo podemos imaginar como una cola de personas que esperan un autobús. Si nos pasamos de listillos e intentamos adelantar puestos en la misma, posiblemente se vea recriminada nuestra conducta con insultos por parte de los presentes.

Otro matiz diferente alberga TStack: una estructura de datos que se caracteriza por mantener una lista de objetos bajo el criterio LIFO (Last In First Out), el último de entra es el primero que sale. Visualmente lo relacionamos con lo que tradicionalmente se denomina Pila de datos. Nos es imposible acceder a cualquier dato interior de la lista si no es retirado aquellos que han sido incorporados con posterioridad.

La segunda tarea que podemos relacionar con los objetos auxiliares es la de Modificación del Registro de Windows y de los ficheros INI. En el desarrollo de las aplicaciones, se hacen necesarios una serie de métodos, que nos permitan almacenar información sobre la configuración, sobre las preferencias del usuario, etc.. Con anterioridad a Windows 95, dicha información quedaba almacenada en ficheros con extensión INI. A partir de Windows 95, se hace uso del Registro de Windows, como centro neurálgico de toda la información del sistema. Es por eso, que nos aparecen, dentro de los objetos auxiliares pertenecientes a este apartado, aquellos objetos capaces de trabajar con dichos ficheros INI, como TIniFile o TMemIniFile, o bien aquellos que tan solo lo hacen con el registro, como TRegistry. Contamos además con un objeto común a ambos TRegistryIniFile, capaz de relacionarse tanto con el Registro de Windows como con los archivos de extensión INI. Podemos ver en la **figura 2** dichos objetos y su estructura jerárquica. Ya en este número, os recuerdo que

Objetos auxiliares I

Carlos Conca, nos va a introducir en los ficheros INI, en la continuación de un extraordinario artículo iniciado en el número anterior: ¿Ficheros de configuración o Base de registros?.

Finalmente, podemos observar el tercer y último grupo de objetos auxiliares considerados por la Guía, en la **Figura 3**. Todos son descendientes de la clase TStream, que implementa los métodos necesarios para acceder a los dispositivos de almacenamiento. Serán sus descendientes, los que particularizarán dicho acceso, ya sean archivos o memoria dinámica, implementado los métodos necesarios para la lectura y escritura en los mismos, copia de bytes o bien situarse en una posición determinada de dichos flujos.

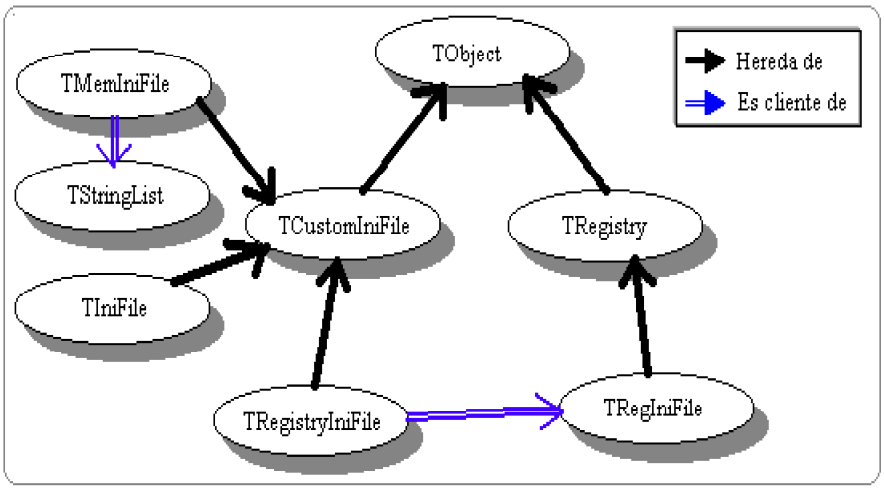

Figura 2. Relación jerárquica de los objetos relacionados con la configuración: almacenamiento lectura y modificación de valores.

Ya tenemos una visión global del concepto de Objeto Auxiliar y de las tareas que éstos desempeñan. A lo largo de varios artículos, si es que decidís seguir soportándome, se profundizará en aquellos detalles más significativos de cada uno de estos grupos. Nos basaremos principalmente en el código fuente que acompaña a nuestro compilador. Y quizás aquí podemos encontrar una primera reflexión al hilo de todo esto, de cierta importancia. ¿Existe un código fuente y está disponible para nosotros? Sí señor... Hay quien dirá: -¡Vaya perogrullada!. ¡Dejaremos de saber que existe un código fuente!.

Posiblemente sea así. Sin embargo haced una reflexión: Si analizáis muchas de las preguntas que se formulan en los foros de programación, la mayoría de ellas, podrían haber encontrado respuesta en las fuentes y sin embargo ahí están las preguntas. No hay una varita mágica. Tan solo nos vale tener un buen conocimiento de la sintaxis de Object Pascal y mucha paciencia. El resto lo dará la experiencia y el deseo .de aprender un poco más cada día, con humildad, con la vista puesta en aquellos que van por delante de nosotros, abriéndonos camino.

Para ser justos, y puestos a decirlo todo, también habrá que reconocer la ausencia de comentarios en las fuentes, que nos hacen considerarlas ampliamente mejorables. Es frecuente escuchar este comentario y como tal me hago eco del mismo.

Retomemos: Según sea la versión de nuestro compilador (Standard, Profesional o Enterprise), vamos a disponer de mayor o menor código fuente, lógicamente.

Si nos referimos en concreto al tema que nos ocupa, deberemos buscar en el modulo Classes.pas, en donde se implementan la mayoría de las clases que nos afectan, salvo los descendientes de TList y los descendientes de TOrderedList, que los encontraremos en el modulo Contrns.pas. Estos módulos, junto al resto de código fuente, se hayan repartidos en distintas carpetas en el directorio "...\Delphi5\Source\".

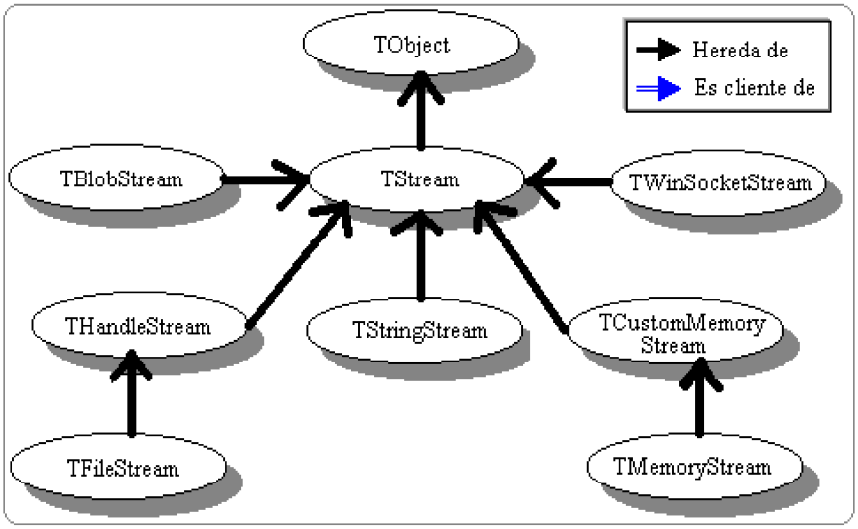

Figura 3. Relación jerárquica de los objetos relacionados con los flujos de memoria.

TList: una lista de punteros.

Decíamos anteriormente, que la clase TList respondía a lo que se podía llamar el TDA Lista, pero lo dejamos en el aire, sin entrar en más detalles. Habría que preguntarnos que podemos entender por TDA o Tipo de Dato Abstracto: Formalmente podemos entenderlo como una colección de valores definidos de forma única mediante un tipo y un conjunto de operaciones definidas sobre el mismo. Hablamos de la abstracción de un tipo de datos, sobre el que somos capaces de asignar determinadas capacidades innatas al mismo, que le son propias por naturaleza. En este ámbito, os invito a que iniciéis la lectura del artículo de Mario Rodríguez, acerca de la librería estándar STL y en el que se reseñan estructuras de datos comunes.

Así consideramos que una lista esta formada por una colección homogénea de elementos, punteros, que gozan de ordenación dentro de ella, es decir que podemos establecer un relación de orden entre dos elementos cualquiera de la misma (gracias a la existencia de un índice), y sobre los que se definen una serie de operaciones u operadores asociados. Respecto a un elemento dado se nos permitirá:

- Insertar un elemento en una lista de elementos.
- Obtener la posición que ocupa un elemento determinado.
- Suprimir un elemento a través de su posición en la lista.
- Vaciar el contenido de un lista.
- Obtener el primer o el último elemento de la misma.

Nos introduciremos en cada uno de estos operadores al comentar la implementación que se sigue en la clase TList.

En el **Listado 1** se nos presenta la interfaz de la clase. A la vista del mismo ya podemos sacar las primeras reflexiones iniciales. Fijémonos en la declaración de tipos; en concreto en la declaración de PPointerList y de TPointerList El primero se declara como un puntero hacia una estructura del tipo TPointerList. Éste último, como un vector cerrado, cuyos elementos son punteros, sin especificación de tipo de puntero, con un rango entre cero y (MaxListSize –1). Si nos vamos unas líneas mas arriba advertiremos que la constante MaxListSize toma el valor de (MaxInt div 16), la división entera entre High(Integer), que es así como se declara MaxInt en el módulo system.pas, y 16. Teniendo en cuenta el valor superior en el rango definido para los enteros, normalmente 2147483647 obtenemos un vector cuyo rango está entre 0 y 134217726.

```
const

  MaxListSize = Maxint div 16;
...
...
type

  PPointerList = ^TPointerList;
  TPointerList = array[0..MaxListSize - 1] of Pointer;
  TListSortCompare = function (Item1, Item2: Pointer): Integer;
  TListNotification = (lnAdded, lnExtracted, lnDeleted);

  TList = class(TObject)
  private
    FList: PPointerList;
    FCount: Integer;
    FCapacity: Integer;
  protected
    function Get(Index: Integer): Pointer;
    procedure Grow; virtual;
    procedure Put(Index: Integer; Item: Pointer);
    procedure Notify(Ptr: Pointer; Action: TListNotification); virtual;
    procedure SetCapacity(NewCapacity: Integer);
    procedure SetCount(NewCount: Integer);
  public
    destructor Destroy; override;
    function Add(Item: Pointer): Integer;
    procedure Clear; virtual;
    procedure Delete(Index: Integer);
    class procedure Error(const Msg: string; Data: Integer);
                                                  overload; virtual;
    class procedure Error(Msg: PResStringRec; Data: Integer); overload;
    procedure Exchange(Index1, Index2: Integer);
    function Expand: TList;
    function Extract(Item: Pointer): Pointer;
    function First: Pointer;
    function IndexOf(Item: Pointer): Integer;
    procedure Insert(Index: Integer; Item: Pointer);
    function Last: Pointer;
    procedure Move(CurIndex, NewIndex: Integer);
    function Remove(Item: Pointer): Integer;
    procedure Pack;
    procedure Sort(Compare: TListSortCompare);
    property Capacity: Integer read FCapacity write SetCapacity;
    property Count: Integer read FCount write SetCount;
    property Items[Index: Integer]: Pointer read Get write Put;
                                                          default;
    property List: PPointerList read FList;
  end;
```

Listado 1. Interfaz de la clase TList.

Hay que saber, que la implementación del TDA Lista podría haber sido desarrollada, dada la naturaleza dinámica de la misma (su longitud va a depender del número de elementos que la formen), mediante el uso del los TDA basados en

punteros: podríamos hablar de un Lista Enlazada o de una Lista Doble Enlazada. Los desarrolladores optaron por un Array o Matriz como medio para almacenar en memoria los elementos que la integran. ¿Entonces, tratándose de un vector cerrado podemos pensar que la asignación de memoria se realiza de forma estática, en tiempo de compilación?. No. La asignación de memoria será dinámica, es decir, se crearán los mecanismos oportunos para la reserva de la memoria necesaria en tiempo de ejecución. Y eso se hará mediante un procedimiento definido en el módulo System.pas: *ReallocMem*. Este método toma como parámetros un puntero al bloque de memoria que será reasignado y el tamaño de la misma en bytes. Analicemos esquemáticamente como se producirán estas asignaciones dinámicas de memoria:

*Situación inicial: No tenemos todavía ningún puntero en nuestra lista.
FCount = 0 (todavía no hemos insertado elemento alguno)
FCapacity = 0 (todavía no se ha asignado memoria)

*Añadimos el primer elemento.

*Comprobamos *FCount = FCapacity*.

Puesto que FCapacity nos indica la cantidad de memoria reservada actualmente por *FList*, de ser igual al numero real de elementos existentes, nos obligará a reservar nueva memoria. Una llamada al procedimiento Grow, un método virtual que podrá ser redefinido por los descendientes de TList, y desde este a *SetCapacity( )*, serán suficientes.

*SetCapaciy comprueba que dicha capacidad esta dentro del rango del vector y de no ser así nos será comunicado mediante una excepción del tipo EListError. En el caso contrario, de ser correcta, y distinta de la actual, ejecuta la siguiente rutina:
*ReallocMem(FList, NewCapacity * SizeOf(Pointer))*

*Y posteriormente se reasigna *FCapacity* con el valor actual.

Veamos la sintaxis de *ReallocMem*:
***Procedure** ReallocMem(**var** P: Pointer; Size: Integer);*
Comentar que el procedimiento *ReallocMem*, según los parámetros recibidos en la llamada, actuará de una forma o de otra. En la situación inicial figurada, FList no estará asignado todavía, entregándonos como parámetro Nil. Asimismo, *Size* tendrá valor mayor que 0,. En este caso, el procedimiento reservará un nuevo bloque de memoria del tamaño de dicho entero.
¿Entonces FCount ha de coincidir con FCapacity? No. Aquí las apariencias nos engañan. FCapacity siempre habrá de ser mayor o igual a FCount y el motivo de ser así, es tan solo por razones de eficiencia. Pensemos que de reservar únicamente memoria para el elemento que se va insertar en la lista produciríamos un continuo ir y venir, reservando y liberando memoria en cada nueva inserción o borrado de un elemento de la lista. Para comprenderlo veámoslo en el procedimiento TList.Grow, que será el que controle el tamaño de la asignación de memoria necesaria:

```
procedure TList.Grow;
var
  Delta: Integer;
begin
  If FCapacity > 64 then
     Delta:= FCapacity div 4
  else
     If FCapacity > 8 then
        Delta:= 16
     else
        Delta:= 4;
  SetCapacity(FCapacity + Delta);
end;
```

Vemos que en la cantidad a reservar de memoria (FCapacity + Delta), Delta tomará un valor que tan solo depende de la capacacidad actual de la lista. Para valores de FCapacity entre 0 y 8 tan solo se reservaran 4 bits adicionales. Para valores entre 9 y 64 se reservaran 16 bits adicionales. La eficiencia en este caso la podemos considerar en el hecho de que, a partir de FCapacity > 65, la cantidad de memoria reservada irá en función de la necesidades actuales, es decir, que para mayor capacidad, reservaremos adicionalmente una cuarta parte de esta capacidad: a mayor demanda entonces mayor reserva de memoria.

El ciclo de destrucción de la memoria asociada a *FList* es similar y se realiza también mediante una llamada a ReallocMem pasando como parámetro 0 en la llamada al método *SetCapacity( )*. El destructor de la clase llamará al método *Clear*, y este a su vez, a los métodos SetCount( ) y SetCapacity( ), en ese orden.

SetCount( ) y SetCapacity( ) son declarados en la zona protegida de la clase TList. Ambos son el método de escritura de dos propiedades respectivas: Count, que nos devuelve el numero actual de entradas del objeto lista y Capacity, que nos devuelve el tamaño de la memoria asignada a FList. Como hemos dicho y explicado no tienen que ser iguales.

Podemos proceder entonces a analizar que es lo que hacen realmente estos métodos al ser llamados por distintos valores. Estos valores los podemos resumir en dos: Para SetCount( ), que el nuevo valor de FCount sea mayor que el anterior, o bien que sea igual o menor, que es el segundo caso. Para SetCapacity tan solo distinguiremos que el valor de FCapacity sea igual o distinto del actual.

Pero veámoslo por separado:

```
procedure TList.SetCount(NewCount: Integer);
var
  I: Integer;
begin
  if (NewCount < 0) or (NewCount > MaxListSize) then
    Error(@SListCountError, NewCount);
  if NewCount > FCapacity then
    SetCapacity(NewCount);
  if NewCount > FCount then
    FillChar(FList^[FCount], (NewCount - FCount) * SizeOf(Pointer), 0)
  else
    for I := FCount - 1 downto NewCount do
      delete(I);
  FCount := NewCount;
end;
```

Lo primero que hará el procedimiento, será comprobar que el nuevo valor, que se pasa como parámetro en *NewCount*, toma valores comprendido en el rango del vector: comprueba que no sea un valor negativo ni mayor que el rango superior, de ser así será ejecutado el método Error( ), generando una excepción de tipo EListError con el mensaje 'List count out of bounds '. Dicha excepción la podremos capturar mediante los métodos habituales ‘try... except’ o ‘try...finally’. Los distintos mensajes generados ante el tipo de excepción EListError los podréis ver en el modulo ‘Consts.pas’.

En caso contrario seguimos la ejecución de las rutinas. SetCapacity reservará memoria tan solo en el caso de que nos encontremos ante un nuevo elemento añadido (NewCount > FCapacity). Esto se hace previo a la evaluación posterior (NewCount > FCount), precisamente para prevenir el caso comentado, para evitar que se pueda incrementar el contador FCount sin haber reservado memoria a tal efecto. En este punto llegamos al momento clave: En el caso de que nos encontremos ante una nueva entrada, FillChar inicializará con ceros la memoria asignada entre FCount y NewCount. En caso contrario, nos encontraremos con una salida o con un borrado definitivo (si NewCount fuera igual a 0), donde procederemos a destruir cada una los elementos entre NewCount (el actual) y FCount –1. Ya para finalizar se reasigna el nuevo valor de FCount.

```
procedure TList.SetCapacity(NewCapacity: Integer);
begin
  if (NewCapacity < FCount) or (NewCapacity > MaxListSize) then
    Error(@SListCapacityError, NewCapacity);
  if NewCapacity <> FCapacity then
  begin
    ReallocMem(FList, NewCapacity * SizeOf(Pointer));
    FCapacity := NewCapacity;
  end;
end;
```

Al igual que el método anterior, previo a cualquier otra rutina, deberemos comprobar que el parámetro NewCapacity, de tipo Entero, está dentro de los rango que admite el vector. Nuevamente de no ser así generaremos la oportuna Excepción del tipo EListError, en este caso con el mensaje 'List capacity out of bounds’.

Cuando evaluamos (NewCapacity <> FCapacity) nos aseguramos que realmente nos ha variado la capacidad previa del vector, y solo en ese caso, procederemos a modificar la cantidad de memoria asignada a FList de acuerdo con el nuevo tamaño de capacidad (NewCapacity). Además, en el caso particular de que NewCapacity tome el valor de 0, cuando procedemos a destruir el objeto de la clase, ReallocMem liberará el bloque de memoria asignado a FList y asignará FList a Nil.

Hasta ahora hemos estado analizando dos de las propiedades, hechas publicas por la clase, que son respectivamente *Capacity* y *Count.* Hemos visto sus métodos de escritura y como se produce el ciclo de asignación de memoria o de liberación de la misma. No hemos comentado, y lo haremos ahora, algún detalle sobre la propiedad List. Finalmente y antes de pasar a los distintos métodos, haremos lo propio con la propiedad Items, que será la que nos permita el acceso, tanto en escritura como en lectura, a los valores de cada uno de los elementos de la lista.

La propiedad List es la que representa a la lista de punteros. Definida como un puntero a una estructura de tipo TPointerList, tan solo nos permitirá su lectura a través del campo privado FList. De tal forma que se protege precisamente la manipulación fuera de los métodos declarados.

TList: particularidades de la propiedad Items.

Abordamos una propiedad que nos puede parecer un poco especial y para la que aprovecharé este apartado que me permite diferenciarla de las demás. Desde el punto de vista funcional podremos decir que en esta propiedad se almacenará un puntero a uno de los elementos del array o vector FList. Es precisamente este puntero, dentro de una lista indexada, el que nos permita resolver los valores de cada uno de los elementos del array.

Como propiedad pertenece a lo que se denominan Propiedades matriciales, propiedades indexadas cuyo índice, de tipo entero, recorre la matriz, desde 0 hasta Count – 1.

Respecto a los métodos de lectura y de escritura en este tipo de propiedades, es decir, lo que se denominan especificadores de acceso, se nos obligará a que dichos especificadores contengan métodos en lugar de campos. En el caso de que hablemos del especificador de lectura éste lo hará a través de un función que tomara como parámetros el mismo numero y tipo de los parámetros enumerados en la lista de índices de la propiedad. Acerca del especificador de escritura, en este caso un procedimiento, se hará el mismo comentario anterior solo que en este caso no devolverá valor alguno.

En el caso que nos ocupa, la propiedad Items toma como especificador de lectura la función Put( ) y como especificador de escritura el procedimiento Get( ).

Detengámonos en ellos unos minutos:

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

Como hemos comentado, la función Get ha de devolver el valor asociado a la propiedad matricial. También hemos visto que en FList se van a ir almacenando punteros en el array. Podemos pensar que dichos punteros, apuntarán a ‘algo’, ya sea un objeto, un entero, una estructura definida, en fin, lo que sea. Leemos línea a línea el código: “Si el entero Index es menor que cero o mayor que FCount entonces lanzarás una excepción del tipo EListError con el mensaje - List index out of bounds -. (De no haber sucedido esto) devolverás a la propiedad Items[index] un puntero al elemento de índice Index del array FList”.

```
procedure TList.Put(Index: Integer; Item: Pointer);
var
  Temp: Pointer;
begin
  if (Index < 0) or (Index >= FCount) then
    Error(@SListIndexError, Index);
  Temp := FList^[Index];
  FList^[Index] := Item;
  if Temp <> nil then
    Notify(Temp, lnDeleted);
  if Item <> nil then
    Notify(Item, lnAdded);
end;
```

Vamos a ver lo que sucede cuando modificamos el valor de la propiedad *Items*. Volvemos a interpretar el código: " Si el valor del índice Index – que es el pasado como parámetro- es menor que 0 y mayor que *FCount*, entonces lanzarás una excepción del tipo EListError con el mensaje comentado ya en el apartado anterior. (De no producirse esto) la variable de tipo Pointer, Temp, apuntará al elemento de índice Index de la lista. Dicho elemento (cuyo tipo es Pointer) apuntará a la variable Item, también de tipo Pointer pasada como parámetro".

De esa forma, al ejecutar el proceso de escritura de la propiedad *Items* lo que producimos es la modificación de uno de los elementos del vector lista, que toma el nuevo valor.

Otro detalle que nos puede pasar desapercibido es la llamada posterior al método virtual Notify. Veamos la implementación que hace la clase de dicho procedimiento:

```
procedure TList.Notify(Ptr: Pointer; Action: TListNotification);
begin
end;
```

Simplemente no hace nada. Es un método virtual que podrá ser redefinido por sus descendientes, los cuales recibirán como parámetros un puntero y una variable de tipo enumerado, TListNotification, definida previamente en la declaración de tipos:

. TListNotification = (lnAdded, lnExtracted, lnDeleted);

Así que, tan solo en el caso de que dicho puntero este asignado, se procederá a notificar que ha sido borrado o añadido, que son los dos casos que aquí se dan.

Operaciones asociadas al tipo lista.

Hemos comentado que existirán unas operaciones asociadas al TDA Lista. Vamos a reconocer en la implementación de la clase TList cuales procedimientos y funciones van a responder a dichos operadores u operaciones.

Insertar un elemento en una lista de elementos:

Podemos considerar que existen dos casos diferentes, en cuanto podemos tanto añadir un elemento por el final de la lista, como insertarlo entre medio de dos elementos cualquiera. En el segundo caso necesitaremos un parámetro adicional, que será el índice que nos indica en que posición deberíamos realizar la inserción. En términos de ir por casa, diremos que al insertar un nuevo elemento, (no hace falta indicar que son punteros), deberíamos desplazar todos aquellos con un índice superior a la posición en la que se inserta el elemento nuevo, en una unidad del mismo valor que representa el puntero (4 bytes).

a-.Insertar un elemento por el final de la lista.

```
function TList.Add(Item: Pointer): Integer;
begin
  Result := FCount;
  if Result = FCapacity then
    Grow;
  FList^[Result] := Item;
  Inc(FCount);
  if Item <> nil then
    Notify(Item, lnAdded);
end;
```

Cada vez que procedemos a añadir un elemento nuevo a la lista debemos comprobar que tenemos capacidad suficiente para poder hacerlo, es decir, si hemos asignado memoria suficiente en una operación anterior (Result = FCapacity) y en el caso de que no sea así, procederemos a ejecutar el procedimiento Grow, ya explicado, para que se aumente la cantidad de memoria asignada. Asignaremos el nuevo puntero en nuestra lista, en la posición FCount, e incrementaremos el valor de FCount en una unidad. Debemos recordar que el primer elemento de nuestra lista no está en la posición 1 del vector, sino que empezamos a contar desde la posición 0 hasta FCount – 1, por lo que la nueva posición a añadir vendría dada por FCount.

Notificaremos la adición del nuevo elemento para finalizar.

a-.Insertar un elemento en una posición de la lista.

```
procedure TList.Insert(Index: Integer; Item: Pointer);
begin
  if (Index < 0) or (Index > FCount) then
    Error(@SListIndexError, Index);
  if FCount = FCapacity then
    Grow;
  if Index < FCount then
    System.Move(FList^[Index], FList^[Index + 1],
      (FCount - Index) * SizeOf(Pointer));
  FList^[Index] := Item;
  Inc(FCount);
  if Item <> nil then
    Notify(Item, lnAdded);
end;
```

Al recibir como parámetro el índice que nos define la posición en la que debemos realizar la inserción del elemento, comprobaremos en primer lugar, como siempre hacemos, que estamos en una posición valida, dentro del rango del vector. También como ya hemos repetido, de no ser así lanzaremos una excepción del tipo EListError.

Volvemos a comprobar si es necesaria la asignación de nueva memoria y en el caso que dicha posición sea menor que FCount, ejecutaremos el procedimiento Move( ) del modulo 'System.pas'.

¿Qué efecto producirá la ejecución de Move( )? Imagino que ya lo veréis con cierta claridad. Hemos dicho que necesitamos desplazar en una posición el contenido del vector a partir de Index. El procedimiento Move copiará los ((FCount – Index)*SizeOf(Pointer)) bytes a partir de la posición FList^[Index + 1]. Antes de ejecutar este procedimiento, debe ser siempre comprobado que existe suficiente memoria disponible para la copia, ya que de no hacerlo así, nos arriesgaríamos a destruir posiciones de memoria contiguas, que no le pertenecen a la variable.

Toda vez que se ha realizado la transferencia tan solo nos quedará insertar en la posición Index del array FList el nuevo puntero e incrementar el contador FCount. Asimismo, procederemos a lanzar la notificación de que ha sido añadido un nuevo elemento en la lista.

Obtener la posición que ocupa un elemento de la lista:

En ocasiones necesitaremos obtener la posición que ocupa un elemento determinado. Para resolver este problema disponemos de la función IndexOf, que recibirá como parámetros un puntero y nos devolverá su posición en la lista

```
function TList.IndexOf(Item: Pointer): Integer;
begin
  Result := 0;
  while (Result < FCount) and (FList^[Result] <> Item) do
    Inc(Result);
  if Result = FCount then
    Result := -1;
end;
```

Intuitivamente lo vemos: Asignamos a cero la posición inicial del índice que nos recorrerá el vector, en este caso, es el resultado que nos ha de devolver la función. Iniciamos un bucle con el que recorreremos el array mientras se cumpla dos condiciones: que no lleguemos al final del índice del vector, a nuestros efectos FCount, y que el puntero que entregamos como parámetro sea distinto del actual, el que tiene como índice a Result. El resto es fácil de imaginar. De haber cumplido FList^[Result] = Item obtendríamos directamente el final de bucle y el retorno correcto de la función. Si no es así, el bucle se verá finalizado al cumplir (Result = FCount) por lo que la función devolverá –1, indicándonos que la búsqueda se ha realizado sin éxito.

Suprimir un elemento a través de la posición en la lista:

Existen dos funciones muy similares. La diferencia es tan solo de matiz. Agradezco a Mario Rodríguez y a Tavo Ibaceta su paciencia y sus consejos para que os lo pueda explicar con propiedad. Veamos:

En los dos casos, la llamada a ambas funciones va a producir la eliminación del elemento *Item* de la lista y también en los dos casos se va a partir de una búsqueda inicial mediante el método IndexOf( ), comentado anteriormente. El matiz lo tenemos en lo que hace la función una vez que ha encontrado dicho elemento (el puntero). En el primer caso, simplemente no se hace nada y directamente llamamos al método Delete( ) que eliminará el puntero de nuestra lista. Observar el valor de devolución: Integer. Lo que nos devuelve la función es la posición del elemento eliminado.

```
function TList.Remove(Item: Pointer): Integer;
begin
  Result := IndexOf(Item);
  if Result >= 0 then
    Delete(Result);
end;
```

En el segundo caso asistimos a una asignación previa al borrado: (Result := Item). Estamos devolviendo a través de la función un puntero al elemento, y seguidamente, con (FList^[I] := nil) nos aseguramos que en la llamada a Delete( ) tan solo se elimine el elemento de la lista y no el objeto al que apunta dicho puntero (ya que dicho puntero ya no apunta hacia ningún otro objeto). De esa forma, se extrae el elemento de la lista pero no se destruye.

```
function TList.Extract(Item: Pointer): Pointer;
var
  I: Integer;
begin
  Result := nil;
  I := IndexOf(Item);
  if I >= 0 then
  begin
    Result := Item;
    FList^[I] := nil;
    Delete(I);
    Notify(Result, lnExtracted);
  end;
end;

procedure TList.Delete(Index: Integer);
var
  Temp: Pointer;
begin
  if (Index < 0) or (Index >= FCount) then
    Error(@SListIndexError, Index);
  Temp := Items[Index];
  Dec(FCount);
  if Index < FCount then
    System.Move(FList^[Index + 1], FList^[Index],
      (FCount - Index) * SizeOf(Pointer));
  if Temp <> nil then
    Notify(Temp, lnDeleted);
end;
```

Comentaré tan solo la línea que me parece más significativa. Más, cuanto parte de lo ya comentado con anterioridad es aplicable al procedimiento. En el caso de que nos encontremos en una posición (Index < FCount), el procedimiento Move, efectuará una copia del contenido desde la posición [Index + 1] hasta [FCount], en la posición actual [Index]. Se copiarán, como ya comentamos en las operaciones de Inserción, ((FCount - Index) * SizeOf(Pointer)) bytes..

Vaciar el contenido de la lista:

Pienso que no hace falta extenderme en este procedimiento pues ya se ha comentado con anterioridad.

```
procedure TList.Clear;
begin
```

```
  SetCount(0);
  SetCapacity(0);
end;
```

Obtener el primer o el último elemento:

```
function TList.First: Pointer;
begin
  Result := Get(0);
end;

function TList.Last: Pointer;
begin
  Result := Get(FCount - 1);
end;
```

En el primer caso se nos devolverá el puntero FList^[0], (el primer elemento de la lista). En el segundo, el puntero FList^[FCount-1], (el último elemento de la lista).

Otras operaciones adicionales:

Ya para finalizar, comentaremos algunos de los métodos que hace públicos la clase TList y que no he incluido dentro de las operaciones mas elementales. En principio por que se pueden expresar como una composición de las anteriores, que es el caso del procedimiento Move, por ejemplo, que producirá el traslado de un elemento en una posición de la lista, hacia otra posición de la misma en la que se insertará. Vemos pues, que es en definitiva, la aplicación de operaciones mas elementales, tales como Delete( ) e Insert( ).

```
procedure TList.Move(CurIndex, NewIndex: Integer);
var
  Item: Pointer;
begin
  if CurIndex <> NewIndex then
  begin
    if (NewIndex < 0) or (NewIndex >= FCount) then
      Error(@SListIndexError, NewIndex);
    Item := Get(CurIndex);
    FList^[CurIndex] := nil;
    Delete(CurIndex);
    Insert(NewIndex, nil);
    FList^[NewIndex] := Item;
  end;
end;
```

Exchange( ) , en cambio, es una sucesión de asignaciones por las que, mediante una variable referenciada (Item) del mismo tipo, intercambiaremos las posiciones de dos elementos de la lista. Como en ocasiones anteriores, antes de proceder a las asignaciones, deberemos comprobar que dichos índices, Index1 e Index2 toman valores pertenecientes al rango definido por el numero de elementos (FCount). Toda vez que se ha comprobado que esto es así, se procederán a intercambiar el contenido de las posiciones de memoria.

```
procedure TList.Exchange(Index1, Index2: Integer);
var
  Item: Pointer;
begin
  if (Index1 < 0) or (Index1 >= FCount) then
    Error(@SListIndexError, Index1);
  if (Index2 < 0) or (Index2 >= FCount) then
    Error(@SListIndexError, Index2);
  Item := FList^[Index1];
  FList^[Index1] := FList^[Index2];
```

```
  FList^[Index2] := Item;
end;
```

Cabe pensar en que momentos puede ser utilizado este procedimiento con el que acabamos este primer artículo. La llamada al procedimiento Pack eliminará aquellos elementos del vector que no están asignados. Es una rutina de limpieza del array, por decirlo de alguna manera sencilla.

```
procedure TList.Pack;
var
  I: Integer;
begin
  for I := FCount - 1 downto 0 do
    if Items[I] = nil then
      Delete(I);
end;
```

Lo que nos espera...

En el próximo artículo, abordaremos una operación común a las clases TStringList y TList: el método de ordenación rápida o Quicksort, implementado en el procedimiento Sort( ). Esto nos permitirá introducir algunos conceptos nuevos, tales como el coste de un algoritmo, o su naturaleza recursiva. Apuntaremos algunos detalles que nos resulten de interés sobre los descendientes de la clase TList y nos centraremos en las clases TStrings y TStringList, acompañando esta parte más teórica con algún ejemplo práctico que nos ayude a utilizar sus métodos.

Un saludo y hasta el próximo número... ;-)

Bibliografía:

Me parece de justicia, nombrar aquellas fuentes en las que me he apoyado a la hora de documentarme, y que me servirán de guía en éste y sucesivos artículos:

Estructuras de Datos y Algoritmos, de R. Hernández, J.C. Lázaro, Raquel Dormido y Salvador Ros

Guía del Desarrollador de la empresa Borland Inprise.

Programación con Delphi 5 de Francisco Charte

Diseño de Programas: Formalismo y Abstracción, de Ricardo Peña Marí.

Además, si queréis ampliar vuestros conocimientos sobre tipo abstractos de datos y algoritmos, el libro del profesor Luis Joyanes Aguilar e Ignacio Zahonero Martínez, “Estructura de datos, Algoritmos, abstracción y objetos” de la Editorial McGrawHill. Uno de los libros más completos hasta la fecha, ilustrado con numerosos ejemplos en Pascal.