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

Objetos Auxiliares (y VII)

La última parada de este corto viaje... Nos dejamos para el final de la serie la implementación que hace Delphi en los TDA´s Pila y Cola. Y lo prometido es deuda.

Es nuestro séptimo y último capítulo, y un momento inmejorable para recapitular... ¿lo hacemos?

Atrás, muy atrás, han quedado las listas de punteros representadas en la clase TList, que fueron motivo de análisis en los dos primeros números de la serie. Avanzamos un paso adelante y nos introducimos en las listas de cadenas, cuyo mayor exponente pudiera estar representado en la clase TStrings. Eran el tercer y cuarto capítulo de la misma, y completaba lo que pretendía ser este acercamiento a algunos de los TDA´s más básicos. Recordad, además, que se completaba esta serie sobre lo que Borland definía como Objetos Auxiliares, con el artículo que nos brindaba nuestro compañero Carlos Conca en el número 4 de esta publicación: ¿Ficheros de configuración o Base de Registros?. En dicho artículo se comentarían aquellas clases especializadas en la manipulación de los valores del registro de Windows y ficheros Ini, como son TIniFile y TMemIniFile, o TRegistry y TRegistryIniFile.

También hemos podido compartir con nuestro incansable e incombustible compañero Mario Rodríguez, bien acentuado en la í para que no se me queje :-) , la visión que nos proporcionaba la óptica de otro compilador, como lo es C++ Builder sobre los mismos TDA´s.

Y ya para terminar, abordamos un pequeño ejemplo que nos dejó en el quinto y sexto capítulo unas pinceladas de uno de los principales descendientes de TStrings, la clase TStringList.

Y aquí, después de seis agotadores e interminables capítulos, nuestra parada merecida. Este va a ser el punto final (¿existe ese punto?) a nuestro pequeño viaje.

¿Y la alegría...?

Dejémonos de tristezas; ya me conocéis, e intentemos dar un toque de gracia a este último rato que vamos a compartir: un poco de alegría para unos botones que corren los cien metros lisos...

TOrderedList: nuestro primer objetivo.

Iniciamos estas lineas con un comentario previo: La unidad **contnrs.pas** es introducida a partir de la versión 5 de Delphi. Antes de esta versión no existía, como no existían ninguna de las clases que introduce dicho módulo. Así pues, y dada la imposibilidad de distribuir con las fuentes dicha unidad, en principio el ejemplo tan solo se "podría" compilar desde dicha versión de Delphi... Así, por favor, tenedlo en cuenta.

Recordáis la figura 1 del primer artículo de la serie...?

Yo desde luego, no. Así que me he cogido dicho ejemplar y lo he abierto por la página 11 y se lee al pie de la figura: "Relación jerárquica de los objetos relacionados con las listas". En ella podemos observar como las clases TQueue y TStack son descendientes de TOrderedList, y esta última, cliente de TList. Es decir: TOrderedList, que desciende directamente de TObject, hace uso de una lista de punteros. Y las clases TQueue y TStack son descendientes de la anterior. Es así de sencillo pero quizá en este punto valdría la pena preguntarse el por qué. La justificación queda más o menos clara tras analizar la declaración que hace

la clase TList y compararla con la que hace la clase TOrderedList. Y es que, a nuestra lista ordenada se le piden cosas distintas y una gran parte de los métodos que publica TList ya no se hacen necesarios. No necesitamos ordenamiento alguno... sobraría entonces el método Sort... No necesitamos intercambiar referencias... sobraría Exchange( )... Ni borrado... sobraría Delete( )... etc...

En realidad, lo único que se le pide a la clase TOrderedList es que implemente las acciones que han de caracterizar de forma conjunta a los TDA´s Pila y Cola: ser una estructura lineal (LIFO en el primer caso y FIFO en el último).

Así pues, la solución inmediata es que la clase TOrderedList haga uso de una variable de tipo TList, donde pueda almacenar todos y cada uno de los punteros de forma ordenada. Y declarando dicha variable como campo privado de nuestro nuevo objeto, conseguimos que todos los métodos que hace públicos TList queden ahora ocultos.

¿Vemos la especificación?

```
TOrderedList = class(TObject)
private
  FList: TList;
protected
  procedure PushItem(AItem: Pointer); virtual; abstract;
  function PopItem: Pointer; virtual;
  function PeekItem: Pointer; virtual;
  property List: TList read FList;
public
  constructor Create;
  destructor Destroy; override;

  function Count: Integer;
  function AtLeast(ACount: Integer): Boolean;
  procedure Push(AItem: Pointer);
  function Pop: Pointer;
  function Peek: Pointer;
end;
```

Ahora posiblemente se puede observar con más claridad. ¿Os dáis cuenta de que el método PushItem, declarado como abstracto y virtual en TOrderedList, va a ser el único método que sobrescriben nuestros dos descendientes?

TStack por un lado:

```
TStack = class(TOrderedList)
protected
  procedure PushItem(AItem: Pointer); override;
end;
```

Y TQueue por otro:

```
TQueue = class(TOrderedList)
protected
  procedure PushItem(AItem: Pointer); override;
end;
```

La especificación Pila nos pide que cada elemento (puntero) añadido a la lista, permanezca en la cima de ella (LIFO – El último en entrar es el primero en salir) en tanto no sea añadido un nuevo elemento, y no sea accesible el resto de elementos si no es retirado de la misma. Ahora quedan sin sentido cualquier método que intente acceder al interior de la lista, por definición.

Es fácil de comparar estos puntos si analizamos la implementación que hace TStack del método PushItem( ):

```
procedure TStack.PushItem(AItem: Pointer);
```

```
begin
  List.Add(AItem);
end;
```

Añadimos al final de la lista la nueva variable de tipo Pointer, o puntero, obteniendo una nueva CIMA. No hay mas...

Por el contrario, la especificación Cola nos pide que cada elemento (puntero) añadido a la lista, sea insertado en la primera posición, y no al final de misma como sucedía en el TDA Pila. Si el TDA Pila simbolizaba visualmente una colección de objetos apilados unos sobre otros, que hacen imposible la retirada de un elemento inferior si no son retirados antes los posteriores, el TDA cola y por poner un ejemplo más real se asemejaría a la cola de personas que se produce ante la llegada del autobús, y que esperan impacientes el turno para subir en el mismo. Los primeros en incorporarse a ella van a ser los primeros que la abandonen y cualquier intento de un elemento por adelantar posiciones puede ser recriminado por el resto de elementos.

```
procedure TQueue.PushItem(AItem: Pointer);
begin
  List.Insert(0, AItem);
end;
```

El nuevo elemento es claramente insertado en la primera posición y será forzosamente el último en salir de la misma. Cualquier otra inserción, adelantará en una posición el elemento añadido anteriormente.

Como nota aclaratoria y para no liarnos en la consideración del primer o último elemento de la lista, establezco como primer elemento de la misma la posición cuyo indice toma como valor 0. El último elemento de la lista, según este criterio, habría de ser aquel que representara la posición enésima en dicho vector. A este último elemento le denominaremos CIMA. Dicho esto, nos será más fácil entendernos.

Puntos comunes: un lugar de encuentro...

Podemos, y lo vamos a hacer, proceder al análisis de la implementación que se hace en cada uno de los métodos de la clase TOrderedList, de forma análoga a como lo hemos hecho en artículos anteriores. Tenemos, o tengo, que suponer que el lector, y hablando de forma menos impersonal, tú que vas a sentarte a leer estas lineas que estamos compartiendo, ya conoces los métodos que publica la clase TList. De no ser así, y dado que los voy a nombrar sin demasiado entretenimiento, te rogaría que hicieras una lectura previa de los dos primeros artículos.

Sin más preámbulos podemos contemplar la creación y destrucción del objeto motivo de nuestro estudio.

```
constructor TOrderedList.Create;
begin
  inherited Create;
  FList := TList.Create;
end;

destructor TOrderedList.Destroy;
begin
  List.Free;
  inherited Destroy;
end;
```

Nada tiene de especial que le haga blanco de nuestro interés. Se crea y se destruye el objeto de clase TList y se hace necesaria en ambos casos, una llamada al constructor y destructor heredado. En el primer caso, construirá la parte del objeto sobre la que nos apoyamos para construir el nuestro, el ascendiente. En el segundo caso, hará posible la destrucción de dicho ascendiente toda vez que hemos destruido la lista. En

cierta forma se comparte un mismo pensamiento: “ocúpate tú de tu parte que yo me ocupo de la mía...”. Así de claro, ¿no? :-)

Añadir un elemento a la lista.

Comentábamos anteriormente que tanto la clase TStack como TQueue, redefinían un método declarado como abstracto en la clase que nos ocupa. TOrderedList hace público el método para insertar un nuevo elemento en FList, y deja que sean ambos descendientes los que nos digan de que forma va a ser insertado el nuevo elemento: al principio de la lista o al final de la misma.

Estamos hablando del método Push( ):

```
procedure TOrderedList.Push(AItem: Pointer);
begin
  PushItem(AItem);
end;
```

PushItem( ), como método abstracto que es, además de virtual, permite ser redefinido y lo que es más importante: ser considerado como método abstracto delega la implementación del mismo en los descendientes. De hecho, esta clase no podría por dicho motivo ser instanciada sin generar la consabida excepción... No. Esto no es del todo cierto. Está mal interpretado. Es lo que diría posiblemente en una tertulia entre amigos. Realmente podríamos instanciar un objeto que contenga métodos abstractos.

Suponed que hacemos una implementación tal que así al pulsar un objeto Button1:

```
var
lista: TOrderedList;
begin
    lista:= TOrderedList.Create;
    try
    ShowMessage(IntToStr(lista.count));
    finally
    lista.free;
    end;
end;
```

Es decir que la pulsación del botón nos mostraría que la lista contiene 0 elementos. No habría excepción de ningún tipo. Eso sí, nuestro compilador nos mostraría un hermoso Warning que no nos impediría lanzar la ejecución del programa:

```
[Warning] Unit1.pas(31): Constructing instance of 'TOrderedList' containing abstract methods
```

Sin embargo, cualquier intento de usar el método declarado como abstracto :

`lista.Push(button2);` en lugar de `ShowMessage(IntToStr(lista.count));`

Sí generaría la excepción. Concretamente:

```
Project Project1.exe raised exception class EAbstractError with message 'Abstract Error'.
```

Consultar un elemento de la lista.

Sabiendo que la única diferencia real entre un TDA Pila y un TDA Lista es la forma en que se van a introducir cada elemento nuevo, podemos entender que sea compartida la forma en la que se obtiene la cima de la lista de punteros. En ambos casos vamos a obtener un puntero al último de los elementos que

componen la lista. El método Peek invocará en su implementación a PeekItem, declarado como virtual en la zona protegida de la clase. Esto nos permite su redefinición en un descendiente. ¡Bueno es saberlo!

```
function TOrderedList.Peek: Pointer;
begin
  Result := PeekItem;
end;

function TOrderedList.PeekItem: Pointer;
begin
  Result := List[List.Count-1];
end;
```

Aquí podemos recordar como el dominio de cualquier propiedad matricial se iniciaba en cero y se extendía hasta (n-1). Como valor de retorno, obtenemos el puntero al elemento esperado. Por eso debemos saber también que, de intentar obtener la CIMA de la pila o de la cola en situaciones en los que no contengan algún elemento, obtendríamos un hermosa excepción lanzada por FList.

```
Project Project1.exe raised exeption class EListError with message 'List Index  Out of
Bounds (-1)'...
```

Este mensaje es obtenido desde el IDE de Delphi. La ejecución del exe fuera del compilador simplemente haría emerger una ventana remarcando el texto encerrado entre comillas simples, como bien es sabido.

Pocas explicaciones merecen: Si (Count = 0), que es precisamente el valor que tiene dicha propiedad cuando tanto la Pila como la Cola no contienen elemento alguno, intentaríamos acceder a posiciones del vector para las que se establece en la implementación de la clase TList el lanzamiento de una excepción.

Extraer un elemento de la lista.

Hemos hablado de consultar el valor del elemento CIMA. Necesitamos lógicamente también implementar un método que nos permita eliminar los elementos de la lista. En ambos casos, por definición ha de ser eliminado tan solo el elemento CIMA. Al igual que anteriormente, la interfaz declara un método público Pop, y un método protegido PopItem, que podrá ser redefinido en un descendiente.

```
function TOrderedList.Pop: Pointer;
begin
  Result := PopItem;
end;

function TOrderedList.PopItem: Pointer;
begin
  Result := PeekItem;
  List.Delete(List.Count-1);
end;
```

Sencillo, ¿no...?

En primer lugar consultamos el elemento CIMA, obteniendo un puntero hacia el mismo. En segundo lugar invocamos el método Delete de TList. Y ¿sabéis lo que pasa cuando se invoca el método Pop sobre una lista con 0 elementos?

Efectivamente. Es de recibo lo dicho anteriormente para Peek. Obtenemos una hermosa excepción en la que se comunica que el valor (-1) esta fuera de rango: 'List Index of Bounds (-1)'.

Así pues, si queremos dejar de obtener estas magníficas alertas deberemos comprobar que tanto nuestro objeto Pila como nuestro objeto Cola, contienen al menos algún elemento

Obtener el número de elementos y otras comprobaciones.

Imagino que se entiende.

```
función TOrderedList.Count: Integre;
begin
  Result := List.Count;
end;
```

No creo que nadie levante la mano para pedir que se lo explique... ;-)

Sin embargo también se nos ofrece un método para obtener si la posición del elemento consultado (el parámetro que recibe de entrada) es menor o igual al número de elementos actual. En mi humilde opinión, que no deja de ser la de un aficionado con muchas ganas de compartir, no se estrujaron demasiado la cabeza y casi por pitorreo lo dejaron caer:

```
function TOrderedList.AtLeast(ACount: integer): boolean;
begin
  Result := List.Count >= ACount;
end;
```

Supongamos que recibe como parámetro el valor 7; de tener 8 elementos nos devolvería Verdadero. Sin embargo para valores menores que cero también devuelve Verdadero. ¡Bonita cuestión... como diría mi buen amigo Mario!.

Un poco de ejercicio por favor...

¿Queréis que practiquemos un poco con todo esto que hemos comentado? Un poco de ejercicio no nos vendrá nada mal. Así que hemos sacado unos cuantos botones a calentar y andan por ahí estirando músculos en el centro de la pista.

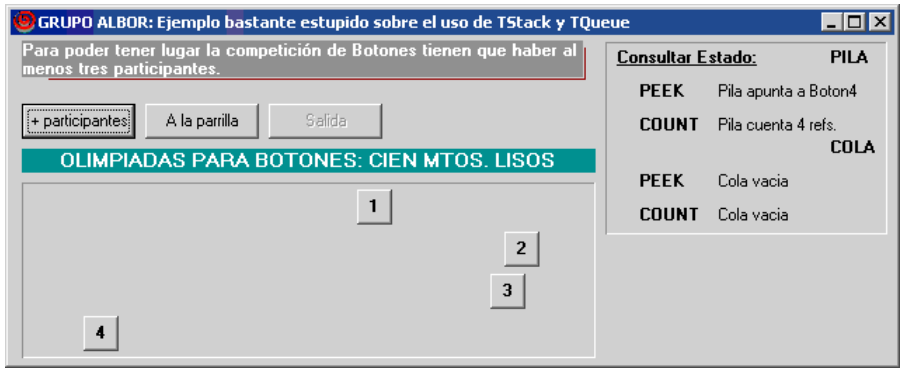

Figura 1 - Figurada carrera de botones. Interfaz gráfico

Si le echáis un vistazo a la **figura 1** y al **listado 1** lo váis a entender. No hay desperdicio.

La idea que mueve este pequeño ejemplo es simplemente la de practicar un poco con los métodos que publican tanto la clase TStack como la clase TQueue. No vamos a detallar paso a paso cada uno de los métodos que se han implementado. De hecho, comento en el código fuente de la unidad con cierto detalle cada una de las lineas que se han escrito, por lo que en cierta forma resulta innecesario repetirlo aquí.

```
unit stack_queue;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, contnrs, ExtCtrls;

const
   max_participantes = 8;   //numero máximo de botones participantes
   separacion = 5;          //máxima separación entre botón y botón
type

  TTimerList = class(TTimer)
  private
    FLista: TList; //lista que alberga una referencia a cada uno de los botones
  public
    constructor Create(aOwner: TComponent); override;
    destructor Destroy; override;
    property Lista: TList read FLista write FLista;
  end;

  TfrmOlimpiadas = class(TForm)
    ...
    ...
    ...
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btb_crearClick(Sender: TObject);
    procedure btb_salidaClick(Sender: TObject);
    procedure btb_parrillaClick(Sender: TObject);
    procedure btb_cancelarClick(Sender: TObject);
  private
    Lista_botones: TTimerList; // nos ayuda en el movimiento de los participantes.
    pila: TStack;  // almacena los participante y facilitará su alineación
    cola: TQueue;  // almacena la llegada de los participantes y emite resultado
    corredores: Integer; //número de corredores que han llegado a meta
    procedure TimerAction(sender: TObject);  //Evento on Timer
    procedure MostrarResultados; //muestra los resultados
    procedure ReiniciarJuego;    //para reiniciar el juego
  public
  end;
```

Listado Nº 1: Declaraciones de tipos en las clases TTimerList y TfrmOlimpiadas.

Sí relataremos, aunque sea de forma breve, lo que hace el programa, de forma que podáis tener una mejor idea del mismo, previa a la lectura de las fuentes. La idea principal era hacer uso de los principales métodos que publican tanto la clase TStack como la clase TQueue, y para eso, implementamos una ficticia carrera en la que los participantes habrán de ser botones, pequeños y bien cuadrados. Perdonad si es una solemne majadería pero es que no se me ocurría otro ejemplo mejor. ¿vale? :-)

Así pues, el procedimiento btb_CrearClick, ejecutado al pulsar el botón para añadir un nuevo participante, ha de crear las nuevas instancias de TButton, que representa cada nuevo participante hasta un número máximo de ocho. En este punto damos ya uso a nuestra variable de tipo TStack que será incrementada a medida que se incorpora un nuevo corredor a la figurada carrera. Hacemos uso de los métodos Push( ) y Peek en la instancia de la variable Pila.

Posteriormente, nos podrá servir está lista (pila) para reordenar cada uno de los botones creados en la linea de salida. Veámoslo:

```
procedure TfrmOlimpiadas.btb_crearClick(Sender: TObject);
var
boton: TButton;
begin
   // solo mientras no rebasemos el número de participantes máximos
   // Hay que tener en cuenta que en este momento Count de la pila
```

```
   // devuelve 0 por lo que iniciamos la cuenta desde dicho valor
   if pila.Count <= max_participantes - 1 then
      begin
      boton:= TButton.Create(self); // creamos el botón (CON PROPIETARIO)
      Lista_Botones.Lista.Add(boton);  // y lo añadimos al temporizador
      boton.parent:= self; //  un padre para el nuevo botón
      boton.name:= 'Boton' + IntToSTr((pila.count + 1)); // y nombre y todo
      boton.caption:= IntToStr(pila.count + 1);  // un rótulo numérico
      boton.Font.Style:= boton.Font.Style + [fsBold]; // que se vea bien !!!
      // cuadradito to...
      boton.height:= 25;
      boton.width:= 25;

      if pila.Count >= 0 then // si hay referencias en la pila
         begin
         // fijamos la posición en altura del boton
         boton.top:= bevel1.top +
                     separacion +
                     (separacion * pila.count) +
                     (pila.count * boton.Height);
         // ajustamos la altura del marco también
         bevel1.Height:= separacion +
                         (separacion * (pila.count + 1)) +
                         ((pila.count + 1) * boton.height);
         // lo dejamos entrenar en la pista hasta la salida
         boton.left:= random( bevel1.width - bevel1.left - boton.width) +
                      bevel1.left +
                      separacion;
         // y dimensionamos correctamente el form
         frmOlimpiadas.height:= bevel1.top +
                                bevel1.height +
                                boton.height +
                                separacion;
         end;
      pila.push(boton); // añadimos una referencia a la pila para finalizar
      // actualizando los avisadores
      lab_ppeek.caption:= 'Pila apunta a ' + TButton(pila.Peek).name;
      lab_pcount.caption:= 'Pila cuenta ' + IntToStr(pila.count) + ' refs.';
      // y el número de participantes en la carrera
      Inc(corredores);
      end;
   // solo dejamos iniciar la salida si hay más de tres corredores
   btb_parrilla.enabled:= (pila.count >= 3);
end;
```

Si disponemos de, al menos, tres botones creados, podremos hacer uso del procedimiento btb_parrillaClick, asignado al evento OnClick del botón cuyo caption es 'A la parrilla'. El uso de la pila nos facilita obtener una referencia rápida a cada uno de los botones participantes, tras la invocación del método Pop. Esto tiene como consecuencia lógica que a medida que resultan alineados los botones en la linea de salida, la pila será vaciada.

Para poder entonces realinear en dicha linea de salida, toda vez que se haya iniciado la carrera y deseemos cancelar la misma mediante el botón cuyo *caption* es “Cancelar”, se hará preciso pues, restaurar la pila apuntando a cada una de las referencias, tal y como estaba antes de iniciarse la misma.

Otra cosa reseñable es que se puede hacer preciso en muchas ocasiones contar los elementos que contienen la pila y evaluar si ésta está vacía o no. Es lo que hacemos cuando escribimos la condición que aparece en la primera linea (Pila.Count > 0) ¿Hay algún elemento en la pila?.

Esa será nuestra condición para seguir extrayendo referencias a los botones participantes que nos permitan ordenarlos.

```
   while (Pila.Count > 0) do
      TButton(Pila.Pop).Left:= Bevel1.Left + separacion;
```

```
procedure TfrmOlimpiadas.btb_parrillaClick(Sender: TObject);
begin
   // estado de los botones
   btb_salida.enabled:= (Pila.Count > 0); // salida activo si hay referencias
   // una vez alineados, ya no es tiempo de realinear o añadir mas botones
   // Se ha cerrado la participación...
   btb_crear.enabled:= False;
   btb_parrilla.enabled:= False;
   // procedemos a ordenar los botones
   while (Pila.Count > 0) do
      TButton(Pila.Pop).Left:= Bevel1.Left + separacion;
   // y anunciamos la nueva situación de la pila
   lab_ppeek.caption:= 'Pila vacia';
   lab_pcount.caption:= 'Pila vacia';
end;
```

Nos falta dar el pistoletazo de salida. Vía libre... Se ha iniciado la carrera.

```
procedure TfrmOlimpiadas.btb_salidaClick(Sender: TObject);
begin
   Lista_botones.Enabled:= True; // temporizador activo
   btb_salida.Enabled:= False;   // desactivamos la salida
   btb_cancelar.visible:= True;  // permitimos cancelar si es necesario
end;
```

Un poco de movimiento por favor.

Toda vez que se ha iniciado la carrera, hemos llegado al procedimiento que nos sirve como motor del movimiento de los corredores en la pista: nos basta una sencilla invocación del generador de números aleatorios Random( ) para incrementar el avance de cada participante. Estamos hablando del evento OnTimer y del procedimiento implementado para el temporizador que lo genera.

No comentaría quizás este procedimiento si no fuera porque nos sirve para hacer uso de la clase TQueue. Para nosotros en este momento, la variable Cola nos ayudará a capturar la entrada de cada uno de los participantes por linea de meta. De igual forma nos va a ayudar a montar el resultado final de la carrera de botones pero eso lo podremos ver en el procedimiento *MostrarResultados*..

```
procedure TfrmOlimpiadas.TimerAction(sender: TObject);
var
xIndice, yIndice: Integer;
boton: TButton;
begin
   Application.ProcessMessages;
   //mientras existan botones en el temporizador
   if Lista_Botones.Lista.Count > 0 then
      begin
      //recorremos la lista para obtener una referencia a ellos
      for xIndice:= 0 to Lista_Botones.Lista.Count - 1 do
         begin
         Application.ProcessMessages;
         //obtenemos la referencia al objeto actual
         boton:= TButton(Lista_Botones.Lista[xIndice]);
         //si no ha llegado a META (me vale tocarla con el extremo)
         if boton.Left + boton.Width <= bevel1.width then
            //procedemos a desplazarlo aleatoriamente
            boton.Left:= boton.Left + Random(15)
```

```
         else  //en caso contrario
            begin
            {aprovechamos el tag del boton para detectar si el
             objeto a llegado a meta. Si lo ha hecho, su tag
             valdrá 1}
            If boton.tag = 0 then    // si no ha llegado a meta
               begin
               // lo incorporamos a la cola de resultados
               cola.Push(Lista_Botones.Lista[xIndice]);
               // lo anunciamos en la etiqueta la incorporación
               lab_cpeek.caption:= 'Cola apunta a ' + TButton(cola.Peek).name;
               // actualizamos la etiqueta contador de referencias
               lab_ccount.caption:= 'Cola cuenta ' + IntToStr(cola.count) + ' Refs.';
               // minoramos en un corredor los participante ptes. de llegada
               Dec(corredores);
               If corredores = 0 then  //si han llegado todos
                  begin
                  Lista_Botones.Enabled:= False; // desactivamos el temporizador
                  // los resituamos en la linea de salida
                  for yIndice:= 0 to Lista_Botones.Lista.Count - 1 do
                     TButton(Lista_Botones.Lista[yIndice]).Left:= bevel1.Left +
                                                                  separacion;
                  // mostramos los resultados de la carrera
                  MostrarResultados;
                  // y reiniciamos el juego por si se quiere repetir
                  ReiniciarJuego;
                  Exit; // ¡¡vamonos fuera de procedimiento !! ¡¡hemos acabado!!
                  end;
               // este boton ya ha cruzado la META
               if Assigned(boton) then boton.tag:= 1;
               end;
            end;
         end;
      end;
end;
```

Y tan solo nos queda comentar el procedimiento que muestra los resultados y que es invocado tan pronto como han sobrepasado la linea de meta todos los botones. Igualmente podemos hacer uso de nuestro objeto TQueue para obtener cada uno de los participantes en el mismo orden en que han llegado a meta. Extraemos sucesivamente y mientras Count sea mayor que 0, los punteros hacia cada uno de los botones. Montar la cadena para mostrarlo y poco mas.

```
procedure TfrmOlimpiadas.MostrarResultados;
var
cadena: String;
ganador: String;
contador: Integer;
begin
   // ¿queremos ver los resultados?
   If Messagedlg('¿Quieres ver los resultados de la carrera?:', mtInformation, [mbOk,
mbCancel], 0) = mrOk then
         begin
         cadena:= '';
         contador:= 0;
         // obtenemos el nombre del objeto "cima" de la "cola".
         // Peek nos devuelve una referencia al mismo que moldeamos para
         // obtener dicho nombre. El puntero NO es removido, solo consultado
         ganador:= 'EL GANADOR ES ' +  TButton(cola.Peek).Name;
         // mientras queden referencias
         while cola.Count > 0 do
            begin
            Inc(contador);  // nos da la posición de llegada
            // vamos extrayendo (POP) y eliminando cada una de las
            // referencias, (el orden es el de llegada)
```

```
            cadena:= cadena + IntToStr(contador) + ': ' + TButton(cola.Pop).Name + #13#10;
            end;
         cadena:= cadena + #13#10 + ganador;
         // y visualizamos toda la cadena de resultados
         ShowMessage('Resultados de la carrera: '#13#10 + cadena);
         end
   else // si no queremos ver los resultados también...
         while cola.count > 0 do cola.pop; //vaciamos la "cola"
end;
```

Resumiendo. Estos son los pasos que hemos seguido y que posteriormente, si lo deseáis, tenéis en las fuentes que acompañan al artículo:

- Creación de los botones. Se incrementa la pila con una referencia a los mismos.
- Mediante la pila reordenamos los botones en la linea de salida.
- La carrera tiene lugar... aun puede ser cancelada. La llegada a meta incrementa la cola con una referencia a cada uno de los botones.
- Hacemos uso de la cola de resultados para obtener el orden de llegada a Meta y visualizarlo al usuario.

Vamos finalizando: dos reflexiones rápidas y acabamos...

La primera reflexión que se me ocurre, es apreciar la facilidad con la que podemos incorporar el uso de listas de punteros a nuestros desarrollos. En nuestro caso particular, el que nos ocupa en estos momentos, queda simplificado más aun, dado que el objeto asociado a ellas establece un propietario, *owner*, con la responsabilidad de su destrucción. Nos despreocupamos de una gran parte de trabajo. Si recordáis también, en algunos de los ejemplos que hemos expuesto en capítulos anteriores, al contrario que el actual, hacíamos uso de estructuras y reservábamos para ellas memoria dinámica. Eso, lógicamente nos obligaba a cerciorarnos de que la memoria asociada a la misma era liberada antes de la destrucción de la lista. Ahora no nos ocurre y será así siempre y cuando dichos objetos sean descendientes de la clase TComponent, la primera clase que implementa el owner o propietario.

Para los que nos iniciamos, estas unidades que resultan ciertamente más sencillas que las anteriormente vistas, ofrecen pequeños detalles que es bueno empezar a apreciar. Cabría reflexionar sobre como se ha declarado un método publico, envolviendo al método protegido y virtual. Era el caso del método *Pop*. Detrás de su invocación se escondía una llamada a *PopItem*, declarado como protegido y virtual. Esto nos permitirá en un futuro redefinir el mismo en un descendiente.

...

Bueno. Ahora sí que tenemos que despedirnos. Ahora ya no os puedo decir eso de: ¿seguro que esta historia acaba aquí...? ;-)

Me siento muy honrado de haber compartido este rato con todos vosotros. Recibid un abrazo,