Abstract

A few weeks ago, I was faced with the problem of designing a component having a list of integer as a property. This is quite easy to write, but once installed in the IDE, this property cannot be edited using the object inspector! Actually, the property is not even serialized in the DFM.

After a few Google searches, I came to the conclusion that no code was readily available for the purpose. So I wrote it and now I'm making it available thru this blog article.

This article is made of:

1. A simple custom visual component using a list of integers as property. This component has been made very simple (and mostly useless) so that you can concentrate of our today's subject: the property made of a list of integers.
2. A simple demo application showing the component in action at runtime and reusing the property editor in the application (normally a property editor is only used by the IDE).
3. A class to implement the list of integers. It looks much like TStringList class. Instead of a list of strings, we get a list of integers.
4. A property editor so that the list of integer could be edited from the object inspector at design time.

Source code: http://www.overbyte.be/frame_index.html?redirTo=/blog_source_code.html

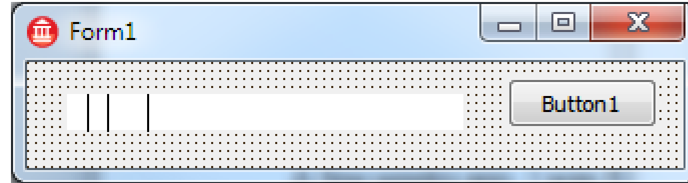
About the author

Francois Piette, an Embarcadero MVP, is well known for his freeware Internet Component Suite (ICS) and his multitier framework MidWare. Francois provides Delphi & C++ Builder consultancy as well as custom development. He can be reached at http://www.overbyte.be.

Visual component having a TIntegerList property

Before talking about the TIntegerList property and his property editor, I would like to show a component using this kind of property. Originally I had to write a rather complex component. This is not interesting to see that component here because our subject would be hidden in a lot of unrelated code. So instead, I made a very simple custom visual component.

I named this component TTickLine because it draws a line of ticks. Each tick is positioned by his coordinate relative to the left corner. This is a perfect case for a TIntegerList: each integer in the list represents the position of a single tick.



Shown with 3 integers in the list

Creating such a component with Delphi is easy using the component wizard: Menu / Component / New component. Then select ancestor TCustomControl, select a class name (here TTickLine), select a palette page (The place where you want to see the component in the IDE), select unit name (Here: TickLineControl.pas) and finally install to either an existing package or to a new one.

The wizard creates the code for us. We have just to add a few lines: the property itself, a constructor, a destructor and a paint procedure.

We need two code lines to define the property: one in the private section and one in the published section.

```
  private
      FTickPos : TIntegerList;
  published
      property TickPos : TIntegerList read FTickPos write SetTickPos;
```

To allocate actual storage to the TIntegerList, we must add a line in the constructor. While we are in the constructor, we can add a few integers in the list. Those will be the default value for the property. Another one is required in the destructor to free the allocated space.

```
constructor TTickLine.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    FTickPos := TIntegerList.Create;
    FTickPos.Add(10);
    FTickPos.Add(20);
    FTickPos.Add(40);
    Width  := 200;
    Height := 20;
end;
```

Of course, defining a property and assigning a value to it doesn't make any effect. We have to use the value somewhere. Here, since we have a visual component, we use the value in the Paint procedure to draw vertical lines whose number and position is given by the list of integers.

The Paint procedure has to draw everything required to make the component looks what we want it to look. In our case, it is very simple: A simple rectangle with no boundary and as much lines as we have integers in our property.

All in all the Paint procedure looks like this:

```
procedure TTickLine.Paint;
var
    I : Integer;
    X : Integer;
begin
    Canvas.Pen.Style := psClear;
    Canvas.Rectangle(0, 0, Width - 1, Height - 1);
    if Assigned(FTickPos) then begin
        Canvas.Pen.Style := psSolid;
        for I := 0 to FTickPos.Count - 1 do begin
            X := FTickPos[I];
            Canvas.MoveTo(X, 0);
            Canvas.LineTo(X, Height - 1);
        end;
    end;
end;
```

To install the component, just build the package you selected when using the component wizard and install the package (Right click in the project manager and select "install"). As it is now, you cannot edit the property we created. We need the property editor. Be patient, we will describe it later in this article.
For your reference, the full source code is published at the end of the article.

A simple demo application

The simplest demo is made of an empty form with our TTickLine control dropped on it. To create the demo, or to open it if you downloaded the source code, you must first install the component: The IDE itself cannot show the component without having it installed.

To create the demo, simply create a new VCL form application and drop our TTickLine control on it. It should immediately show the few lines corresponding to the integer list created in the constructor.

No need to add any code manually. Just compile and run and verify that the component is visible with his tick lines displayed.
Later we will make this demo a little bit more demonstrating by adding an editor and also some code to demonstrate component serialization.
For your reference, the full source code is published at the end of the article.

A class to implement the list of integers

Here begins the interesting part! We will now create a class to implement a list of integer so that this list is suitable as a component property with a property editor to be able to define the integers in list while at design time, using the object inspector.

The first requirement for a class which must be usable as a component property which you can see and use at design time is the ancestor. Your class must derive, directly or indirectly, from TPersistent.

This requirement is somewhat annoying because the easiest way of creating a list of integer is to use a TList derived class. Thanks to the generics implementation in Delphi, a single line is enough:

```
  MyList: TList; // Must add Generics.Collections in the uses clause
```

If we use this simple approach, the property will not be usable in the object inspector and will not even be serialized in the DFM! This is because TList don't have TPersistent in his ancestors. We must derive from TPersistent or anything itself deriving from TPersistent. So let's do it. It's easy:

```
type
    TIntegerList = class(TPersistent)
    end;
```

Of course with that definition alone, we don't have any list. So let's add a field which is a list of integers, add a property exposing the list as an array of integers (indexed property) and add methods to add or remove integers to/from the list. Here is the declarations we need:

```
TIntegerList = class(TPersistent)
protected
    FList : TList;
    function GetItems(nIndex: Integer): Integer; virtual;
    procedure SetItems(nIndex: Integer; const Value: Integer); virtual;
public
    constructor Create; virtual;
    destructor Destroy; override;
    function Add(const Value : Integer) : Integer; virtual;
    procedure Delete(Index: Integer); virtual;
    property Items[nIndex : Integer] : Integer read  GetItems
                                               write SetItems; default;
end;
```

The FList variable is the list, built using Delphi generics. If you have an old Delphi, you can create such a class by hand, you'll find a lot of old article explaining how. Generics is really one of the feature which by itself alone justify the upgrade to the current Delphi version!

As you probably know, the base TList is a kind of dynamic array of pointers. A TList<Integer> simply replace all pointers by integers. Every method or property to handle the list is automatically converted to use an integer instead of a pointer. Incredibly easy and powerful!

To expose our FList variable as a kind of array of integer, we need to define an indexed property and make it the default property:

```
property Items[nIndex : Integer] : Integer read  GetItems
                                           write SetItems; default;
```

We you have defined such a property; you can then use it without specifying his name (Items) because it is the default property. So in our TTickLine demo component, since his property TickPos is of type TIntegerList, we can simply write: `TickPos[2] := 27;`
The compiler will translate this line as if you wrote `TickPos.SetItems(2, 27);`

Writing the setter and getter is trivial. They just access the underlying TList items:

```
function TIntegerList.GetItems(nIndex: Integer): Integer;
begin
    Result := FList[nIndex];
end;

procedure TIntegerList.SetItems(nIndex: Integer; const Value: Integer);
begin
    FList[nIndex] := Value;
end;
```

Wait! We are accessing the FList variable. But it is a class. We need to create an instance in the constructor and free it in the destructor.

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

destructor TIntegerList.Destroy;
begin
    FreeAndNil(FList);
    inherited Destroy;
end;
```

Finally we must implement the Add and Remove methods to add an integer to the list and remove an integer from the list. Here again it is trivial: the methods are directly mapped to the same methods of the underlying FList variable:

```
function TIntegerList.Add(const Value: Integer): Integer;
begin
    Result := FList.Add(Value);
end;
```

```
procedure TIntegerList.Delete(Index: Integer);
begin
    FList.Delete(Index);
end;
```

Actually, TList has a lot of methods to manipulate the list's elements. In the final code, available at the end of this article, you'll see that I implemented many of those useful methods. They are transfer methods like Add and Remove we have seen above.

If you compile the package with that new version, you'll see that it still doesn't work as expected in the object inspector because the runtime routines have no idea about our property. It is not one of the known base types. No problem, make it available!

To make a custom property available, it is enough to override the existing method DefineProperties. This is where things become more complex...

DefineProperties will be called when components are serialized or deserialized, typically when the IDE create the DFM file or at runtime when a form is loaded. A form actually read the data from the DFM file which is compiled as a resource and bundle in the executable file.

DefineProperties will receive the a TFiler which will do the actual I/O. DefineProperties must call the filer for each property to define, passing the name of the property, a method ReadData to read data, a method WriteData to write data and a method HasData to tell if there is data or not.

The three methods will be called at appropriate time to read/write data, each one receiving the reader or writer component instance used to do the I/O. Reader and Writer components have methods to read/write a few basic data types such as the integer we need.

Once we know all those features, we can write code. First let's write code for WriteData:

```
procedure TIntegerList.WriteData(Writer: TWriter);
var
    I : Integer;
begin
    Writer.WriteListBegin;
    for I := 0 to Count - 1 do
        Writer.WriteInteger(FList[I]);
    Writer.WriteListEnd;
end;
```

WriteData is really simple: a for loop is used to write all integers in the list using the writer object. The loop is between two calls used to begin and end a list of values.

If you look at the DFM of a form containing our component, you'll see it is serialized as:

```
  object TickLine1: TTickLine
```

```
    Left = 8
    Top = 43
    Width = 257
    Height = 25
    TickPos.Integers = (
      10
      20
      40
      15)
  end
```

The lines starting with "`TickPos.Integers`" is the part created by DefineProperties. Here it is shown with 4 integers. By the way, to see the DFM as text, right click on the form and select "View as text".

ReadData is almost as simple. The code has to iterate thru the list of integer values presented by the reader and add the values to the list. Here again, the loop is surrounded by two calls to start reading a list and to end reading a list. Of course the list is cleared before reading any value.

```
procedure TIntegerList.ReadData(Reader: TReader);
begin
    Clear;
    Reader.ReadListBegin;
    while not Reader.EndOfList do
        Add(Reader.ReadInteger);
    Reader.ReadListEnd;
end;
```

DefineProperties is a little bit trickier to write. Basically it has to call Filer.DefineProperty, passing the 4 arguments as I explained above. The 4th argument is the "HasData" method which is called immediately. We will use a local method to have access to the filer argument.

Another difficulty is the ancestor issue. Filer has an "ancestor" property which may or may not be assigned. If the Ancestor property is assigned then data has to be read/written except if assigned value is our current instance. If ancestor is not assigned, then all we have to do is to look if our list is empty or not.

Assuming we want to name the property 'Integers', the code looks like:

```
procedure TIntegerList.DefineProperties(Filer: TFiler);

  function HasData: Boolean;
  begin
      if Filer.Ancestor <> nil then begin
          Result := TRUE;
          if Filer.Ancestor is TIntegerList then
              Result := not Equals(TIntegerList(Filer.Ancestor))
      end
      else
          Result := FList.Count > 0;
```

```
  end;

begin
    Filer.DefineProperty('Integers', ReadData, WriteData, HasData);
end;
```

With this code, we have now a much better component. The Integers property will appear in the object inspector and will be serialized to the DFM. But something is still wrong: the object inspector does not display the property value and refuse to edit it! This is because the IDE don't know how to edit a list of integers. To tell the IDE how to edit a list of integers, we have to write a so called "property editor" and register it.

There are a few enhancements required to make the component work as expected. I won't explain it in details here since it is not specific to our TIntegerList property.

a) Make sure the component is repainted each time the property is changed. For that, we call "Invalidate" at appropriate times.
b) Have a notification mechanism when the list changes. For that purpose, we implement an OnChange event. When the list has to be updated a lot of time, provide a BeginUpdate/EndUpdate mechanism to avoid triggering the OnChange to much.
c) Provide conversion to/from string: implement the methods ToString and FromString.
d) Provide an Assign method to help assigning the list of integers from another list of integers.

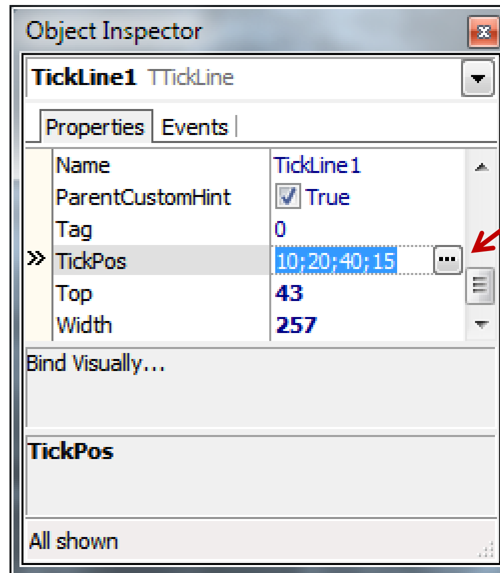
The complete code is available at the end of this article.

A property editor

A property editor is a piece of software that the IDE use to extend the object inspector and allow the user to manipulate a property in a way that is specific to the property data type.

Actually you use a property editor every time you use the object inspector! The object inspector is made of a collection of property editors for many kinds of data types. All property editors required for components delivered with Delphi are already in the box and you use them transparently. Of course when you build your own components, Delphi automatically reuses existing property editors. You need to create a property editor only when you design a property having a data type not already known by the IDE. And this is our case…

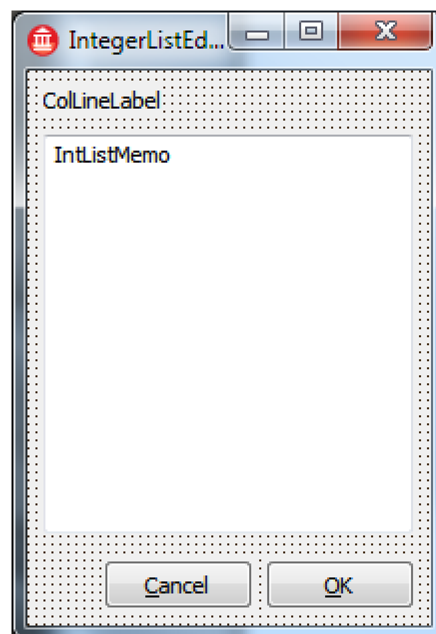
The existing property editor which is the closest to what we need for our TIntegerList property is the one used for TStringList property. When you use the object inspector to setup the lines in a TMemo component, you use the TStringList property editor.



Object inspector

In the next few paragraph, we will build a property editor suitable for our TIntegerList property. Visually, it is a simple form with a TMemo, an OK button and a cancel button. This form will pop up from the object inspector when you click on the […] button near the TIntegerList property.

The memo is filled by the integers currently in the list, one per line. You can add or remove integers at will. It is a memo and you can write whatever you like in it. Once you click on the OK button, the property editor check for the validity of each integer. In case of a bad integer, a message is display, the cursor is positioned where the bad integer has been found and the user must fix it before clicking OK again. Obviously, cancel button will dismiss any change done.



Property editor form

Writing such a property editor is actually very easy! This involves 3 steps:

1. Create a TForm as the editor as you like it to be.
2. Create a new class deriving from TPropertyEditor and add 4 methods: SetValue, GetValue, GetAttributes and Edit.
3. Register the property editor by including the unit into a package, and register it by creating a register procedure which calls RegisterPropertyEditor. Install the package.

First step: This is just plain basic Delphi programming: create a VCL form application and in that application create a form which looks and behave like you want your property editor to look and to behave. Don't forget to give that form a property of the type you want to edit. Also create two events one which is triggered when the OK button is clicked after integers are checked and one for the cancel button is clicked.

I will not enter in the details of this form since it is very basic. The resulting code can be found at the end of this article, it is less than 150 lines.

Second step: Create a new class deriving from TPropertyEditor and add the few methods required for the property editor to work. The methods are:

GetValue: This method returns the string that should be displayed by the Object Inspector. This gives the visual display of the property.
SetValue: This procedure is passed the new value as a string, and should update the actual property with this value.
GetAttributes: This method returns a set of attributes that the property editor has.
Edit: This method is called when the '...' button for the property is clicked in the object inspector. It displays the property editor dialog box, and sets the property to the new value when the dialog is closed (unless it is cancelled).

The resulting class declaration is as follow:

```
TIntegerListPropertyEditor = class(TPropertyEditor)
protected
    procedure EditDone(Sender : TObject);
public
    procedure SetValue(const Value: string); override;
    function  GetValue: String; override;
    function  GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
end;
```

As you can see, there is a 5th method (EditDone) which will be used as event handler for the editor form we created in the first step.

Let's have a look at the Edit method. Remember it is called when the user clicks on the […] button in the object inspector. We need to write it so that the editor form created at step one is displayed and receive a copy of the property value. The code is as follow:

```
procedure TIntegerListPropertyEditor.Edit;
var
    PropertyEditFrm: TIntegerListEditForm;
begin
    PropertyEditFrm := TIntegerListEditForm.Create(Application);
    try
        PropertyEditFrm.OnEditDone := EditDone;
        PropertyEditFrm.IntList    := TIntegerList(
                                 GetObjectProp(GetComponent(0), GetPropInfo));
        PropertyEditFrm.ShowModal;
    finally
        PropertyEditFrm.Free;
    end;
end;
```

This code create an instance of the editor form created at step one, protect it with a try/finally, assign the OnEditDone event with our handler, assign the property value to be edited and call ShowModal. Once the user has finished editing, it either clicks OK or cancel button. If he clicks OK, then the integers are validated (this is done in the editor form) and the OnEditDone event is triggered, which in turn go back to our property editor code to set the value to the component being edited.

The line to get the property value and assign it to the editor form is worth a few words of explanations. This is probably the most complex line in the property editor! The line is:

```
 PropertyEditFrm.IntList := TIntegerList(
                            GetObjectProp(GetComponent(0), GetPropInfo));
```

We have 3 calls in there: GetObjectProp, GetComponent and GetPropInfo.

GetObjectProp utilizes Delphi's RTTI system to return the current value of a component's property where that property is an object type. TIntegerList, is an object type. The first argument passed to GetObjectProp is the object instance and the second argument is the run time information about the property. The value returned by GetObjectProp is cast to TIntegerList which is the data type of the property being edited.

To get the instance of the component, we use GetComponent(0) which is a method of the base class TPropertyEditor made for that purpose. The argument 0 means we want to get the first component select if the user selected several components at once.

To get the run time information about the property, we use GetPropInfo which is another method of the base class TPropertyEditor. It returns a pointer to the record containing useful information about the property.

Another interesting line is within the EditDone event handler. Remember that handler is called when the user has finished editing the integer list. The editor form has his own IntList property updated. We must transfer that value to the property being edited.

```
procedure TIntegerListPropertyEditor.EditDone(Sender: TObject);
var
    PropertyEditFrm: TIntegerListEditForm;
begin
    PropertyEditFrm := Sender as TIntegerListEditForm;
    TIntegerList(GetObjectProp(GetComponent(0), GetPropInfo))
        .Assign(PropertyEditFrm.IntList);
end;
```

The assignation is very similar to the reverse assignation we explained above. The exact same calls are used to get hand on the property object value and to call Assign on behalf of it.

The Assign method has been created in the TIntergerList class and is used to copy the value of another TIntegerList. Here to copy the list of integer from the editor form property to the property being edited.

The remaining code in the property editor is not really worth describing it in details.

GetAttributes just returns [paDialog] to tell the object inspector that the property editor is a dialog (a form). GetValue and SetValue are getting hand on the object instance as explained above and calling ToString or FromString methods to translate to/from string.

Finally, the register procedure has the task to register the property editor when the package containing it is installed. It registers the property editor as general editor for TIntegerList data type, whatever the class is using it.

Source code

Below is the actual source code for all pieces of the article. You can download it from http://www.overbyte.be/frame_index.html?redirTo=/blog_source_code.html

```
==================================================================

unit TickLineControl;

interface

uses
    SysUtils, Classes, Controls, Graphics, Generics.Collections, IntList;

type
    TTickLine = class(TCustomControl)
    private
        FTickPos : TIntegerList;
```

```
        procedure SetTickPos(const Value: TIntegerList);
        procedure TickPosChange(Sender : TObject);
    protected
        procedure Paint; override;
    public
        constructor Create(AOwner : TComponent); override;
        destructor  Destroy; override;
    published
        property TickPos : TIntegerList  read  FTickPos
                                         write SetTickPos;
    end;


implementation

{ TTickLine }

constructor TTickLine.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    FTickPos          := TIntegerList.Create;
    FTickPos.OnChange := TickPosChange;
    FTickPos.BeginUpdate;
    FTickPos.Add(10);
    FTickPos.Add(20);
    FTickPos.Add(40);
    FTickPos.EndUpdate;
    Width  := 200;
    Height := 20;
end;

destructor TTickLine.Destroy;
begin
    inherited Destroy;
    FreeAndNil(FTickPos);
end;

procedure TTickLine.Paint;
var
    I: Integer;
    X : Integer;
begin
    Canvas.Pen.Style := psClear;
    Canvas.Rectangle(0, 0, Width - 1, Height - 1);
    if Assigned(FTickPos) then begin
        Canvas.Pen.Style := psSolid;
        for I := 0 to FTickPos.Count - 1 do begin
            X := FTickPos[I];
            Canvas.MoveTo(X, 0);
            Canvas.LineTo(X, Height - 1);
        end;
    end;
end;

procedure TTickLine.SetTickPos(const Value: TIntegerList);
begin
```

```
    if not Assigned(FTickPos) then
        Exit;
    FTickPos.Assign(Value);
    Invalidate;
end;

procedure TTickLine.TickPosChange(Sender: TObject);
begin
    Invalidate;
end;

end.
```

```
==========================================================================
unit IntListTestAppMain;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, ExtCtrls, Forms, Dialogs, StdCtrls,
    ComponentToString,
    TickLineControl,
    IntList, IntListEditor;

type
    TForm1 = class(TForm)
        EditorPanel: TPanel;
        WriteButton: TButton;
        Memo1: TMemo;
        ReadButton: TButton;
        TickLine1: TTickLine;
        procedure FormCreate(Sender: TObject);
        procedure FormShow(Sender: TObject);
        procedure WriteButtonClick(Sender: TObject);
        procedure ReadButtonClick(Sender: TObject);
    private
        FTickEditForm : TIntegerListEditForm;
        procedure TickEditDone(Sender : TObject);
        procedure TickEditCanceled(Sender: TObject);
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}


procedure TForm1.WriteButtonClick(Sender: TObject);
begin
    Memo1.Clear;
    Memo1.Lines.Add(ComponentToStringProc(TickLine1));
end;

procedure TForm1.ReadButtonClick(Sender: TObject);
begin
    StringToComponentProc(Memo1.Lines.Text, TickLine1);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    FTickEditForm := TIntegerListEditForm.CreateParented(EditorPanel.Handle);
    FTickEditForm.BoundsRect     := Rect(0, 0, EditorPanel.Width,
EditorPanel.Height);
```

```
    FTickEditForm.Visible        := TRUE;
    FTickEditForm.OnEditDone     := TickEditDone;
    FTickEditForm.OnEditCanceled := TickEditCanceled;
end;

procedure TForm1.TickEditDone(Sender: TObject);
begin
    TickLine1.TickPos := FTickEditForm.IntList;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
    FTickEditForm.IntList        := TickLine1.TickPos;
end;

procedure TForm1.TickEditCanceled(Sender: TObject);
begin
    FTickEditForm.IntList := TickLine1.TickPos;
end;

initialization
    RegisterClass(TTickLine);


end.
```

```
======================================================================
unit IntList;

interface

uses
    Types, Classes, SysUtils, StrUtils, Generics.Collections;

type
    TIntegerList = class(TPersistent)
    protected
        FList          : TList<Integer>;
        FUpdating      : Integer;
        FUpdated       : Boolean;
        FOnChange      : TNotifyEvent;
        function  GetItems(nIndex: Integer): Integer; virtual;
        procedure SetItems(nIndex: Integer; const Value: Integer); virtual;
        procedure DefineProperties(Filer: TFiler); override;
        procedure ReadData(Reader: TReader); virtual;
        procedure WriteData(Writer: TWriter); virtual;
        procedure ListNotifyHandler(Sender     : TObject;
                                    const Item : Integer;
                                    Action     : TCollectionNotification);
        procedure TriggerOnChange; virtual;
    public
        constructor Create; virtual;
        destructor  Destroy; override;
        function  Add(const Value : Integer) : Integer; virtual;
        procedure Assign(From : TPersistent); override;
        procedure BeginUpdate; virtual;
        procedure Clear; virtual;
        function  Count : Integer; virtual;
        procedure Delete(Index: Integer); virtual;
        procedure EndUpdate; virtual;
        procedure FromString(const S : String); virtual;
        function  IndexOf(const Value: Integer): Integer; virtual;
        procedure Insert(Index: Integer; const Value: Integer); virtual;
        function  Remove(const Value: Integer): Integer; virtual;
        procedure Sort; virtual;
        function  ToString: String; override;
        property Items[nIndex : Integer] : Integer read  GetItems
                                                   write SetItems; default;
        property OnChange     : TNotifyEvent       read  FOnChange
                                                   write FOnChange;
    end;

implementation

{ TIntegerList }

function TIntegerList.Add(const Value: Integer): Integer;
begin
    Result := FList.Add(Value);
end;
```

```
procedure TIntegerList.Assign(From: TPersistent);
var
    I : Integer;
begin
    if From is TIntegerList then begin
        FList.Clear;
        for I in TIntegerList(From).FList do
            FList.Add(I);
    end
    else
        inherited Assign(From);
end;

procedure TIntegerList.BeginUpdate;
begin
    Inc(FUpdating);
end;

procedure TIntegerList.Clear;
begin
    FList.Clear;
end;

function TIntegerList.Count: Integer;
begin
    Result := FList.Count;
end;

constructor TIntegerList.Create;
begin
    inherited Create;
    FList           := TList<Integer>.Create;
    FList.OnNotify := ListNotifyHandler;
end;

procedure TIntegerList.ReadData(Reader: TReader);
begin
    Reader.ReadListBegin;
    BeginUpdate;
    try
        Clear;
        while not Reader.EndOfList do
            Add(Reader.ReadInteger);
    finally
        EndUpdate;
    end;
    Reader.ReadListEnd;
end;

function TIntegerList.Remove(const Value: Integer): Integer;
begin
    Result := FList.Remove(Value);
end;

procedure TIntegerList.WriteData(Writer: TWriter);
```

```
var
    I : Integer;
begin
    Writer.WriteListBegin;
    for I := 0 to Count - 1 do
        Writer.WriteInteger(FList[I]);
    Writer.WriteListEnd;
end;

procedure TIntegerList.DefineProperties(Filer: TFiler);

  function HasData: Boolean;
  begin
      if Filer.Ancestor <> nil then begin
          Result := TRUE;
          if Filer.Ancestor is TIntegerList then
              Result := not Equals(TIntegerList(Filer.Ancestor))
      end
      else
          Result := FList.Count > 0;
  end;

begin
    Filer.DefineProperty('Integers', ReadData, WriteData, HasData);
end;

procedure TIntegerList.Delete(Index: Integer);
begin
    FList.Delete(Index);
end;

destructor TIntegerList.Destroy;
begin
    FreeAndNil(FList);
    inherited Destroy;
end;

procedure TIntegerList.EndUpdate;
begin
    Dec(FUpdating);
    if FUpdating = 0 then begin
        if FUpdated then
            TriggerOnChange;
        FUpdated := FALSE;
    end;
end;

procedure TIntegerList.FromString(const S: String);
var
    SArray : TStringDynArray;
    N      : String;
begin
    SArray := SplitString(S, ';');
    Clear;
    for N in SArray do
        Add(StrToIntDef(N, 0));
```

```
end;

function TIntegerList.GetItems(nIndex: Integer): Integer;
begin
    Result := FList[nIndex];
end;

function TIntegerList.IndexOf(const Value: Integer): Integer;
begin
    Result := FList.IndexOf(Value);
end;

procedure TIntegerList.Insert(Index: Integer; const Value: Integer);
begin
    FList.Insert(Index, Value);
end;

procedure TIntegerList.ListNotifyHandler(
    Sender     : TObject;
    const Item : Integer;
    Action     : TCollectionNotification);
begin
    if FUpdating <> 0 then
        FUpdated := TRUE
    else
        TriggerOnChange;
end;

procedure TIntegerList.SetItems(nIndex: Integer; const Value: Integer);
begin
    FList[nIndex] := Value;
end;

procedure TIntegerList.Sort;
begin
    FList.Sort;
end;

function TIntegerList.ToString: String;
var
    N       : Integer;
begin
    Result := '';
    for N in FList do
        Result := Result + IntToStr(N) + ';';
    if Length(Result) > 0 then
        SetLength(Result, Length(Result) - 1);
end;

procedure TIntegerList.TriggerOnChange;
begin
    if Assigned(FOnChange) then
        FOnChange(Self);
end;

end.
```

```
unit IntListPropEditor;

interface

uses
    Windows, Types, SysUtils, Classes, Forms, Controls,
    DesignIntf, DesignEditors,
    TypInfo,
    IntList,
    IntListEditor;

type
    TIntegerListPropertyEditor = class(TPropertyEditor)
    protected
        procedure EditDone(Sender : TObject);
    public
        procedure SetValue(const Value: string); override;
        function  GetValue: String; override;
        function  GetAttributes: TPropertyAttributes; override;
        procedure Edit; override;
    end;

procedure Register;


implementation

{ TIntegerListPropertyEditor }

// This is called when the '...' button for the IntValue property is clicked.
// It displays the Property Editor dialog box, and sets the property to the
// new value when the dialog is closed (unless it is cancelled). }
procedure TIntegerListPropertyEditor.Edit;
var
    PropertyEditFrm: TIntegerListEditForm;
begin
    PropertyEditFrm := TIntegerListEditForm.Create(Application);
    try
        PropertyEditFrm.OnEditDone := EditDone;
        PropertyEditFrm.IntList    :=
TIntegerList(GetObjectProp(GetComponent(0), GetPropInfo));
        PropertyEditFrm.ShowModal;
    finally
        PropertyEditFrm.Free;
    end;
end;

// The GetAttributes method returns a value indicating the attributes that
// should be used by the Object Inspector for displaying the property.
procedure TIntegerListPropertyEditor.EditDone(Sender: TObject);
var
    PropertyEditFrm: TIntegerListEditForm;
begin
    PropertyEditFrm := Sender as TIntegerListEditForm;
    // Assign the new value back to the component.
    TIntegerList(GetObjectProp(GetComponent(0), GetPropInfo))
        .Assign(PropertyEditFrm.IntList);
```

```
end;

function TIntegerListPropertyEditor.GetAttributes: TPropertyAttributes;
begin
    Result := [paDialog {, paReadOnly}];
end;

// The GetValue method returns the string that should be displayed by
// the Object Inspector.
function TIntegerListPropertyEditor.GetValue: String;
var
    IntList : TIntegerList;
begin
    IntList := TIntegerList(GetObjectProp(GetComponent(0), GetPropInfo));
    Result  := IntList.ToString;
end;

// The SetValue procedure is passed the new value as a string, and should
// update the actual property with this value.
procedure TIntegerListPropertyEditor.SetValue(const Value: string);
var
    IntList : TIntegerList;
begin
    IntList := TIntegerList(GetObjectProp(GetComponent(0), GetPropInfo));
    IntList.FromString(Value);
end;

procedure Register;
begin
    RegisterPropertyEditor(TypeInfo(TIntegerList), nil, '',
TIntegerListPropertyEditor);
end;

end.
```

```
=========================================================================
unit IntListEditor;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs,
    IntList, TypInfo, StdCtrls;

type
    TIntegerListEditForm = class(TForm)
        ColLineLabel: TLabel;
        OKButton : TButton;
        CancelButton: TButton;
        IntListMemo: TMemo;
        procedure OKButtonClick(Sender: TObject);
        procedure CancelButtonClick(Sender: TObject);
        procedure IntListMemoChange(Sender: TObject);
    private
        FIntList                : TIntegerList;
        FParentedFlag           : Boolean;
        FOnEditDone             : TNotifyEvent;
        FOnEditCanceled         : TNotifyEvent;
        function  GetIntList: TIntegerList;
        procedure SetIntList(const Value: TIntegerList);
    public
        constructor Create(AOwner : TComponent); override;
        constructor CreateParented(ParentWindow: HWnd); virtual;
        destructor  Destroy; override;
        property IntList : TIntegerList read  GetIntList
                                        write SetIntList;
        property OnEditDone             : TNotifyEvent read  FOnEditDone
                                                       write FOnEditDone;
        property OnEditCanceled         : TNotifyEvent read  FOnEditCanceled
                                                       write FOnEditCanceled;
    end;


var
  IntegerListEditForm: TIntegerListEditForm;

implementation

{$R *.dfm}

{ TIntegerListEditForm }

procedure TIntegerListEditForm.CancelButtonClick(Sender: TObject);
begin
    if Assigned(FOnEditCanceled) then
        FOnEditCanceled(Self);
end;
```

```
constructor TIntegerListEditForm.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    if FParentedFlag then
        BorderStyle := bsNone;
    FIntList      := TIntegerList.Create;
    IntListMemo.Clear;
end;

constructor TIntegerListEditForm.CreateParented(ParentWindow: HWnd);
begin
    FParentedFlag := TRUE;
    inherited CreateParented(ParentWindow);
end;

destructor TIntegerListEditForm.Destroy;
begin
    FreeAndNil(FIntList);
    inherited;
end;

function TIntegerListEditForm.GetIntList: TIntegerList;
begin
    Result := FIntList;
end;

procedure TIntegerListEditForm.IntListMemoChange(Sender: TObject);
var
    Line, Col : Integer;
begin
    Line := IntListMemo.Perform(EM_LINEFROMCHAR, IntListMemo.SelStart, 0);
    Col  := IntListMemo.SelStart - IntListMemo.Perform(EM_LINEINDEX, Line,
0);
    ColLineLabel.Caption := IntToStr(Col) + ', ' + IntToStr(Line);
end;

procedure TIntegerListEditForm.OKButtonClick(Sender: TObject);
var
    Line : Integer;
    S    : String;
begin
    // Remove empty lines at the end
    Line := IntListMemo.Lines.Count - 1;
    while (Line >= 0) and (Trim(IntListMemo.Lines[Line]) = '') do begin
        IntListMemo.Lines.Delete(Line);
        Dec(Line);
    end;
    // Check all values. We don't accept empty lines but we accept spaces
    Line := 0;
    try
        while Line < IntListMemo.Lines.Count do begin
            S := IntListMemo.Lines[Line];
            StrToInt(Trim(S));
            Inc(Line);
        end;
```

```
    except
        // An error occured, display error message and quit
        ShowMessage('Invalid integer value "' + S +
                    '" at line ' + IntToStr(Line));
        IntListMemo.SelStart := IntListMemo.Perform(EM_LINEINDEX, Line, 0) ;
        IntListMemo.SelLength := Length(S);
        IntListMemo.SetFocus;
        IntListMemo.Perform(EM_SCROLLCARET, 0, 0);
        Exit;
    end;
    // Then convert values to integer list
    FIntList.BeginUpdate;
    try
        FIntList.Clear;
        for Line := 0 to IntListMemo.Lines.Count - 1 do
            FIntList.Add(StrToInt(Trim(IntListMemo.Lines[Line])));
    finally
        FIntList.EndUpdate;
    end;
    if Assigned(FOnEditDone) then
        FOnEditDone(Self);
    if not FParentedFlag then
        Close;
end;

procedure TIntegerListEditForm.SetIntList(const Value: TIntegerList);
var
    I : Integer;
begin
    FIntList.Assign(Value);
    IntListMemo.Lines.BeginUpdate;
    try
        IntListMemo.Clear;
        for I := 0 to FIntList.Count - 1 do
            IntListMemo.Lines.Add(IntToStr(FIntList[I]));
    finally
        IntListMemo.Lines.EndUpdate;
    end;
end;

end.
```