



## ***Linking chart Series to custom Arrays and speed tips***

TeeChart Pro 4.0 - May 1998.

Copyright 1995-1998 by David Berneda

This technical document explains a new feature of TeeChart Pro 4.0 that allows using existing arrays of values as the default Series storage objects.

This feature should be considered for advanced developers only, as it is necessary some run-time code and good knowledge of the internal architecture of both Delphi and TeeChart objects.

This document also discusses how to obtain the faster speed using TeeChart.

### **Why custom Arrays ?**

#### **Benefits using custom Arrays**

#### **How to use Arrays ?**

#### **Restrictions**

#### **Future**

### ***Why custom Arrays ?***

Many applications make use of Arrays to organize and store data. Delphi Arrays are a very low-level and basic memory storage variable types:

```
Var MyData : Array[ 0..99 ] of Double ;
```

With TeeChart, the normal procedure to plot the above array is this:

```
Series1.Clear;  
for t:=0 to 99 do Series1.Add( MyData[ t ], ", ", clTeeColor );
```

TeeChart does not use internally Arrays to store point values. Instead, chart Series use standard Delphi "TList" objects to store values. ( See Delphi's help file topics for TList object for more information ).

One of the main reasons of not using Arrays is that Arrays do not support dynamic sizing. That is, once you have defined an array of, for example, size 1000, you cannot change it later to be 2000 or 50.

Of course, dynamic allocation can be easily implemented using arrays of pointers and make the Series to allocate and deallocate pointers, but this is exactly what a Delphi's TList object does, so that's why TLists where chosen as the default storage mechanism for Series point values.

On the other hand, Arrays give of course faster execution speed when reading or writing values and also use less memory as there is no overhead of having one pointer variable for every element on the array.

Rumours say the new Delphi 4 version will introduce "dynamic arrays" right inside the Object Pascal language, but TeeChart cannot make a big use of this as it should work also with the other versions of Delphi and C++ Builder, which do not have dynamic array resizing.

### ***Benefits using custom Arrays***

The main two benefits are: Memory Size and Execution Speed.

#### **Memory Size**

Using Arrays instead of TLists, used memory is reduced.  
A technical explanation follows:

Every point on a chart Series uses this amount of bytes:

Bytes	Used for...
4	a pointer inside the "Y" values TList.
8	a "double" variable to hold the "Y" value.
4	a pointer inside the "X" values TList.
8	a "double" variable to hold the "X" value.
4	the point "color" in "long-integer" form.
4	a pointer to the "Label" point text.

If the point has an associated "text label", the length in bytes of the text string is also stored in memory.

This adds up to 32 bytes per point, plus the length of the point text (if point text label is not empty).

So, if we have for example 10000 (ten thousand) points in a Series, the Series is using a minimum of 32\*10000 bytes of memory ( about 312 kilobytes ).

Using Delphi Arrays, the amount of memory is just and exactly the size of the array. So in our example, we can allocate 10000 values using an array of doubles:

```
Var MyData: Array[ 0..9999] of Double;
```

As each "double" occupies 8 bytes, the above array uses 8\*10000 bytes of memory ( about 80 kilobytes ).

This amount can further be reduced using "integers" or "bytes" instead of "doubles", at the expense of losing floating point decimals and reducing the allowed values range:

```
Var MyData: Array[ 0..9999] of Integer; ( 4*10000 bytes = about 40 kilobytes )
```

So, it's clear using Arrays instead of TList's will save quite a lot of memory, but if we need to store text for point labels or point colors, then the advantage is not that relatively big.

The use of "custom Arrays" in chart Series is intended for cases where we want to plot large quantities of values using Fast-Line series, having all points the same color and not having any associated text. ( See below example ). If memory or speed is not a critical problem, then simply calling the Series "Add" or "AddXY" method for each element in the array is enough simply and efficient.

## Execution Speed

When using chart Series, two main phases account for total speed:

### A) Adding points to a Series

Adding points to a Series takes time because the Series does the following steps for each added point:

A1) First, the Series calls the "OnBeforeAdd" event (if used by the developer) to notify that a new point is going to be added to the Series.

A2) The point's "X" and "Y" values are added to the internal "TList" objects. If the "XValues" or "YValues" Series properties have assigned the "Order" property, then the point values are inserted at their corresponding sorted position in the TList object. When the developer does not specify an "X" value for the point, an automatic X position is calculated and added to the "XValues" list. ( A sequential number: 0,1,2,3.. etc ).

A3) The desired point Color is stored in the "ValueColor" property. ( Developers can specify the "clTeeColor" constant as the point color, thus the color is not stored, and the global SeriesColor property is used for all points. ).

A4) The point's label text is stored in the "XLabel" Series property. Specifying an empty text string for the point text, stores a "nil" pointer in the internal Labels array.

A5) If the Series is a "datasource" for any other Series, the Series notifies to all the "linked" Series that a new point has been added. ( A Series is a "datasource" of other Series when the later use "Functions" ). This is necessary for "linked" Series to recalculate their functions.

A6) Finally, the Series notifies Delphi and Windows that the Chart should be repainted by calling the standard "Invalidate" method. When many points are added to a Series (for example in a "for...do" loop), only the last call to "Invalidate" will produce a "real" chart repaint, because Windows "caches" consecutive calls to Invalidate.

Of course, these steps are fully optimized, and, for the worst case, we are talking here of a very small amount of milliseconds for each point, depending on the Windows version, Delphi version, and most important, the kind of CPU used. For example, adding more than 15000 (fifteen thousand) points takes about 1 second on a Pentium 166Mhz machine using Windows 95 and Delphi 3.

Other very important aspects that affect performance are compiling the application with "range checking" or "overflow checking" ( these options are at Delphi "Project->Options->Compiler dialog ). Compiler checkings also need some small amount of time that can be important in the global overall process.

Also, two main TeeChart properties have big importance in optimizing the above steps:

Chart1.AutoRepaint : Boolean default True  
Series1.XValues.Order default "loAscending"

### **AutoRepaint**

"AutoRepaint" is a new public property introduced in TeeChart Pro 4.0 version.

This property controls if a Chart or Series component will call or not to Delphi's and Window's "Invalidate" method every time the chart should be repainted.

When adding new points to a Series, the Series calls the "Invalidate" method. This method tells Windows that the space occupied by the Chart should receive a message to force it to repaint.

In theory, calling "Invalidate" is a very fast operation, as it simply makes Windows to flag the Chart window handle as a "dirty" screen space, so when we finish adding points, Windows can send a "WMPaint" message to the Chart control.

In practice, the call to "Invalidate" takes surprisingly a lot of milliseconds, so it's ridiculous to spend hours of programming trying to optimize all the other parts of the source code when a so simply thing can take the same or more time to execute than the rest.

The solution is to simply not call to Invalidate until we are sure and we are finished adding all points to a Series. When we have finished, we then tell the Chart to repaint:

```
Chart1.AutoRepaint:=False;  
for t:=0 to 9999 to Series1.Add( MyData[ t ] , " , cITeeColor );  
Chart1.AutoRepaint:=True;  
Chart1.Repaint;
```

### **XValues.Order**

This property can also speed up adding points a little bit more. Setting the Order to "none", makes the Series to not try to re-order points based on their "X" coordinate.

Of course, disabling X ordering implies the developer is adding points in the right X order.

The internal code responsible to sort points is very fast for new added points, specially when new points are always the "last" in the X range.

But in order to avoid a small amount of time, setting Order to None speeds up adding points a little bit more.

## **B) Displaying points**

When all points have been added to a Series, now it's time to display them.

There are many factors that affect display performance. These are explained in the existing TeeChart Pro 3.0 "TeeManual.doc" document but they will be rewritten here with more detail:

1) Windows version

Every different Windows version has a different implementation of "GDI.EXE" Windows module, which is the responsible software that "talks" between the operating system and the video card driver software. Inside GDI there are many graphics methods ( line, ellipse, polygon, text output and so on), that will finally communicate with the video card driver to actually turn screen pixels on and off.

During these years since Windows 1.0, Microsoft has been optimizing these graphics methods as they are quite important in the overall total speed of Windows execution (after all, Windows is graphical operating system ).

For example, in Windows NT4, most graphic operations are much faster than the equivalent implementation in Windows 95. This might be due because moving the "GDI" module to a better and faster 32bit assembler to shorten the way for pixels to run until they get to the video driver memory.

So, it's always important to compare performance between each different Windows version and Windows "Service Packs" patches.

## 2) Video card and driver

Every single different video card has a different speed. Even cards of the same brand and model have different versions of hardware, with different speeds. The card manufacturer in most cases also develops the video driver, which is a piece of software that has big importance in how, for example, a polygon is filled into the video card memory.

So, better video cards and better video card drivers can give us a better drawing speed. ( And sometimes new bugs too ).

Test your video card with different resolution and color modes. Some cards might be faster in 256 color mode than 32K or 16 million color modes. Also some cards run faster in 800x600 pixels than 1024x768 or greater. Some features that video cards provide like "hardware zoom" or "power saving" can also reduce performance.

## 3) CPU

If you try to install the best and fastest video card on a 486 machine, you will notice it might go slower than the worst video card on a Pentium 166. Of course, modern Pentium 400 Mhz machines should be faster than other Pentiums with the same video card. Speed improvement is quite linearly proportional to the CPU speed.

## 4) Chart dimensions

TeeChart uses a hidden internal bitmap object to paint everything on it, and then to send this bitmap to the screen in a single Windows API call ( bitblt ).

This basic technique has two benefits:

### 4.1) No flickering between several chart redraws

As the internal bitmap is transferred in a single call to the screen, flicker is almost inexistent.

### 4.2) Reducing the need to redraw

If we move another Window over a Chart, the internal bitmap is reused to repaint the Chart once the new window is outside the chart rectangle. This means it's not necessary to repaint again the Chart every time we move other windows over it.

You can disable this so-called "double-buffering" by setting the Chart.BufferedDisplay property to False. Moving a bitmap to the screen takes an amount of time that is almost exactly proportional to the bitmap size ( the Chart Width and Height properties ).

So, a half-width and half-height chart should display twice faster if it has the same number of points.

The rule-of-thumb is to try to make the Chart size the smaller as possible to obtain the best performance.

## 5) Number of Series and Points

More Series and more Points also takes more time to display. For every point, the Series needs first to calculate its "physical" position in the screen and then execute the appropriate graphic calls to display it.

The faster Series in TeeChart is "Fast-Line". This Series simply calls Windows "MoveTo" and "LineTo" graphic methods, which should be the faster way for any Windows version and video card. ( Of course, a single dot is faster than a line, so a PointSeries with a single-dot style can maybe be faster than a Fast-Line series ).

## 6) 2D or 3D

New introduced in TeeChart Pro 4.0 is 3D rotation, elevation and zoom. While all these features have been highly optimized, they will for sure take more time to calculate than simply drawing in "2D" mode. ( Switching between 2D and 3D is done by the Chart.View3D property )

## 7) Memory

If we are adding many points to a Series, there's a chance we run out of free Windows memory. When that happens, you should know Windows starts "swapping" (moving) memory to disk files to allow more free memory for applications. This feature has big benefits but takes a lot of time to execute as hard disks are hundreds of times slower than memory chips. The solution is always adding more physical memory, which is cheaper and cheaper every day.

The "double-buffer" feature ( see above 4 ), also uses the memory needed to maintain the internal bitmap.

## 8) Axis scales

If the Chart axes scales are "Automatic" (the default), that means axes should calculate the axis Minimum and Maximum values depending on Series points minimum and maximum values.

That's easy to do. The axis Maximum will be the Series point with the highest value and the axis Minimum the lowest point value. But, in order for the Series to calculate which is the high and low values of points, the Series should traverse all points, which can delay some milliseconds if it has many many points.

Now in TeeChart Pro 4, all this process has been optimized in a way that, if axis scales are not Automatic, the Series will never calculate or traverse all points as it's not necessary to do so.

So, setting the axis Maximum and Minimum can avoid calculations and make display faster, ie:

```
Chart1.LeftAxis.SetMinMax( -10000, 10000 );
```

## 9) Other tips

Several other tips can reduce the time it takes for a Chart to display, like:

- Disabling Chart Gradient background filling
- Disabling Chart BackImage picture filling
- Using small Font sizes.
- Using solid pen styles always of Width 1.
- Not using Brush styles. (Always solid brush pattern )
- Hide the Chart Legend, axis Titles and Chart Title and Foot
- Hide Axis grid lines
- Show the minimum number of Series at the same time.
- Avoid using "TeeFunctions"
- Not using circle or ellipse shapes

## How to use Arrays ?

### Speeding things even more. Using arrays.

There might be situations when even after solving the above speed problems we need more speed or less used memory.

This is where the new "custom arrays" TeeChart feature takes place.

In our tests, 250000 ( two hundred fifty thousand points ) takes less than two seconds to add and display, using Delphi 3, Windows NT4 with an old Matrox Milenium video card.... and custom arrays.

One important thing to remember is also that, if you display more points than pixels your screen mode provides, many points will be duplicated ( they will be displayed using the same screen pixel ), so its very

important always to try to display the less number of points as possible. ( If you use a 1024 x 768 video mode, more than 1024 points will start using the same pixels ). You can reduce the number of plotted points by for example discarding points (plot only 1 of every 10 points), or plotting the averages ( plot one point as the average of every ten points ), or dividing a chart in "pages" (showing a portion of the total points at a time).

Another important thing is to use Fast-Line Series. It has few sense to plot a Pie chart with one million slices, unless you use twenty monitors at ultra high resolution !

Let's see the source code to link an array to a Series. Complete code and example of use is provided with TeeChart Pro 4.0 version.

### 1) The Array

You might already have an existing array of values, so the following code is just a "template" on how the array should look:

```
Const NumPoints=25000;

type
  TMyValues=Array[0..NumPoints-1] of Double;

Var MyArrayX : TMyValues;
    MyArrayY : TMyValues;
```

We will use two arrays. One for X values and another for Y values. Using two arrays will guarantee the best results, as if we only use one array for Y values, the Series will use the standard TList for X values, thus we will not get the best speed and memory advantage.

### 2) Filling the Arrays with values

Now let's add some random values to the Arrays. It's supposed you already have your arrays filled with real values, so the following code is only necessary for the "demo":

```
{ first random point... }
MyArrayX[0]:=0;
MyArrayY[0]:=1000;
{ next random points... }
for t:=1 to NumPoints-1 do
begin
  MyArrayX[t]:=t;
  MyArrayY[t]:=MyArrayY[t-1]+Random(50)-24.5;
end;
```

Notice that the "X" array is filled with a sequential X value, starting at zero.

### 3) Creating a test Project

Let's now create a new Project, add a TChart to our Form1, and add a Fast-Line Series to our Chart1. ( You should already know how to do this ! <g> )

### 4) Setting axes scales

The next step would be setting our axis scales to accomodate our desired range of points:

```
Chart1.LeftAxis.SetMinMax(-10000,10000);
Chart1.BottomAxis.SetMinMax(0,NumPoints);
```

## 5) Replacing the Series values with our Arrays

And finally, we need to tell the Series to use our custom Arrays. This is done by first creating a small object and then calling a Series method passing the object:

```
With Series1 do
begin
  ReplaceList(XValues,TMyList.Create(Series1,'X'));
  With TMyList(XValues) do
  begin
    Order:=loAscending;
    MyCount:=NumPoints;
    MyArray:=@MyArrayX;
  end;
  ReplaceList(YValues,TMyList.Create(Series1,'Y'));
  With TMyList(YValues) do
  begin
    MyCount:=NumPoints;
    MyArray:=@MyArrayY;
  end;
end;
```

Note the following in the above code:

We first create and replace the Series "X" values. For "X" values, we optionally set the Order to ascending, although it's not necessary in this example as we provide all X values already sorted. Then we set the "MyCount" and "MyArray" properties to our number of points and our "X" array of values.

We do then the same thing for the Series "Y" values.

That's it. If we now compile and run our project, the Chart will show a fast-line containing 25000 points. No extra memory than our arrays will be used to store the point values, as they are directly accessed using the small "TMyList" objects.

In a similar way, we can replace the Series values lists with other kinds of "TMyList" objects. One of the first new variants of TMyList would be a "TMyTable" object, which will allow charting a, for example, 1 million records table without needing to retrieve all records to fill the Series.

### What is the TMyList object ?

This is the most important thing in this document. This object represents an interface between your arrays (or any other custom data) and the Series.

Here is the interface declaration for "TMyList" object:

Type

```
TMyList=class(TChartValueList)
  protected
  Function GetValue(ValueIndex:Longint):Double; override;
  Procedure SetValue(ValueIndex:Longint; Const AValue:Double); override;
  Procedure ClearValues; override;
  Function AddChartValue(Const Value:TChartValue):Longint; override;
  Procedure InsertChartValue(ValueIndex:Longint; Const Value:TChartValue); override;
  public
  MyArray:^TMyValues;
  MyCount:Integer;
  Constructor Create(AOwner:TChartSeries; Const AName:String); override;
  Function Count:Longint; override;
  Procedure Delete(ValueIndex:Longint); override;
  Procedure Scroll; override;
end;
```

It derives from "TChartValueList" class, which is the one used internally by Series by default.

*Note:* For Delphi 1.0, a conversion of "TMyValues" to "^TMyValues" is necessary, as Delphi 1 16bit does not allow a data-segment of more than 65520 bytes. ( See Delphi 1.0 help about "datasegment" ). Also, the same restriction applies to maximum array size, which can be of no more than 8000 doubles (depending on your application total use of the data segment).

The virtual methods of TMyList object are overridden to access the "MyArray" variable. The most important methods are "GetValue" and "SetValue", which access an individual item in the array. The implementation is quite simple:

```
Function TMyList.GetValue(ValueIndex:Longint):Double;  
begin  
    result:=MyArray^[ValueIndex];  
end;
```

```
Procedure TMyList.SetValue(ValueIndex:Longint; Const AValue:Double);  
begin  
    MyArray^[ValueIndex]:=AValue;  
end;
```

The Series will call these methods whenever it needs to retrieve or store a value. That means we can still use most Series methods to access the YValues and XValues as usually, no matter if the Series storage is one of our new "TMyList" objects or not.

This is an important concept, as, for example, we can still use the Series "Add" or "AddXY" methods together with our "MyList" array objects.

Other methods in the above object simply return the number of points or delete an item from the array. See the complete source code to see how it's done.

The "MyCount" property is very important. The Series needs to know in advance how many points exist in the array as it cannot determine if all points in the array contain valid values. It's also important because it's not possible to know the Low and High array bound limits in a generic way for all possible array sizes and array types.

## Restrictions

As always, a benefit comes attached to some restrictions. Specially in this case, we are using a "back door" of TeeChart Series objects. This "back door" can be very useful but has some limitations.

### -Not Changing Series types

As this feature has only sense for a big number of points, and only for when we need full-speed, it makes big sense to use only Fast-Line series with it. If we have few points or we need to use a different Series type, then the increase of speed that we obtain using arrays is ridiculous compared to the loss of speed of using other Series types which do most drawings than a simple line.

If this is the case, then follow the above tips of disabling AutoRepaint and the Series XValues Order. Do your own profiling tests with your real Charts and data values, using arrays or not, and using a Fast-Line series or not. Try the same tests on different machines to see how hardware affects performance.

### -Needs custom source code

The "TMyList" object cannot be embedd internally inside the Series components. It's practically not possible to make a generic "TMyList" object which will accept all kinds of arrays of all kind of variable types and of all possible array sizes. It's possible however to create a generic object wrapping around Variant types, but this will not work in Delphi 1.0 and 2.0 and will maybe be less efficient than the current default implementation of using TList objects.

Reminder:  
All Series already have since TeeChart 3.0 the following method:



Function AddArray(Const Values:array of Double):Longint;

This method will simply call the Series "Add" method for all elements in the "Values" array parameter:

```
Series1.AddArray([ 1,4,8,2,5,7,9,0,2,11] );
```

## **Future**

As always, more and more ways to make plotting values faster and smoother will be considered and experimented. One very important thing is to maintain compatibility with existing code, existing demos and existing projects. Hundreds of thousands of developers have projects using TeeChart controls.

In Delphi 1.0 times, where an array of doubles is limited to 8000 points (even dynamically created), it made big sense to use the less-restricted TList object. Now in the 32bit arena this limits have been eliminated in a way that is possible to create huge arrays.

So, opening the Chart Series architecture to support arrays has sense for applications like real-time or with high-volume of data points.

For applications compiled with Delphi 1.0 16bit, or running on Windows 3.1x or on machines slower than a Pentium, the techniques described in this document are the fastest ones.