An MFC Chart Control with Enhanced User Interface

geoyar

17 Jun 2013 CPOL

An MFC linear chart control with enhanced appearance.

Download documentation - 3.5 MB

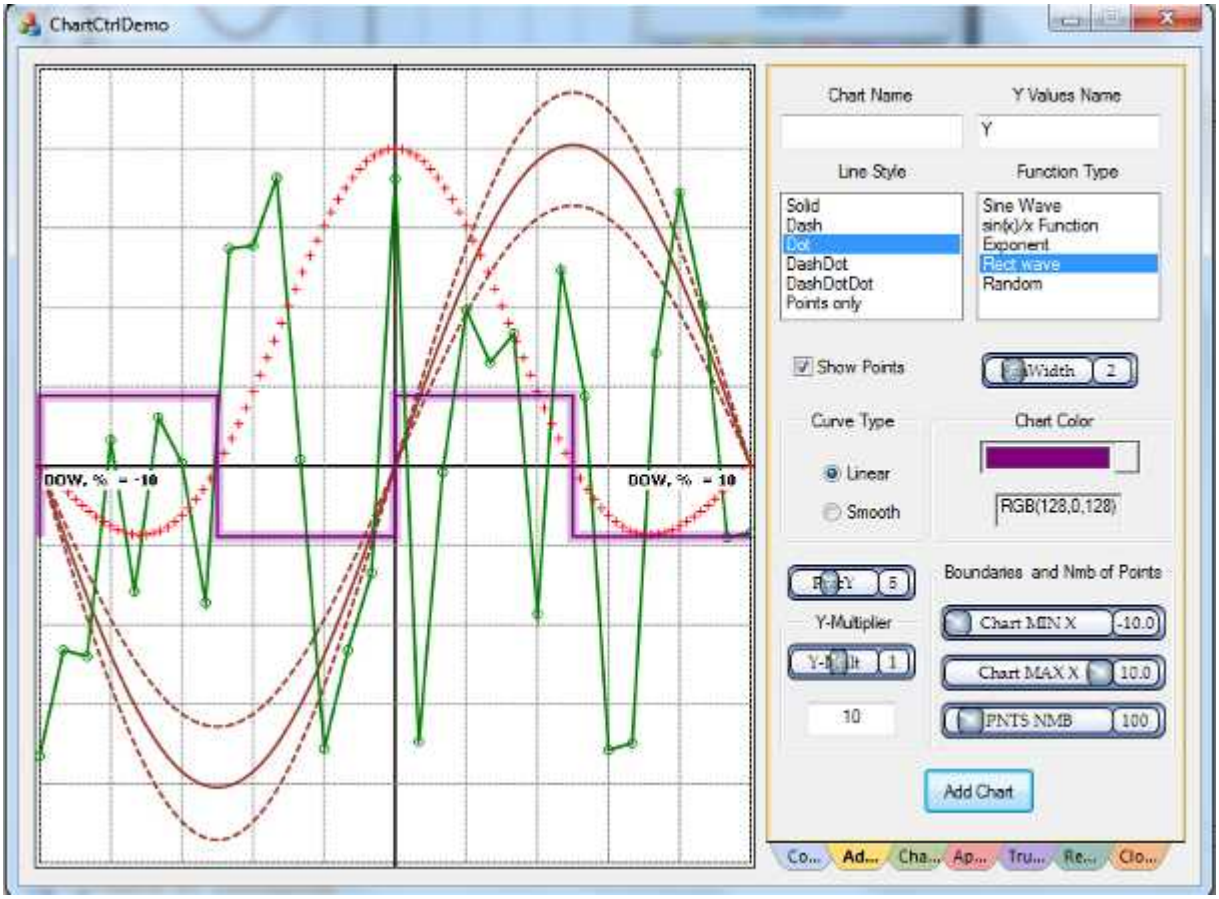
Download source code - 92 KB

Download demo source code - 236 KB

Download demo - 1.8 MB

Download ChartCtrlLibKit - 3.2 MB

Download ChartCtrlLibKitVS2012 (port to VS2012) - 3.2 MB



Demo application: Chart curves

Demo application: Container with labels

Table of contents

Introduction

Why do we want to see a set of data points as a curve? Maybe it is because a human eye is the best instrument for qualitative analysis. We can easily view ups and downs, trends, locate an overshoot here and a minimum there, etc.

Quantities will come later, when we want to know how big the overshoot is, or at what X the minimum occurs. So we need to have a clutter-free picture (without numbers, text labels, etc.) first, and call the data values on the screen later, and hide them to clear the picture again.

After seeing and trying many chart controls, I decided to develop my own. My intent was to get the minimum of clatter on the screen with the maximum features at my fingertips. There is the result.

Acknowledgements

I got some input from online tutorials for printing from dialog-based apps. I want to thank Igor Tandetnik for the help with template predicates for data points and with a `_VARIADIC_MAX` parameter (later for porting to VS2012).

I used some code from Microsoft VS2010 Help examples in the function `SaveContainerImage`.

Changes in version 2.0 are (mostly) results of my discussions with the user/reader Alberl (see posts to this article.)

General description

The screen snapshots above show this chart control in a dialog-based demo application. The first snapshot shows the chart curves, the second one shows the control with two invoked child windows: the data label and the names label. The second snapshot also features a vertical data line and the

chart data points nearest to it. We will call these points the selected points. The data line is going through the user selected X-coordinate, X_0. The data label shows the chart names, names of the X- and Y- axes, and the X and Y values of the corresponding selected points. Also shown are the X-axis labels. Normally, the children, the line and data points, and the X labels are not visible. They could be shown or hidden upon the user's requests. The names on the snapshots are the user's (mine) choice, just to show what could be done with this control.

This control is implemented as an MFC static class library that renders ordered series of 2-D data points as cardinal spline curves. (From Microsoft's web page: "... The spline is specified by an array of points and a tension parameter. A cardinal spline passes smoothly through each point in the array...".)

We will call the curve a chart, and the control a chart container.

You can insert any number of charts, up to 25 charts, in the container. The limit is defined by the number of popup nenu items reserved to hide/show individual charts. The chart visuals: the tension (line smoothness), the line color, dash style, and width of each chart, might be set individually. The chart data might be appended or truncated at any time. To avoid clutter on the screen, you can temporarily hide some of the charts. Since the version 1.1, you can also set the X-axis name, Y-axis names and Y-precisions for each chart. You also can supply your own formatting functions for X- and Y-values.

You can zoom and pan along the charts horizontally and vertically, change the vertical scale of the individual chart, and view the series of chart data points as a table in a separate window. If you select some data points in the table, you will see the exact positions of these points on the chart's curve.

You can print a selected chart, all visible charts, or the chart's data table.

Chart attributes and data series can be saved as an XML file. Later the file can be loaded into a chart container. You can also export the chart's data series as STL vectors.

In the version 1.2 you may save the chart container as image, in any picture format (BMP, JPEG, PNG, etc) your Windows OS supports.

You can control and manipulate the charts with a mouse and keyboard, with the container's popup menu, or programmatically.

All drawing is done in GDI+ and is double buffered to avoid flicker.

There is heavy use of STL containers: vectors, maps, and multimaps. We are extensively using STL algorithms and predicates across the code. All predicates are overwritten for use with 2-D data points.

The code is written in MS VS2010 VC++ 10 and tested under Windows 7 Pro.

The visual layout of the demo app is tuned to the screen resolution 96 DPI logical pixels, isotropic. The mouse is supposed to have three buttons and a wheel, but the option for a two-button mouse is provided.

I choose the type `double` for the internal representation of the chart data because it allows the best combination of range and precision. Besides, scientific data usually are double precision floating-point numbers.

The data series is a vector of points with coordinates `double X`, `double Y`. This vector is a data member of the chart class. The data points define the data space. The container's client rectangle comprises the client, or the screen space. Transforms from one space to another are calculated automatically.

The chart container is, yes, a container, `std::map`. The map keys are chart IDs, the values are pointers to the charts. You are not supposed to deal with this map, and with charts, directly.

The Doxygen-generated project documentation is provided as a zip file "*ChartCtrlLibDoxigen.zip*". Unzip it, open "*Index.htm*" in the folder "*html*" and your default browser will show you the main page. To use the documentation links to the source files, you have to keep the folder structure like the one saved in the source zip file.

What is new in version 1.1?

Readers' feedback and my own experience in using ChartCtrl prompted me to add some new features to the control. The reader Haekkinen asked to remove the compiler options /GL and /LTCG to make it possible to link the library with projects using other compilers. Jeff Archer asked to add the least squares curve fitting to the library. I myself felt a need for some additional representation features. As a result, version 1.1 includes the following additions and changes:

- Compiler options related to optimization, /GL and /LTCG, are removed. Librarian used VS 2010 default /MACHINE:X86.
- A new curve style that draws chart data points as disconnected crosses has been added to the chart dash styles.
- User can set the X-axis name according to his/her choice instead of the default "X".
- User can set the Y-axis name for each chart individually instead of the default "Y".
- Y-precision can be set individually for each chart.
- User can supply formatting functions for X-values and for Y-values of each chart.
- "Set" functions for charts now accept -1 as a chart Idx. This means "All visible charts".
- The chart container now sends notification messages to its parent when the chart's visibility, the extension of the X-axis, or/and "Show/Hide data points" flag is/are changed from the container's context menu.
- Definitions and functions to access the library version information are added to the code.

These additions forced further changes in many container functions to accommodate the new functionality. To demonstrate these new features, I added the new tab "Change Chart", and have added some new controls in the old tabs in the tab control of the demo application.

What is new in version1.2?

Again, the readers' feedback and my own experience in using ChartCtrl prompted me to add some new features to the control.

- The reader Nelisse asked how to save a chart container to some kind of BMP file. In response I added the function `CChartContainer::SaveContainerImage`. The function enumerates the picture formats supported by your version of the Windows OS, and saves the chart container window to the file in the format selected by the user.
- If you have a chart with `Y<sub>max</sub> = 10.0`, and another with `Y<sub>max</sub> = 10<sup>-5</sup>` , you had to manually set the local Y scale for the second chart to see it. I added the function `CChartContainer::EqualizeVertRanges(double spaceMult, bool bRedraw)` to programmatically equalize the charts on the screen. The function automatically sets the local scales Y for the charts that make the visible chart vertical sizes as a progression of the `spaceMult`. For example, if `spaceMult = 0.9`, the visible `Y<sub>max</sub>` for the second chart is `0.9*Y<sub>max </sub>`of the first chart.
- I added the feature and functions `CChartContainer::IsUserEnabled()` and `CChartContainer::EnableUser(bool bEnable, bool bClearState)` to block/allow the user access to the container's popup menu and input to the container from keys and mouse. The disabled container is "read only": you can only view it.
- The signature of `CChartContainer::SaveChartData` was changed to allow to save all charts in the container: now it is `HRESULT CChartContainer::SaveChartData(bool bAll)`. The default value of `bAll` is false.
- The constraint `pntNmb > 2` is removed from the functions `AddChart` and `AppendChartData`. Now you can add a chart without data to a container. It will set all chart attributes passed as parameters to the function `AddChart` or add one or two points to the chart via `AppendChartData`.
- I changed the signatures of the overloadeded functions `AddChart`, `AppendChartData`, and `ReplaceChartData` for time series. Now the user can define the time origin and time steps for the time series. For example, now we have `CChartContainer::AppendChartData(int chartIdx, std::vector<double>& vTmSeries, double startX, double stepX, bool bUpdate)`.
- The functions `CChartContainer::SetChartVisibility` and `CChartContainer::GetChart` now accept the parameter `chartIdx = -1`. For visibility it means "all charts", for `GetChart` it returns the first chart in the container.
- I added notification with code `CODE_REFRESH` that is sent when the user selects the "Refresh" menu item of the container's popup menu.
- The port of *ChartCtrlLib.lib* to Visual Studio 2012 (VC++ 11) is added.

What is new in version 2.0?

As I have mentioned above, the major changes and additions in version 2.0 are the results of requests and suggestions of the user/reader Alberl. We have discussed them at length in posts to this article.

Some features I agreed to are added to the control:

- In version 1.2 the user can add charts without data to a chart container, but the charts were invisible until they acquired at least three data points each. If you are getting one data point an hour, would you wait three hours to see the chart? In version 2.0 the charts without data or with one or two data points are full citizens of the chart container: if there is any.number of data points, you can zoom, pan, see names and data legends, etc. You even can save the chart without data into XML file and load it back.
- I added zooming and panning along Y-axis to a chart container. In version 2.0 zooming and panning along X and Y axes are two separate processes, and the user can control them independently.
- I did some fine tuning of the existing features, like added better synchronization between the container's window and the data view.
- From the posts to this article I learned that there is need for more programmatic control of the ChartCtrl. So I added and refined functions to control the container from its parent.
- And, of course, I have fixed the bugs I have found.

The changes are massive, so it is version 2.0.

Features

Here are four ways to manipulate and control the chart container and charts:

- Mouse
- Keyboard
- Container's popup menu
- Programmatically

The features of the container and charts you can use and manipulate are:

- The length of the data series is limited by common sense only: there is no point to plot one billion data points onto 300 pixels, but the `std::vector` in theory might handle this number.
- The number of charts in the chart container, again, is limited by common sense only. The clutter of too many curves on the screen might be partially alleviated by temporarily hiding some or most of the charts. In my experience, ten or twelve charts per container are enough to make your life miserable: you will have to hide most of them to make sense of your data. For now 25 popup menu items are reserved for representing the individual charts.
- The chart container accepts charts with mufti-valued data series. The mufti-valued data series could have many points with the same coordinate X.
- The chart container accepts charts without data. Obviously, these charts can not be visible, but the container's popup menu shows them as the charts without data.
- The chart data vector can be programmatically appended, truncated, or replaced at any time. The chart might be removed from the container altogether or its data vector might be cleared of all data points.

- Colors of the container's background, border, axes, and all other colors could be programmatically changed at any time.
- Chart's color, line style, pen width, and tension could be changed programmatically at any time.
- The name of the container and that of the container's X-axis could be set or changed programmatically at any time.
- The name of the chart's Y-axis could be set when the chart is inserted into the container, and/or changed programmatically at any time. Each chart could have the individually set Y-axis name.
- The user could supply a formatting function for the container's X-axis values, and formatting functions for the Y-axis values for each chart. The default formatting functions simply convert the numbers to the string.

- Chart's vertical scale could be changed with the mouse wheel, keyboard arrow keys, or programmatically. There is a possibility of programmaticaly equalize the set of local vertical scales. If the user uses a mouse of keypad, the container sends `WM_NOTIFY<code>` to itd parent

- The container automatically calculates axis positions. The user might display or hide the leftmost and rightmost values of the axis X extent.
- The container allows zooming and panning of charts along X and Y axes with the mouse buttons and the mouse wheel, keyboard arrow keys, from the container popup menu, and programmatically. The container keeps history of these actions separately for X and Y axes and allows undoing these actions. If these operations are started from the popup menu, the container sends a notification to its parent (`WM_NOTIFY`).
- The container allows change the visibility and the "Show/Hide data points" flag of every chart programmatically or from the container popup menu. If these operations are started from the popup menu, the container sends a notification to its parent (`WM_NOTIFY`).
- The precision of the presentation of the container's X- and the chart's Y-values could be programmatically changed at any time. The Y-precision is set individually for each chart.
- The user could see the list of names of the visible charts (the names legend) and the list of the X and Y coordinates for all data points closest to the selected value of the X coordinate (the data legend). The legends are child windows of the container and could be hidden if needed.
- The data series of the selected chart could be shown as a table in a separate data window. The window is synchronized with the chart in the container: points selected in the data window will be shown on the chart curve, and the changes in the chart's data vector, names of the chart, X- and Y-axis, formatting functions, and X- and Y-precision are reflected in the data window.
- The container's visible charts, or selected chart, or all charts, visible and not visible, can be saved in an XML file. The file format is proprietary, but the file could be accepted by MS Excel (it is not a genuine Excel file, but it allows manual editing to the perfect Excel format). The charts from this XML file might be loaded in any chart container again.
- The container image can be saved to the picture file in any format supported by OS. The supportet formats (like BMP, JPEG, PNG, and others) are automatically enumerated by the container and presented to the user to choose from.
- All visible charts or the selected chart from the container could be printed as an 8.5" x 11" page.
- The data view table pages could be printed too.
- The chart data vectors could be exported programmatically in three different formats.

- The user's access to the charts might be blocked programmatically. With a blocked (disabled) user the container is "read-only": no chart manipulation is allowed.
- The information about the version of the chart control library can be accessed programmatically.

How to use it

To use the chart control in your application, you should start with some preliminary steps:

- You have to add to your application some code to initialize GDI+ at the start and free it at exit.
- You have to include the static library *ChartCtrlLib.lib* and the header files *ChartDef.h* and *ChartContainer.h* in your project. The alternative is to include in the project all control source files. To make your life easy, use the zip file *ChartCtrlLibKit.zip* to insert a chart container in the application.
- Prepare the data series and add them to the container.

Prepare your application: Enable GDI+

To enable GDI+:

1. Include a private data member `ULONG_PTR m_nGdiplusToken` in your application:

   C++

   ```
   class CMyApp : public CWinAppEx
   {
       ......................................
   private:
       ULONG_PTR m_nGdiplusToken;
       ......................................
   }
   ```

2. In the function `CMyApp::InitInstance`, add two lines:

   C++

   ```
   BOOL CMyApp::InitInstance ()
   {
       ......................................

       Gdiplus::GdiplusStartupInput gdiplusStartupInput;
       Gdiplus::GdiplusStartup(&m_nGdiplusToken,
                           &gdiplusStartupInput, NULL);

       ......................................
   }
   ```

3. To exit your application gracefully, add the function `ExitInstance()` to `CMyApp` if it is not there already, and insert in its body the line:

   C++

```
int  CMyAppApp::ExitInstance()
{
     ..........................................
     Gdiplus::GdiplusShutdown(m_nGdiplusToken);
     ..........................................
}
```

Prepare your application: Enable the static library

There are two library files included to this article: the debug version *ChartCtrlLibD.lib* and the release version *ChartCtrlLib.lib*.

To add the library to your project enter the full path to the library in *"Linker\Input\Additional Dependencies*" in the project properties dialog box.

Yet another way to add the reference to the lib to the project is the use of VS macros. Copy the library *ChartCtrlLibD.lib* into your Solution (Project) Directory *Debug*, and *ChartCtrlLib.lib* into Solution (Project) Directory*Release*. In the appropriate configuration (e.g., Debug or Release) Project Properties select *Configuration Properties/VC++ Directories*. Enter *$(SolutionDir)$(Configuration)* into *Library Directories*. If there is entries after this directory, add a semicolon. In *"Linker\Input\Additional Dependencies.*" enter the name of the library, *ChartCtrlLibD.lib or ChartCtrlLib.lib*. Again, add the semicolon if needed. It will automatically select an appropriate lib vesion when you switch between configurations.

You have to include two files, *ChartDef.h* and *ChartContainer.h*, in your project or add the path to these files in the "VC++ Directories\Include" directory in the project property pages.

If you are going to debug your project and the library together, include the path to the library source files in the "Source Directories" of this page.

Be aware that the paths entered in the VC++ Directories page in the can be inherited by your later projects. If you do not want it to happen, enter the include path into "*C++\General\Additional Include Directories*" instead of the "VC++ Directories" page and enter the path to the static library file *ChartCtrlLib.lib* into "Linker\General\Additional Library Directories".

Of course, you can include in your project all library headers and source files instead of the compiled library file.

If you are using this library together with other libraries, e.g. Boost or Windows SDK, you could get linker and compiler errors "Multiple Definitions." If the multiple definition are in the Windows SDK and/or MFC files, try to ignore them: add linker option /FORCE:MULTIPLE to the command line. It will transform errors to warnings. For other ways to mitigate these errors search MS forums.

If your project is being developed under *MS Visual Studio 2012 VC++ (VC++ 10)*, you have to use the libraries developed under VC++ 10. I have added them in *ChartCtrlLibVS2012.zip* archive.

In addition, in VS 2012 Microsoft have implemented some templates, including *tuples,* using *faux variadics*. They have set the default max number of variadic template parameters to five instead of 10 in VS 2010. As a result, to be able to use *ChartCtrlLib.lib* in VS 2012 C++ project, you have to manually set the `_VARIADIC_MAX` parameter to 10. Igor Tandetnik wrote to me that the most convenient way to do it is to use the Project Properties. So, go to your Project Properties/C++/Preprocessor /Preprocessor Directives and enter `_VARIADIC_MAX=10` into line of the preprocessor directives.

Prepare your application: Set the chart container

If your application is dialog-based, in the resource editor select the picture control in the toolbox and drag it to the place of the chart container in the dialog box. Adjust the control's size and position. In the control properties window, enter a control ID (something like `IDC_STCHARTCONTAINER`). Make sure that the `NOTIFY` control property is set to `TRUE`.

Add the data member to your `CDialog` class definition, like:

```
CChartContainer m_chartContainer;
```

Add the function `DDX_Control` to the dialog function `DoDataExchange(CDataExchange* pDX)` to subclass the control, like:

```
DDX_Control(pDX, IDC_STCHARTCONTAINER, m_chartContainer);
```

On the other hand, if your application is a document-view application, declare the `CChartContainer` data member in a view class, like:

```
CChartContainer m_chartContainer;
```

and call

C++

```
m_chartContainer.CreateChartCtrlWnd(DWORD dwExStyle, DWORD dwStyle,
                const CRect& wndRect, CWnd* pParent, UINT nID);
```

to create a container window. The `dwStyle` parameter will be combined with the `WS_CHILD` and `WS_VISIBLE` styles inside the function.

Set the chart container attributes

The chart container attributes are the container name, X-axis name, formatting function, the X range, precision, and the colors of the container's elements. Actually, you can live happily with the default values: the X-axis name is "X", the Y-axis is "Y", and X- and Y-precision is three. The default formatting functions just convert the numbers to strings with given precision. The container name is used only to print charts. The X range might be set automatically to show all charts in their full extent. If you have some special needs, you can set the X-range with the function:

```
CChartContainer::UpdateExtX(double minExtX, double maxExtX, bool  bRedraw = false)
```

To set the container name, you can supply the name in the constructor like:

```
CChartContainer myContainer(string_t(_T("Demo"))
```

or call the function:

```
SetContainerName(string_t name);
```

Here, `string_t` is `typedef` for `std::basic_string<TCHAR>`.

To set the X-axis name, call the function `CChartContainer::SetAxisXName(string_t nameX, bool bRedraw = false)`.

To set the X-value formatting function, you have to include the code for this function in your application, and register it with the call to the container function `CChartContainer::SetLabXValStrFn(val_label_str_fn pLabValStrFn, bool bRedraw = false)`. `pLabValStrFn` is the pointer to your formatting function. More info about the formatting functions I will provide later.

The value of precision is the number of significant digits to show when the numbers are being converted to strings. The container precision is the precision of X values. The default value of three means that there would be three significant digits shown in any number: 1233.4567 is presented as 1230 and 0.1234567e-45 is 1.23e-045.

Use `SetContainerPrecision(int precision, bool bRedraw = false)` to set precision. The parameter `bRedraw` is a flag to request redrawing of the chart container and its children (data label and chart names legend). The precision influences only the representation of data. It does not change the precision of the chart data series.

The default constructor sets the default colors (white background, black axes and border, gray dot grid, light yellow background for data and name labels, etc., as is shown on the demo snapshot

above). If you want to name the container at the beginning, pass the name to the container's constructor.

The member functions of the chart container to set colors are in the file *ChartContainer.h*.

Be careful with colors because they are interconnected: e.g., changing the background might force you to change the colors of all other container elements and charts. Try to use the demo to see the results of your changes.

All chart container attributes could be changed at any time, not only at initialization.

Prepare your data

Each chart keeps its data series as a vector of data points. The data point is an instantiation of the class template:

```
template <typename T> class PointT
```

for type `double`.

There are `typedef`s:

```
typedef PointT<double> PointD;
typedef std::vector<PointD> V_CHARTDATAD;
```

If you decide to use `V_CHARTDATAD` to prepare your data, remember that your app must see the definition of `PointT` to instantiate it for `double`. The definition of `PointT`, `typedef`s of `PointD` and `V_CHARTDATAD` are in the file *ChartDef.h*.

You might not use `PointD` to prepare your data altogether. The chart container also accepts data series in the form of `std::vector<std::pair<double, double> >`, `std::vector<double>` (time series), and couple of vectors, `std::vector<double>`, one for X-coordinates and one for Y-coordinates. The vector of `PointD` and the vector of pairs will be automatically sorted by X; the time series does not need to be sorted because the X-coordinates will be assigned automatically. By default, time series origin is set to 0.0, and time step is 1.0. It means that the first data point in the time series will be `PointD(0.0, Y<sub>0</sub> )`, the second point will be `PointD(1.0, Y<sub>1</sub> )`, etc. If you want or you need (as to append a time series chart, you have to pass your time origin and/or time step value to appropriate function (`AddChart`, or `AppendChartData`, or `ReplaceChartData`.)

The user is responsible for the last two vectors. They must have the same size; the vector for `X` must be sorted; the `Y` values should be arranged in order corresponding to the sorted Xs.

The container displays the data series with at least one data point (Obviously, you cannot draw the chart with zero data points). Actually, you need to have more: the `Gdiplus::DrawCurve` routine draws an ugly curve if it does not receive enough data points. On this occasion, interpolate yourself. Usually 30 - 40 points for 350 pixels are enough. You also can play with the tension to beautify your curve. For the chart without data `AddChart` will set all chart attributes it can to the function parameters. It will not update container's `X`- and `Y`-extensions of the container.

If all charts in the container have only one data point with the same X coordinates each, the container will automatically set X extension equal to 1% of the X value. If all charts have have only one data point each withe the same Y coordinates, the Y extension is set to 5% i=of the Y value. The container will adjust its X and Y extensions according to acquired chart data. You also might set the extensions beforehand with functions `UpdateExtX` and `UpdateExtY`.

The different charts might have different X- and Y-extents and different number of data points but usually you want to work with related sets of data series in one container. Usually it means the same number of data points and the same X-extension.

Add charts to the chart container

Now it is time to add your charts to the container.

Use the member function of `CChartContainer`:

C++

```
int AddChart(bool bVisible,
             bool bShowPnts,
             string_t label,
             string_t labelY,
             int precisionY,
             Gdiplus::DashStyle dashStyle,
             float penWidth,
             float tension,
             Gdiplus::Color colChart,
             V_CHARTDATAD& vData,
             bool bRedraw = false);
```

Most of the function's parameters are self-explanatory.

If the minimal `X`-distance between two neighboring data points is big enough, the data points will be encircled by small circles. Sometimes it is not desirable to clutter the picture with these circles. Set `bShowPnts = false` to hide them. If the chart has only one data point, it will always render as a circle.

By design, each chart must have a unique name and ID. `AddChart` calculates the IDs automatically. The IDs are unique for each session, which is the time from the entry of the first chart into an empty container to the time when the last chart is deleted from this container.

If the name you entered for the chart is not unique for this session, `AddChart` will add a suffix to the name. The suffix is a chart's ID. E.g., if you enter the name "Sine Wave" for the chart with ID = 8, and the container already has a chart with this name, `AddChart` will add a suffix to it: "Sine Wave_8". If you supplied empty strings for the names, `AddChart` will generate names with the same suffixes like "Chart_0", "Chart_8", etc.

The length of the chart names, and names of the X- and Y- values is limited to 18 characters. If the string length is greater than 18 characters, the string supplied as a parameter to `AddChart` (and to all other functions dealing with the names) will be truncated to this length. The end of the truncated string will comprise of the delimiter, "^", and the last character of the string. For example, the string "Very, very, strange and long, long, long string" will be truncated to "Very, very, stran^g".

The choice of the tension depends on type of the curve and number of data points. Obviously, the random data are better with a linear curve (tension = 0), but ten points of a sine wave might look good with tension = 0.6, and ugly with tension = 1.0.

In addition, there are three overloads for `AddChart` that accept the vector of pairs of doubles, `std::vector<std::pair<double, double> >&`, the time series `std::vector<double>&`, and two vectors `std::vector<double>& X` and `Y`. Use any of them. To add the time series to the container you have to supply the start value of the X coordinate and the value of the step to increase the X value for next points.

The chart color, name, name of the Y-values, dash style, pen width, visibility, and "Show/Hide points" attributes you have supplied might be programmatically changed at any time.

The newly added chart has the default Y-formatting function, `string_t __stdcall GetLabelValStr(double val, int precision, bool bAddEqSign)`. It formats the value string as a number with the given precision.

The function `AddChart` returns the ID of the new chart on success, or -1 on failure.

Remember that charts are allocated on heap; the container stores only pointers to the charts. All container functions take pain to delete the charts if it is appropriate, but it is your responsibility to do that if you are trying to get rid of charts outside the container.

You allowed to supply empty data vector to `Add Charts`. If there is no data points the container will set all chart attributes supplied as `AddChart` parameters, but will not modify the container X- and Y-extensions and will not display the chart. This chart will be listed in the container popup menu as a chart without data.

If you want, supply the formatting function

It is not always convenient to show chart data points as naked numbers. Suppose you have a chart displaying the average monthly temperature versus months. It is naturally to display the X-coordinates as month names and Y-coordinates as the "^{0}F". This is a case to invoke the user supplied formatting functions.

The formatting function is defined in *ChartDef.h* as:

C++

```
typedef string_t (__stdcall *val_label_str_fn)(double val, int precision, bool bAddEqSign);
```

The function takes a number's value and precision and returns a string. The parameter `bAddEqSign`, when it is set to true, adds the prefix "=" to the string. The function is a callback that is called when the container is preparing to display values. The application could set the formatting functions calling the container's "Set" functions:

C++

```
void CChartContainer::SetLabXValStrFn(val_label_str_fn pLabValStrFn, bool bRedraw = false);
bool CChartContainer::SetLabYValStrFn(int chartIdx,
     val_label_str_fn m_pLabYValStrFn, bool bRedraw = false);
```

If the container is not able to find the chart with the given `chartIdx`, it will do nothing and return false.

The default formatting functions are same for the X and Y values:

C++

```
string_t __stdcall GetLabelValStr(double val, int precision, bool bAddEqSign)
{
  sstream_t stream_t;
  stream_t << std::setprecision(precision) << val;
  return bAddEqSign ? string_t(_T("= ")) + stream_t.str() : stream_t.str();
}
```

If you want to display month names along the X-axis, you should write something like:

C++

```
string_t __stdcall GetLabelValStrMonths(double val, int , bool)
{
  if (in_range(-0.5, 0.5, val))
   return string_t(_T("January"));
  else if (in_range(0.5, 1.5, val))
   return string_t(_T("February"));
//etc.
}
```

For Y values it might be:

C++

```
string_t __stdcall GetLabelValStrGradF(double val, int precision, bool bAddEqSign)
{
  sstream_t stream_t;
```

```
  stream_t << std::setprecision(precision) << val << _T(" <sup>0</sup>F");
  return bAddEqSign ? string_t(_T("= ")) + stream_t.str() : stream_t.str();
}
```

You have to register these functions with

C++

```
myContainer.SetLabXValStrFn(GetLabelValStrMonth);
myContainer.SetLabYValStrFn(myChartIdx, GetLabelValStrGradF);
```

User's manual

So for now you have entered all your charts into your container and have the screen as shown in the first demo snapshot above (without child windows). Let us play with it.

Some operations on the container can be performed only programmatically, from outside of it. They are add, append, truncate, and delete charts to/from the container, and access function to the chart and container attributes like names, precision, etc. For other operations, the container has a built-in popup menu and the handlers for mouse clicks, mouse wheel, and keyboard arrow keys. These operations could also be performed with the interface member functions of the container.

There is a synopsis of the built-in operations:

- **To show/hide the names label**, right click on the chart control and select "Show legend" from the popup menu. The legend window will be shown at the upper right corner of the control window with the names of the visible charts. The color of the name characters is the chart color, and a short line before the name has the color, width, and dash style of the chart curve. The content of the label window will be updated when a new chart is added or deleted to/from the container, or when the visibility of the container charts is changed.
- **To view/hide the X-axis labels**, go to the popup menu and select the menu item "Show the axis X boundaries". While on screen, the X-labels will follow the X-axis extent: panning and zooming will change the values they show.
- **To view the chart data point values at the selected X-coordinate**, enable the tracking mode first. Click the mouse middle button. The cursor will be changed to a cross-hair cursor. Now move the cursor to the desired X-position and click the left button. You will see a vertical line at the selected X position and circles around the data points nearest to the line. The nearest data point X-coordinate is the closest to the line in the neighborhood X ± 3 pixels. It might happen that some or all charts will have no such points in this neighborhood, and nothing will be shown. If it happens, zoom and pan the container until the data points are marked by circles around them, select the X, click to see the data window, and undo zoom/panning using the popup menu, if you want to see a full X extent. The data window will follow the selected X-coordinate if the container window is zoomed or panned, hiding when the selected X-coordinate goes out of view, and appearing again. Again, only data for the visible charts are shown. To disable the tracking mode, click the mouse middle button again. The cursor will return to the arrow shape. The data label is shown on the second demo snapshot.

- **For some operations on the charts, you might need to select the chart first**. Move the cursor to the chart you need and left-click close to the curve when the CTRL key is pressed down. The selected chart will be marked by a diffused and wider line. If the cursor is far from any data point and there is no selected chart, the container will select the visible chart with the lesser ID. The second CTRL + left click deselects the selected chart and selects the next chart if the cursor is close to one of the data points of the second chart.
- **To change the vertical scale of the particular chart**, select the chart first as described above. After that, use CTRL + mouse wheel or UP, DOWN, PAGE UP, or PAGE DOWN keys to change the Y-scale of the chart. If you deselect the chart later, it will keep its new Y-scale. This operation does not change the chart's data vector.
- **To view the chart's data table in the date view, select the chart first**. After that go to the popup menu and click the menu item "View Chart Data". The data view window for the selected chart will be shown. If you click on the "View Chart Data" without selecting the chart, the visible chart with the lesser Idx will be selected and its data will be displayed. You can move over pages of the view with the view's arrow buttons. If the chart attributes are changing, you will stay on the same page of the data view if it is possible; if not, you will go to the first page. You can print some or all pages of the view using the view "Print" button. The MFC "Print" dialog will be displayed. Select the printer, set the printer properties, and print the view. If there is no selected chart, the container will show the data for the visible chart with the lesser ID. You can select one or more data points on the data view with a left click on the table cell. The selected point will be shown on the appropriate curve in the container. The second left click on the selected cell will deselect it. Right click on the data view window will deselect all selected table cells. A snapshot of the data view is shown below.

- **You can save any group of charts in an XML file**. Any chart or group of charts from the file can be loaded back in this container or in any other container. The charts might be added to the container or replace the old container's charts. To save only one chart, select it first. To save a group of charts, hide all the charts you are not going to save (use the popup menu to hide the charts). If one of the visible charts is selected, deselect it. After that, go to the popup menu and select "Save Chart(s)" from the "Save/Print Charts" submenu. The MFC "Save As" dialog will be displayed. Enter a file name or select the file to overwrite, and click the dialog's "Save" button. The file format is proprietary, but you can load the file into MS Excel. (The dialog has a default directory to save the file set to the $(SolutionDir)Charts. If you provide this directory in your project, it will be opened on the start of the dialog.) If you save charts programmatically, you can save all charts, visible and not visible, by passing appropriate value of the bool parameter `bAll to the function SaveChartData(string_t pathName, bool bAll)` .

- **You can save the container window as an image.** Hide all charts you do not want to be in the picture. Invoke/hide name and/or data label windows. When you are ready, go to the popup menu and select "Save/Print Charts"/"Save Charts As Image" menu item. The container will enumerate all picture formats your Windows version supported, and present the standard MFC "Save As" dialog box with these file extensions. Click OK, and you are done. (Again, the default directory is set to `$(SolutionDir)Images`.)

- **To print** one of the container's visible charts, select it first. To print a group of charts, hide all other charts. After that, go to the popup menu and click the "Print Charts" item from the "Save/Print Charts" submenu. Select the printer from the popup MFC "Print" dialog, and set the printer properties. Click the OK button and get the printout. The container does not print the label windows together with the chart curves. Instead, it shows the legend strings under the container's image. To facilitate the measurements, the X-axis labels are always visible on the printout. Each legend string includes the chart vertical scale value. If the data label was visible on the screen, the legend string shows the coordinates of the chart data point(s) closest to the selection line. If there are no selected points, the legend strings will show min and max Y values of the chart. If there is no selected chart, the container will print all visible charts. A sample of the printout is shown below.

- **There are two ways to zoom in the container**: using the mouse and from the popup menu. To zoom in horizontally using the mouse, left click on the container with the SHIFT key is depressed. A vertical line will show you the first of the X-boundaries of a zoom extent. The second left click with the depressed SHIFT key will show you the other boundary of the zoom extent. On `LBUTTONUP`, the X-axis of the container will be changed to the new X-extent, and the container window will be updated. The data label position will be adjusted according to the new position of the selected X-coordinate. The data label is hidden if the selected X went out of view. To

horizontally zoom from the popup menu, select the "Zoom In X" menu item from the "Zoom/Move" submenu. The new X-axis extent will be 80% of the old extent. The container saves the values of the previous boundaries of the X-axis to allow undoing the zooming.

- **To zoom in vertically** you also can use the mouse or the context menu. With the mouse you start with the depressed SHIFT key and double left click. A vertical line will be replaced by a horizontal line along a first vertical zoom boundary. The second left click (just left click, not the double left click) will display the second vertical boundary. Release the mouse left button, and the new, zoomed in vertical extent will be displayed. To use the popup menu, use the "Zoom In Y" menu item from the "Zoom/Move" submenu. Again, there will be 80% of the old Y-extent. The data label content will be synchronized with the picture in the container's window. There is no zooming on the container's window if there is no visible data points: the X and Y-extensions will not change. The container saves the values of the previous boundaries of the Y-axis to allow undoing the zooming.
- **There are three ways to pan the container along the X_axis**: using the mouse, or arrow keys, or from the popup menu. To pan the container along the X-axis with the mouse wheel, press the SHIFT key and rotate the wheel. To pan along X-axis with arrow keys, you have to press the LEFT or RIGHT arrow key. To pan from the popup menu, go to the "Zoom/Move" submenu and select the "Move Right" or "Move Left" menu item. The image in the container window will be moved 10% of the X-extent to the left or to the right. The data label will be moved accordingly. The container saves the values of the previous boundaries of the X-axis to allow undoing the panning.
- **There are two ways to pan along the Y-axis:** using arrow keys or from the context menu. The UP, DOWN, PgUP, and PgDn arrow keys have a double function: they control the local Y-scale of the selected chart or they pan the container up or down if there is no selected chart. So make sure than no chart is selected, and use the UP or DOWN key to move the charts 1% of the initial Y-extent, or use PgUp or PgDn to move the charts 10% of the initial Y extent. The popup menu items "Move Up" and "Move Down" of the "Zoom/Move" submenu are moving the charts 10% of the initial Y-extent up or down. Again, the data legend content is synchronizing with the content of the container's window, and old values of the Y-extent boundaries are saved to allow a separate undoing of the zoom/move actions along the Y-axis.
- **To undo the last zooming or panning step**, go to the popup menu and select the "Undo Last Zoom/Move X" or "Undo Last Zoom/Move Y" menu item. The popup menu shows these items only if there is a history of the previous zoom/pan actions. Again, if in tracking mode, the data label will be shown in an appropriate position, even if it was hidden before.
- **To undo all zoom/move steps**, go to the popup menu and select the "Refresh Charts" menu item. Again, if there is no history of zoom/move events, the item is not visible.
- **If you changed your mind after the first click for the horizontal or vertical zoom,** depress the "Delete" key. The first boundary line will be deleted, and the container state before the click will be restored.
- **To hide/show circles around a chart's data points**, select the chart first. After that, go to the popup menu and click the "Show/Hide Chart Pnts" menu item. The circles around the data points are visible only if the minimal X-distance between any two adjacent points is greater than six pixels. If there is no selected chart, the visible chart with the lesser ID is selected and the

circles around its data points are hidden or shown. The checkmark to the left of the "Show/Hide Chart Pnts" menu item indicates the state of this property for the selected chart. If it is checked, the points should be visible when the minimal distance between adjacent data points is greater than six pixels. You can set this property at any time, even if the data points are too close to one another and not visible at this time. You also can to show/hide the circles programmatically calling the function `CChartContainer::AllowChartPnts(int chartIdx, bool bAllowed, bool bVisibleOnly, bool bRedraw)`. If `chartIdx == -1`, the function will operate on all chart or all visible charts. This action has no effect on the charts with only one data point: it is always rendered as a circle.
- **To hide/show the particular chart**, click on the "Show (chart name)" item of the popup menu. When the item is checked, the chart is visible.

Here is a summary of the built-in controls:

Mouse events

- `MBUTTONDOWN`: turns on/off the tracking mode. The tracking mode allows display of values of the visible charts' data points nearest to the selected X-coordinate. In the tracking mode, the cursor over the chart container changes from arrow to a cross-hair shape.
- `LBUTTONDOWN`: In tracking mode, invokes the data label window or changes the data label position. The data label then shows the names and X- and Y-values of the data points closest to the X-coordinate of the click. The data points must be in six-pixel X-neighborhood, centered on the X-coordinate of the click. The vertical data line that goes through the X-coordinate and the circles around these data points also are shown.

- SHIFT + LBUTTONDBLCLK: Sets the first boundary of the vertical extent of the zoom in Y. The red horizontal line marking this boundary will be displayed.

- SHIFT + `LBUTTONDOWN`: Zoom in. The first click selects the first X-boundary of the zoomX extent. The effect of the second click depends of the previous mouse actions: it sets the second X-boundary if it ios the second SHIFT + left click in row, or it sets the second boundary of the vertical zoom in if it follows the SHIFT + LBUTTONDBLCLK. On `LBUTTONUP`, the container will be zoomed. The previous values of the container's X- od Y-extent will be saved in the X- or Y-zoom/move histories.
- CTRL + `LBUTTONDOWN`: Selects/Deselects the visible chart. The mouse click must be at a neighborhood of any data point of the chart. If there is no data points in the click vicinity and there is a selected chart, it will be deselected. It there are no selected charts, the visible chart with the lesser ID will be selected.
- SHIFT + MOUSEWHEEL: Pans the container along X-axis. The data line and the data label child window will move together with the previously set data line.
- `RBUTTONDOWN`: Invokes the popup menu.
- `LBUTTONDOWN` in the data view: Selects/deselects the row in a data table. The selected data point is shown on the appropriate chart in the container window.

- `RBUTTONDOWN` in the data view: Deselects all selected row in data tables and removes all selected data points of the data view from the container window.

Keyboard commands

- **LET/RIGHT ARROWS**: Pans the chart container along the X-axis.
- **UP/DOWN/PAGEUP/PAGEDOWN**: If there is no selected chart, pans the chart container along Y-axis; otherwise changes the Y-scale of the selected chart.
- **DELETE:** Undoes a first step of horizontal or vertical zoom in.

Popup menu commands

- **"Show Legend"**: Shows the names of the visible charts in the names label window. The window is located at the right upper corner of the container window.
- **"Show Axis X Boundaries"**: Shows/hides the X-axis labels.
- **"View Chart Data"**: Shows the data series of the selected chart as a table in a separate window.
- **"Save/Print Charts"**: Opens the submenu with menu items "Save chart(s)", "Save Charts As Image", and "Print Charts". Any number of visible charts might be saved as XML files, and any number of chart curves might be printed on an 8.5" x 11" page.
- **"Zoom/Move"**: Opens a submenu with the items "Zoom In x", "Move Right", "Move Left", and "Zoom In Y", "Move Up", "Move Down". The container could be zoomed or panned horizontally or vertically. The new zoomed range is 80% of the old range, and the move distance is 10% of the range.
- **"Undo Last Zoom/Move X" and "Undo Last Zoom/Move Y"**: Does just it. This menu items are visible and enables only if the container has a non-empty histories of zooming/panning along X- and Y-axes. The "Undo X" undoes the last horizontal zoom/pan action, the "Undo Y" undoes last vertical action. The positions of the data and name labels are updated if corresponding labels exist. The X-axis labels, if visible, are updated also.
- **"Refresh Charts"**: Restores the initial X- and Y-coordinates of the container. This menu item is visible only after the container is zoomed/panned at least once. The refresh updates the positions of the data and name labels if the container is in tracking mode. If the X-axis labels are visible, they are updated too.
- **"Show/Hide Chart Pnts"**: Shows/hides circles around data points of the selected chart if the minimal X-distance between adjacent points is big enough (about 6 pixels). The chart will remember the state after it will deselected.
- **"Show (chart name)"**: Toggles the visibility of this chart.

Points of interest

First, let us discuss some design solutions.

There are no virtues in placing unrelated data curves in the same window. We suppose to analyze related sets of data. We can expect the X-ranges of related data sets to overlap, albeit not completely.

So I have designed the chart container with a common X-scale and X-axis extent for all its charts. The initial X-extent should be, at least, the union of X-ranges for all charts. The user could select any part of the X-extent for his/her perusal by zooming and/or panning the container.

Because there is only one X-axis for all charts, for the X-axis there might be only one common name, one precision value, and one formatting function.

The charts can display very different values along Y-axis. For example, we can have one chart for altitude (Y) vs. distance (X), and the second chart for the temperature (Y) vs. the same distances (X). Therefore, each chart can have its own Y-axis name, Y precision, and the Y formatting function.

Obviously, for better presentation, the Y scales might be different, even very different for the different data sets. Still, the container is implemented with a common Y-scale and Y-extent for all its charts. Again, the initial Y-extent should be the union of Y-extents of all charts. I have provided means for the user to change the onscreen Y-scale for any selected chart to get the best picture he/she wants.

Because there might be more than one data point for one pixel, I provided the means to zoom and pan along X-axis. You can see every data point after an appropriate zooming and panning.

In my opinion, it eliminates the need for zooming/panning along Y-axis. Zoom enough along the X-axis, and read Y-values point by point all way along. But the reader Alberl was wery insistent asking for the zoom along both axes, so I wilt. Alberl suggested to use a rectangle as a border for a zoomed area: you click and drag the mouse to delineate this area. I found it extremely unconvenient for the user: once you started, you will get zoomed when you released the mouse, even the rectangle is in the wrong place and has wrong size. I ended up with the two separate processes for X- and Y-zooming/panning. You do it separately and you can undo them separately.

What about axes? They are placed automatically. The user should not care about them.

How is the user supposed to get information about the X and Y values of the points in the data series? The data point coordinates and chart names are displayed in the container's child windows. The user selects the X-coordinate, and the container should show the values for all data points closest to the point of request.

Many users/readers are asking for new features, like X- and Y-axes with named ticks, default name labels, etc. I understand them, but I want to say: guys, if you decided to use ChartCtls, you should accept the chart model this control is based on. This is an interactive control which assumes the user is sitting before a PC display. The control renders a minimum info by default, but provides an access to the lot of data. If the user needs some details, he/she can easily get them, but he/she must request them by some actions. If you do not want to click mouse button or depress key, it is OK: you can programmatically control the charts from your main application. For example, there is no and will be no axis tiks with labels, but if you cannot live without them, you have access to the container's X- and Y-extensions. Take them, draw tiks and names in some transparent window, and overlay it over the container. But remember, this is an interactive control, and the user is present. Otherwise, why draw charts?

More about design and implementation will follow.

The chart control code consists of seven headers and six source files (add *stdafx.h* and *stdafx.cpp*). I think it is too much to include all of them in every C++ project, so I made it a static library. Only two header files, *ChartDef.h* and *ChartContainer.h*, must be included in any project using this chart control. Of course, the reference to the library file *ChartCtrlLib.lib* should be included too.

In farther discussion, we will use the aliases for the STL containers (the file *ChartDef.h*):

C++

```
typedef std::basic_string<TCHAR> string_t;
typedef std::basic_stringstream<TCHAR> sstream_t;
typedef std::pair<double, double> PAIR_DBLS;

// Typedefs for data vectors and other STL containers
typedef std::vector<PointD> V_CHARTDATAD;
typedef std::vector<Gdiplus::PointF> V_CHARTDATAF;
typedef std::vector<string_t> V_VALSTRINGS;
typedef std::multimap<int, PointD> MAP_SELPNTSD;

// Used to count multiple Y values for the same X:
// the first member is the iterator to the first occurrence, the second
// is the count of the data points with the same X-coordinate
typedef std::pair<V_CHARTDATAD::iterator, int> PAIR_ITNEAREST;
// Output of some algorithms returning two iterators
typedef std::pair<V_CHARTDATAD::iterator, V_CHARTDATAD::iterator> PAIR_ITS;
typedef std::vector<string_t> V_CHARTNAMES;

// Typedefs for the History container
typedef std::pair <double, double> PAIR_POS;
typedef std::vector<PAIR_POS> V_HIST;

// Typedefs for container chart map
class CChart;
typedef std::map<int, CChart*> MAP_CHARTS;

// For load from file
typedef std::map<string_t, Gdiplus::Color> MAP_CHARTCOLS;
```

Classes

There are nine classes in the library: a class template `PointT`, classes `CChart`, `CDataWnd`, `CPageCtrl`, `CDataView`, `CChartDataView`, `CChartXMLSerializer`, a struct `MatrixD`, and a class `CChartContainer`. The library exports only two classes, `PointT` and `CChartContainer`.

Classes: PointT

The basic representation of a data point in a data series is the instantiation of a class template `PointT` for `double`s (see *ChartDef.h*):

C++

```
template <typename T>
class PointT
{
public:
    PointT(T x = 0, T y = 0) : X(x), Y(y) {}
    PointT(const PointT &pntT) {X = pntT.X; Y = pntT.Y;}
    PointT(const Gdiplus::PointF& pntF) {X = static_cast<T>(pntF.X);
        Y = static_cast<T>(pntF.Y);}
    ................................................
// Conversion function
    operator Gdiplus::PointF()
         {return Gdiplus::PointF(float(X), float(Y));}

public:
    T X;
    T Y;
};

typedef PointT<double> PointD;
```

The class definition also includes overloaded operators =, +, -, *, /, and ==.

Because GDI+ functions accept only REAL (float) floating point numbers, there is the constructor that accepts `Gdiplus::PointF` as a parameter, and the cast operator to cast `PointT` to `Gdiplus::PointF`. You have to make the definition of this class visible at every point of your application where you are calling any container member function that returns or accepts a parameter of type `PointD`. It means you have to include *ChartDef.h*.

Classes: CChart (Chart.h)

It is a rather dumb class. Mainly, it is used as storage for the chart's data series and attributes.

First, it stores a vector of data points, `V_CHARTDATAD` in `CChart::m_vDataPnts`. The vector must be sorted by X-coordinates in ascending order. The vector must have three data points at least, to have at least one data point inside the container's window.

The chart attributes include min and max values of the X- and Y-coordinates of the data points `m_fMinValX`, `m_fMaxValX`, `m_fMinValY`, and `m_fMaxValY`. The container uses them to calculate its horizontal and vertical scales. The member `m_fLocScaleY` stores the multiplier to magnify/decrease the chart's curve along the Y-axis.

Other attributes are visuals: chart's color, dash style, tension, and pen width.

The chart has a unique Idx and name. It also has its own precision for the Y-values, the own Y-axis name, and the own formatting function for the Y-values. The lifetime of uniqueness of the chart Idx and name is the lifetime of the session. The container generates the unique Idx and checks the uniqueness of the supplied chart's name when it adds the chart. If the name is already assigner to the other chart, the container will append the supplied name with the suffix that is the chartIdx. For

example, if the name "SineWave" is already assigned, and the chart Idx = 8, the chart name will be "SineWave_8". If no name is supplied, the container will generate the name "Chart_4". The names with more than 18 characters will be truncated.

The most important member function of `CChart` is `DrawChart (...)`. We will discuss it later.

Classes: Auxiliaries

The classes `CDataWnd`, `CPageCtrl`, `CDataView`, `CChartDataView`, `CChartXMLSerializer`, and struct `MatrixD` encapsulate functionality needed for some user actions (see later).

Classes: CChartContainer (ChartContainer.h)

It is, well, a container: `std::map<int, CChart*>`. This class is also a gateway to all functionality of the chart control. It is the only big class exported by the chart control library. You are never supposed to address all other classes directly: use `CCharContainer` public member functions instead.

I think that the best way to describe the inner working and interaction of the chart control classes and points of interest is to consider the tasks the chart control should perform.

Task: Add chart

The task is performed by the function:

C++

```
int CChartContainer::AddChart(bool bVisible, bool bShowPnts, string_t label,
                              string_t labelY, int precisionY,
                              DashStyle dashStyle, float penWidth, float tension,
                              Color colChart, V_CHARTDATAD& vData, bool bRedraw)
{
  int chartIdx = GetMaxChartIdx() + 1;
  bool bAddIdx = false;
  if (!label.empty())
  {
    label = NormalizeString(label, STR_MAXLEN, STR_NORMSIGN);
    CChart* twinPtr = FindChartByName(label);
    if (twinPtr != NULL)
      bAddIdx = true;
  }
  else
  {
    label = string_t(_T("Cnart"));
    bAddIdx = true;
  }

  if (bAddIdx)
  {
    _TCHAR buffer_t[64];
    _itot_s(chartIdx, buffer_t, 10);  // Chart idx to string
    string_t idxStr(buffer_t);
```

```
    label += string_t(_T("_")) + string_t(buffer_t);
  }

  CChart* chartPtr = new CChart;

  chartPtr->SetChartAttr(bVisible, bShowPnts, chartIdx, label, labelY,
                                    precisionY, dashStyle, penWidth, tension, colChart);

  size_t dataSize = vData.size();

// Now transfer data and set min max values
  if (dataSize > 0)
  {
    chartPtr->m_vDataPnts.assign(vData.begin(), vData.end());
    chartPtr->m_vDataPnts.shrink_to_fit();

// It is cheaper to sort right away than to look for max/min x and sort later if needed
    if (dataSize > 1)
      std::sort(chartPtr->m_vDataPnts.begin(),
                chartPtr->m_vDataPnts.end(), less_pnt<double, false>());

    double minValX = chartPtr->m_vDataPnts.front().X;
    double maxValX = chartPtr->m_vDataPnts.back().X;

// Find min and max Y; works even for one-point vector
    PAIR_ITS pair_minmaxY =
        minmax_element(chartPtr->m_vDataPnts.begin(), chartPtr->m_vDataPnts.end(),
                                                      less_pnt<double, true>());
    double minValY = pair_minmaxY.first->Y;
    double maxValY = pair_minmaxY.second->Y;

// Save in the CChart
    chartPtr->SetMinValX(minValX);
    chartPtr->SetMaxValX(maxValX);
    chartPtr->SetMinValY(minValY);
    chartPtr->SetMaxValY(maxValY);
  }

// Just in case: idx is unique for this container
  if (m_mapCharts.insert(MAP_CHARTS::value_type(
        chartPtr->GetChartIdx(), chartPtr)).second == false)
  {
    delete chartPtr;
    return -1;
  }

// Now update the container's min maxes, saving the history of X
  if (dataSize > 0)
  {
// Wil automatically take care of previous one-point charts
    UpdateExtX(chartPtr->GetMinValX(), chartPtr->GetMaxValX());
    UpdateExtY(chartPtr->GetMinValY(), chartPtr->GetMaxValY());

    if (IsWindow(m_hWnd) && m_bTracking && IsLabWndExist(false))
      PrepareDataLegend(m_dataLegPntD, m_epsX, m_pDataWnd->m_mapLabs, m_mapSelPntsD, true);

    if (bRedraw && IsWindow(m_hWnd)&&IsWindowVisible())
```

```
        UpdateContainerWnds(-1, true);
    }

    return chartIdx;
}
```

The pseudo code for this function is:

- Check whether the data series `vData` has at least three data points; if not, return -1
- Generate the chart's unique Idx for this session
- Process the supplied chart name (label) for uniqueness and change it if necessary (function NormalizeString())
- Allocate the new chart on the heap
- Set chart attributes using the chart member `SetChartAttr(...)`
- Assign the data series `vData` to the chart data member `CChart::m_vDataPnts`
- Sort data by X-coordinate
- Get min/max values of the data series X- and Y-coordinates and set the appropriate data members of the new chart
- Insert the new chart into `m_mapCharts` of the chart container with the key `chartIdx` and the pointer to the new chart as a value
- Update container's X- and Y- extension
- If the container is in the tracking mode, update the map of the points closest to the selected X-coordinate of the container, if the selection exists
- If `bRedraw == true`, redraw the container
- Return the new `chartIdx`

Obviously, if the chart has no data points, steps to process them are skipped. Nothing in rendering other charts will be changed.

The added chart has the default formatting function. If you need to replace it, call `CChartContainer::SetLabYValStrFn`.

Several points in this pseudo code need additional remarks.

The map `MAP_CHARTS` is a `std::map`. By definition, the map is sorted by key values. So to generate the unique chart ID for the current session, you just get the key to the last element of `m_mapCharts` by calling `m_mapCharts.rbegin().first` and increasing the key value by one. The call to `AddChart` is the only way to add a chart to the container, so it will always generate a unique ID for the given container.

Some algorithms we are using in the chart control can operate only on ordered sequences. So the charts data series must be ordered by X-coordinates of the data points. It is much cheaper to call the algorithm `std::sort` outright than iterate over all elements of the data series to find out whether it was sorted and sort it after that scan if it was not. This is a reason we call the `std::sort` on the data vector outright.

To get the chart's X- and Y-extents, we need to find minimum and maximum values of the X and Y coordinates.

In the sorted by X data vector, the min X is the X-coordinate of the first element, and the max X is the last X. It is no-brainer: just get the chart's `m_vDataPnts.front().X` and `m_vDataPnts.back().X`.

The min/max values of the Y-coordinates can be located anywhere in the vector. So I used the algorithm:

C++

```
minmax_element(chartPtr->m_vDataPnts.begin(),
    chartPtr->m_vDataPnts.end(), less_pnt<double, true>())
```

There is no operator '<' for the data point class, so I have to write and use my own predicate, `less_pnt`. Other algorithms used in this library also need similar predicates, so let us go to `less_pnt`.

Predicates and algorithms: less_pnt

We need a predicate capable to work with X or Y values of chart data points, according to our choice. This seems to be very simple:

C++

```
template <typename T, bool bY>
struct less_pnt
{
    bool operator () (const T& Left, const T& Right)
    {
         if (bY) return Left.Y < Right.Y;
         return Left.X < Right.X;
    }
};
```

Still, what can be passed as a `typename`? If you pass some class, the template could be instantiated for any class with public members X and Y. It does not matter what types X and Y are: they only have to have the operator <. What will happen if this operator is not defined? The compiler will complain only at the moment of template instantiation for this weird class. It might happen much, much later in the development or long later, during a software upgrade.

So it is better to use same POD data as a template parameter and `PointT<T>` as an argument to the operator `()` (thanks to Igor Tandetnik).

Second, because the value of the non-type parameter must be known at compilation time, only one of the branches of the '`if`' statement is executed at run time for each particular instantiation of this template. It is unknown whether the compiler will optimize out the other branch. Maybe we are going to have an unnecessary comparison at run time. It is better to optimize it by hand using a partial specialization of the template. We will have:

C++

```
template <typename T, bool bY>
struct less_pnt
{

    bool operator () (const PointT<T>& Left, const PointT<T>& Right)
    {
         return Left.Y < Right.Y;
    }
};

// Partial specialization for X
template <typename T> struct less_pnt<T, false>
{
    bool operator () (const PointT<T>& Left, const PointT<T>& Right)
    {
         return Left.X < Right.X;
    }
};
```

The data space, the window client surface, and transforms

The data points in the chart's data vector are in the chart's data space. It is natural to think about this space as a rectangle in the Cartesian (rectangular) coordinate system. The left and right boundaries of this rectangle are the minimal and maximal X-coordinates of the data points; the top and bottom are maximal and minimal Y-coordinates. Because normally in scientific data the Y-axis goes up, the top value is greater than the bottom value.

The data space of the container is a union of the data spaces of all charts in the container. It also is a rectangle. Usually we delegate to the container the calculation of the container data space, but if needed, we can set the boundaries of the container data space from outside the container, using the functions:

C++

```
bool CChartContainer::UpdateExtX(double minExtX, double maxExtX, bool bRedraw = false)
void CChartContainer::UpdateExtY(double minExtY, double maxExtY, bool bRedraw = false)
```

(The parameter `bRedraw = true` forces the immediate update of the container window.)

These functions are calculating the minimal and maximum X and Y extensions. For example, if the greatest value of the charts maxX is 10, but you have passed 100 as a parameter maxExtX, the max extension X will be set to 100.

There is an another couple of functions,

```
PAIR_DBLS SetExtX(bool bRedraw = false);
```

```
PAIR_DBLS SetExtY(bool bRedraw = false);
```

These functions calculate the unions of X- and Y- extebsions of all charts in the container, and set the results as container dimension in the data space.

If there is histor y of X and/or Y zooming/panning, the functions substitute the new extension limits at the front of the history vectors; you will see the new X limits only if you undo all zooming/panning along the X-axis, and the new Y extent after undoing the Y-history.

The origin and axes of the coordinate system might be at any position relative to the data space rectangle: inside, to the left, below the bottom, etc.

We might use the entire container data space or only part of it (think about zooming and panning).

At any given moment, the boundaries of the used container data space are stored in `CChartContainer` data members:

C++

```
double m_minExtY;       // Min coordinate of the Y axis
double m_maxExtY;       // Max coordinate of the Y axis
double m_startX;        // Leftmost X coordinate
double m_endX;          // Rightmost X coordinate (not included)
```

We will refer to the pair `m_minExtY`, `m_maxExtY` as a container's Y-extent, and to `m_startX`, `m_endX` as a container's X-extent.

When we are drawing charts in the container window, we are drawing in the container's client space. The origin of the coordinate system here is at the left upper corner of the client rectangle, and the Y-axis goes down (top is less than bottom). Therefore, before the drawing starts, we have to map the container data space into the client space.

The transform matrix

The mapping into the client space consists of translations and scaling. These transforms are described by a transform matrix. In two-dimensional graphics, it is:

Here a_{11} = scale X, a_{22} = scale Y, a_{31}= offset X, a_{32} = offset Y. For the chart control, the transforms are translations and scaling only, so a_{12} and a_{21}will always equal to zero.

In GDI+, we have the `Matrix` class. Unfortunately, the elements of the transform matrix have the type `float`.

In theory, the container's data space might be completely or partially out of the range of the type `float`. Passing out of range data to the transform or to drawing functions of GDI+ could result in errors. So we have to transform the data space into a client space working with `double` values first, and cast the results to `float` type next. These casts will always be in the range of the `float` because the values of the client coordinates (logical pixels) are limited by the range of the type `int`. Of course, we will lose precision, but this does not matter: the `float` coordinates will be rounded to pixels eventually. What about a reverse transform, from the client space to the data space? Well, we have to be cautious, always remember about lost precision. For example, if we are looking for a data point that corresponds to a given pixel, we have to look for the data point closest to the pixel transformed into the data space, not equal to it.

There is no point in making this matrix a generic template class: to transform from the screen to the data space, we have to include the inversion operation in the class. The inversion demands division, so only floating-point numbers will do. We already have `Gdipus::Matrix` for `float`s, so there is the transform matrix class for `double`s (*ChartDef.h*):

C++

```
// This MatrixD is only for translation and scaling of double numbers
typedef class MatrixD
{
public:
    double m_scX;
    double m_scY;
    double m_offsX;
    double m_offsY;
public:
    // The constructor yields an identity matrix by default
    MatrixD(double scX = 1.0, double scY = 1.0,
    double offsX = 0.0, double offsY = 0.0): m_scX(scX),
                        m_scY(scY), m_offsX(offsX), m_offsY(offsY) {}
    // Transforms
    void Translate(double offsX, double offsY)
    {m_offsX += offsX*m_scX; m_offsY += offsY*m_scY;}
    void Scale(double scX, double scY)
    {m_scX *= scX; m_scY *= scY;}

    // Operations on matrixD; if the matrix is not invertible,
    //returns false; uses explicit formulae for inversion
    bool Invert(void);

    MatrixD* Clone(void)
    {
        MatrixD* pMatrix = new MatrixD;
        pMatrix->m_scX = m_scX;
        pMatrix->m_scY = m_scY;
        pMatrix->m_offsX = m_offsX;
        pMatrix->m_offsY = m_offsY;
        return pMatrix;
    }

    // Transforms PointD to PointF and PointF to PointD
    Gdiplus::PointF TransformToPntF(double locScY, const PointD& pntD);
```

```
    PointD TransformToPntD(double locScY, const Gdiplus::PointF& pntF);
private:
    MatrixD(const MatrixD& src);
    MatrixD operator =(const MatrixD& src);
} MATRIX_D;
```

`Gdipus::Matrix` allows only a clone operation, so to behave similarly, the copy constructor and the assignment operator in `MATRIX_D`are made private.

The default constructor creates an identity matrix `I`. To calculate the transform matrix, the chart control must multiply the identity matrix by the scale matrix `S` and/or the translation matrix `T`:

That is done by applying `Scale` and `Translate` functions to the instance of `MATRIX_D`. Matrix multiplications are not commutative. We use a `MatrixOrderPrepend` order in the chart control, when the multiplier is placed to the left of the original matrix. It means that if we want to scale first and translate the scaled results next, we have to multiply in this order:

The equivalent of this is:

```
myMatrix.Translate(...); //First
myMatrix.Scale (...);    // Second
```

The result is the vector:

To invert a matrix, I used explicit formulae for the matrix with only translation and scale members:

$a_{31} = -a_{31}/a_{11}$;

$a_{32} = -a_{32}/a_{22}$;

$a_{33} = 1.0$;

$a_{11} = 1.0/a_{11}$;

$a_{22} = 1.0/a_{22}$;

$a_{12} = a_{13} = a_{21} = a_{23} = 0$.

To get the drawing rightly, we have to scale the container data space to the client space and translate the results to the origin in the client space.

Task: Get the transform matrix

First, in what function should we calculate the transform matrix? We must have the correct transform matrix every time we are going to draw the charts. The best place for it is in the `CChartContainer::OnPaint()`. We always call this function, directly or indirectly, after changes to the container are made.

We begin with the container's X- and Y-extensions that are either calculated by the container or set by the application. For the translation in the client space, we need to know the axes origin.

The functions:

```
PAIR_XAXPOS CChartContainer::GetXAxisPos(RectF rChartF,
                                         double minY, double maxY)
PAIR_YAXPOS CChartContainer::GetYAxisPos(RectF rChartF,
                                        double startX, double endX)
```

calculate the Y-position of the X-axis and X position of the Y-axis.

The idea is very simple: if the minimum is negative, and maximum is positive, place the axis between them; otherwise, attach it to the appropriate border of the client rectangle. E.g., for the X position of the Y-axis:

```
if ((startX < 0)&&(endX > 0))
{
    double offsYX = rChartF.Width*fabs(startX)/(endX - startX);
    horzOffs = rChartF.GetLeft() + float(offsYX);
    // Somewhere between minX, maxX
}
else if (startX >= 0)
    horzOffs = rChartF.GetLeft();
else if (endX <= 0)
    horzOffs = rChartF.GetRight();
```

We declare `offsYX` as `double` because X- and Y-extensions are `double`s, but the calculations are in the client space.

The axes origin in the client space is `Gdiplus::PointF(offsXY, offsYX)`.

For the scaling, the function `CChartContainer::UpdateScales(drawRF, m_startX, m_endX, m_minExtY, m_maxExtY)` does the job. It just calculates:

```
scX = drawRectWidth/rangeX;
scY = drawRectHeight/rangeY;
```

Before calculation, we have deflated the client rectangle width and height 10% to make the picture look better.

With the offsets and scales, we are ready to calculate the transform matrix:

```
MatrixD matrixD;
matrixD.Translate(pntOrigF.X, pntOrigF.Y);
matrixD.Scale(m_scX, -m_scY);
```

The minus sign in the third line reverses the direction of the Y-axis.

We are not done yet. Remember, when the origin is out of the client rectangle, we just place the axis along the left or right or top or bottom boundaries of the rectangle. In this instance, one more translation is due; this is in the data space:

```
matrixD.Translate(translateX, translateY);
```

It is a translation to the borders of the data space: `translateX` is `-startX` (left position of the Y-axis) or `-endX`, and `translateY` is `-minExtY` (top position of the X-axis) or `-maxExtY`. The order of the transforms is: first, if needed, translate the origin in the data space; second, scale; third, move the result to the origin in the client space.

Sometimes we need the transform matrix before `OnPaint()` is called, e.g., to track the data label. The function `MatrixD* CChartContainer::GetTransformMatrixD(double startX, double endX, double minY, double maxY)` calculates the matrix for a given X- and Y-extents outside the `OnPaint()`.

To see the transform code, you have to go to this function or to `CChartContainer::OnPaint()`.

Task: Draw the chart

Now we have the transform matrix and can draw the visible charts. This task is performed by the function:

```
bool CChart::DrawChartCurve(V_CHARTDATAD& vDataPntsD, double startX, double endX,
                  MatrixD* pMatrixD, GraphicsPath* grPathPtr, Graphics* grPtr, float
dpiRatio)
{
  if (vDataPntsD.size()== 0)
  // Just for safe programming; the function is never called on count zero
    return false;

  V_CHARTDATAF vDataPntsF;
// Convert the pntsD to the screen pntsF
  if (!ConvertChartData(vDataPntsD, vDataPntsF, pMatrixD, startX, endX))
    return false;

  V_CHARTDATAF::iterator itF = vDataPntsF.begin();
  V_CHARTDATAF::pointer ptrDataPntsF = vDataPntsF.data();
  size_t vSize = vDataPntsF.size();

// Add the curve to grPath
  Pen pen(m_colChart, m_fPenWidth*dpiRatio);
  pen.SetDashStyle(m_dashStyle);
  if (!m_bShowPnts&&(vSize == 2))   // Are outside or at boundaries of clipping area
  {                                 // Make special semi-transparent dash pen
    Color col(SetAlpha(m_colChart, ALPHA_NOPNT));
    pen.SetColor(col);
  }
```

```
  if (m_dashStyle != DashStyleCustom)
  {
    if (vSize > 1)
    {
      grPtr->DrawCurve(&pen, ptrDataPntsF, vSize, m_fTension);

      if (m_bSelected && (dpiRatio == 1.0f))  // Mark the chart as selectes on screen only
      {
        Pen selPen(Color(SetAlpha(m_colChart, ALPHA_SELECT)),
                   (m_fPenWidth + PEN_SELWIDTH)*dpiRatio);
        grPtr->DrawCurve(&selPen, ptrDataPntsF, vSize, m_fTension);
      }
    }

// Now add the points
    if (m_bShowPnts || (vSize == 1))
    {
      itF = adjacent_find(vDataPntsF.begin(), vDataPntsF.end() ,
                          lesser_adjacent_interval<PointF,
                          false>(PointF(dpiRatio*CHART_PNTSTRSH, 0.0f)));
      if (itF == vDataPntsF.end())    // All intervals are greater than CHART_PNTSTRSH
      {
        itF = vDataPntsF.begin();    // Base
        for (; itF != vDataPntsF.end(); ++itF)
        {
          RectF rPntF = RectFFromCenterF(*itF, dpiRatio*CHART_DTPNTSZ,
                                                               dpiRatio*CHART_DTPNTSZ);
          grPathPtr->AddEllipse(rPntF);
        }
      }
    }
  }
  else
  {
    PointF pntF;
    PointF pntFX(dpiRatio*CHART_DTPNTSZ/2, 0.0f);
    PointF pntFY(0.0f, dpiRatio*CHART_DTPNTSZ/2);

    for (; itF != vDataPntsF.end(); ++itF)
    {
      pntF = *itF;
      grPathPtr->StartFigure();
      grPathPtr->AddLine(pntF - pntFX, pntF + pntFX);
      grPathPtr->StartFigure();
      grPathPtr->AddLine(pntF - pntFY, pntF + pntFY);
    }
    if (vSize == 1)
    {
      grPathPtr->StartFigure();
      grPathPtr->AddEllipse(RectFFromCenterF(pntF, 2.0f*pntFX.X, 2.0f*pntFY.Y));
    }
  }

  if (grPathPtr->GetPointCount() > 0)          // Has points to draw
  {
    pen.SetWidth(1.0f*dpiRatio);
    pen.SetDashStyle(DashStyleSolid);
```

```
    grPtr->DrawPath(&pen, grPathPtr);
    if (((m_dashStyle == DashStyleCustom)||(vSize == 1))&& m_bSelected && (dpiRatio ==
1.0f))
    {
      pen.SetColor(Color(SetAlpha(m_colChart, ALPHA_SELECT)));
      pen.SetWidth(m_fPenWidth + PEN_SELWIDTH);
      grPtr->DrawPath(&pen, grPathPtr);
    }
    grPathPtr->Reset();
  }
  return true;
}
```

The parameters `startX`, `endX` might cover the entire container's X-extension or only part of it (after zooming in or panning). The parameter `dpiRatio` is for adjusting pen widths for printing. We will discuss it later.

The function `DrawChartCurve` performs following operations:

- It searches for two border data points. These points should be closest to, but not inside the range `startX`, `endX`.
- Converts the part of the data vector between these data points to the client space float coordinates using parameters `startX`, `endX`, and the pointer to the transform matrix `pMatrixD`.
- It draws the curve with transformed data points using the pen with the chart's color, dash style, width, and using the chart's tension. If the dash style is `DashStyleCustom`, the function does not draw a curve; instead it renders the chart data points as crosses.
- If the chart is selected, overdraws the chart a second time with a semi-transparent color and increased pen width (see the screenshot at the beginning of this article).
- Gets the minimal screen distance between two adjacent data points; if it is greater than six pixels, draw the circles around all visible charts' data points.
- If the chart has only one data point, it draws only a circle around the point.

First, let us consider the search for the border data points.

Assume we have the X-extension of the data space -10.0, 10.0, and the space between data points X-coordinates equals 2.0. If `startX`= -7.0 and `endX` = +7.0, the leftmost points inside the drawing range has X=-6.0, and the rightmost point inside the range has X=6.0.

If we call `Gdiplus::DrawCurve` on the inner points only, the curve will run from -6.0 to +6.0, not from -7.0 to +7.0. We will perceive it as a curve with beginning at X=-6.0 and end at X=6.0. However, the curve definitely exists outside of these points. To have a beautiful curve covering all X-extent, we have to extend the data point set to the nearest left and right data points outside or at the limits of the X range.

The algorithm and predicate to do this job are in *Util.h*. The algorithm is used by the function:

```
PAIR_ITS CChart::GetStartEndDataIterators(V_CHARTDATAD& vDataPnts, double startX, double
endX)
{
    PAIR_ITS pair_its = find_border_pnts(vDataPnts.begin(),
       vDataPnts.end(), not_inside_range<double, false>(startX, endX));
    return pair_its;
}
```

The predicate is:

```
template <typename T, bool bY> struct not_inside_range
```

We use only partial specialization for the X-coordinate:

```
template <typename T >
struct not_inside_range<T, false>
{
    T _lhs;
    T _rhs;
    bool _bFnd;
    not_inside_range(T lhs, T rhs) : _lhs(lhs), _rhs(rhs), _bFnd(false) {}

    inline std::pair<bool, bool> operator () (const PointT<T>& pntT)
    {
         bool bLeft = false;
         bool bRight = false;
         if (pntT.X < _lhs)
               bLeft = true;
         else if (!_bFnd && (pntT.X == _lhs))
         {
               bLeft = true;
               _bFnd = true;
         }
         else if (pntT.X >= _rhs)
               bRight = true;
         std::pair<bool, bool> pair_res(bLeft, bRight);
         return pair_res;
    }
};
```

The predicate takes the point's coordinate (here it is `pntT.X`) and tests it against the left boundary of the interval first. If the coordinate is to the left of or at the left boundary, the predicate returns `std::pair(true, false)`. If the coordinate is to the right of or at the right boundary, the predicate returns `std::pair(false, true)`. If the coordinate is inside the interval, the return value is `pair(false, false)`.

For a multi-valued function, the predicate selects the first point from the group of points with the same X-coordinates and ignores all other points in the group.

The algorithm iterates over the range of iterators `_First, _Last on the ordered sequence`. The algorithm refreshes the first iterator in the internal pair of iterators each time the predicate returns `pair_res.first = true`. On the first return of `pair_res.second = true`, the algorithm saves the current iterator in the second member of the pair. Because the sequence is ordered, there is no need to continue the test. The algorithm returns this pair of iterators.

Next, we have to map the points between iterators returned by `GetStartEndDataIterators` from the data space to the client space. The algorithm `std::transform` with the custom predicate `transform_and_cast_to_pntF (ChartDef)` does the job:

```
// Predicate to use with the STL algorithm transform
typedef struct transform_and_cast_to_pntF
{
  double _locScY;
  MatrixD* _pMatrixD;

  transform_and_cast_to_pntF(double locScY, MatrixD* pMatrixD) :
                              _locScY(locScY), _pMatrixD(pMatrixD) {}
  inline Gdiplus::PointF operator() (const PointD& pntD)
  {
    double X = _pMatrixD->m_scX*pntD.X + _pMatrixD->m_offsX;
    double Y = _locScY == 1.0 ? pntD.Y : _locScY*pntD.Y;
    Y = _pMatrixD->m_scY*Y + _pMatrixD->m_offsY;
    return Gdiplus::PointF(float(X), float(Y));
  }
} TRANSFORM_TO_PNTF;
```

The predicate just applies the transform matrix `_pMatrixD` to each point in the algorithm range and casts the result to `Cdiplus::PointF`. Before applying the matrix, the predicate multiplies the Y-coordinate of the point by some value. The value is stored in the predicate member `_locScY`. It allows us to modify the Y- scale of the given chart to change the vertical size of the chart curve without modifying the chart's data space. The predicate gets `_locScY` and `_pMatrixD` at the construction time.

All this preparation job, search for border points, transform, and cast, is done in the function `Chart::ConvertChartData`. The function returns the resulting vector of `PointF` in the function `out` parameter `std::vector<Gdiplus::PointF> vDataPntsF`.

To draw the chart, we just call `Gdiplus::DrawCurve` on all dash styles except DashStyleCustom.

`Gdiplus::DrawCurve` accepts only a pointer to the arrays of `Cdiplus::PointF`. To get the pointer, we call the `std::vector::data()` function:

```
V_CHARTDATAF::pointer ptrDataPntsF = vDataPntsF.data()
```

The number of data points per pixel changes when the user changes the container's X-extent. For example, in the demo app, the width of the client rectangle is 476 pixels. If the chart has 1000 data

points, it equals 2.1 data points per pixel. Suppose we zoomed in the container, and now only 10 data points are visible in the client rectangle. Now we have 47.6 pixels between adjacent data points. To distinguish between the actual data points and the spline interpolation pixels, we have somehow to mark the actual data points. We do it by drawing the circles around the data points.

How to decide when to draw these circles? We have chosen to draw circles if the minimal distance between adjacent data points is greater than or equal to six pixels. This decision is based entirely on aesthetic considerations: the chart curve with circles looks not too cluttered at this distance.

Therefore, `DrawChartCurve` applies the algorithm `std::adjacent_find` with the predicate `adjacent_interval_pntF` to `vDataPntsF`(*Util.h*). If the minimal distance between the adjacent data points is greater or equal to 6.0 pixels, `DrawChartCurve` adds circles around the data points to the graphics path, and draws the path. We use the minimum as criterion to avoid ambiguity. Otherwise, there is no way to know if some curve segment is empty or overpopulated by data points.

Sometimes it is not desirable to draw these circles. The member `CChart::m_bShowPnts` suppresses drawing of circles if it is set to `false`.

And what about `DashStyleCustom`? We reserved it to draw a chart as a set of disconnected data points. Each data point is represented as a small cross. There is no way to make `Gdiplus::DrawCurve` to draw disconnected crosses with any `DashStyle`. So we use the `DashStyleCustom` as a flag to switch to another drawing routine. To draw the set of the points, we add each cross to `GraphicsPass` instance, `grPass`. To have the cross's lines disconnected from each other, we have to prefix the insertion of each line of each cross with `grPath::StartFigure()`:

C++

```
for (; itF != vDataPntsF.end(); ++itF)
{
  pntF = *itF;
  grPathPtr->StartFigure();
  grPathPtr->AddLine(pntF - pntFX, pntF + pntFX);
  grPathPtr->StartFigure();
  grPathPtr->AddLine(pntF - pntFY, pntF + pntFY);
}

Graphics* grPtr;
grPtr->DrawPath(&pen, grPath);
```

Task: Drawing the container (flicker free drawing)

To draw without flicker, we use double-buffered drawing: we draw into memory first, and transfer the result onto a display's surface after that. In MFC GDI, it is all about `CreateCompatibleDC`, selecting a bitmap in this DC, and finally, transfers the bitmap bits from the in-memory DC into the screen DC.

The GDI+ analog of this task is:

```
CPaintDC dc(this);     // Device context for painting
Graphics gr(dc.m_hDC); // Graphics object from paintDC
// Although we use only floating-point numbers with Gdiplus
// functions, we start with integers because no Bitmap constructor
// accepts REAL (float) numbers
Rect rGdi;
gr.GetVisibleClipBounds(&rGdi);  // Same as GetClentRect

Bitmap clBmp(rGdi.Width, rGdi.Height);          // Mem bitmap
Graphics* grPtr = Graphics::FromImage(&clBmp);  // In-memory

RectF rGdiF = GdiRectToRectF(rGdi); // From Util.h, to allow
// float numbers

// Draw with grPtr
.............................................

gr.DrawImage(&clBmp, rGdi);      // Transfer into the screen
delete grPtr;                   // Free the memory
```

In addition, when we need to update the container's window, we call the function:

```
void CChartContainer::RefreshWnd()
```

This functions calculates the region that excludes areas under the container's children, and calls `RedrawWindow()` on this region. We use the regions because even double-buffered drawing sometimes blinks.

If the children of `CChartContainer` are visible, and/or the data table of one of the visible chart is displayed in the data view window, `RefreshWnd()` will not update these windows. The user should call:

```
void CChartContainer::UpdateContainerWnds(int chartIdx, bool bMatrix, DATAVIEW_FLAGS
dataChange)
{
  if (m_bTracking && IsLabWndExist(false))
  {
    UpdateDataLegend(bMatrix);
  }
  else
    RefreshWnd();

  if (IsLabWndVisible(true))
    ShowNamesLegend();

  UpdateDataView(chartIdx, dataChange);
}
```

This function updates the data label window if it exists, redraws the chart names label (again, if it is visible), and updates the data view. Because the need for update of the data label might arise when

the X-extension of the container was changed with zooming or panning, we will need to recalculate the transformation matrix (`bMatrix == true`). The flag `dataChange` tells the data view that the chart's data vector is changed and need a special treatment. The function has default parameters: `chartIdx = -1`, `bMatrix = false`, `DATAVIEW_FLAGS = F_NODATACHANGE`. We are using the defaults if there is no dataView window, no change of the X-extension (e.g. we hide or show the chart). If the data view is present, and the chart name, formatting function, or other chart attributes (but not the data vector) are changed, we have to pass to the function the chart Idx only.

Task: Show data point values

We see the charts in the container widow, and we want to know exact values of the charts' data points at the selected value X_0 of the X-axis. It must be easy for the user: just select X_0, and the container will show the X and Y values of the charts' data points closest to this X_0. X_0 should be selected by left mouse click or programmatically. We will call X_0 a request point. If the user has done with it, he can order the container to delete this information or to go to the next X. Otherwise, the container should track X_0 when the container client space is changing.

Now let us go to the details. The process consists of three tasks:

- Collecting a set of data points closest to the request point X_0 and visible in the container's window
- Rendering of the collected data points onto the screen
- Tracking the collected points and their image on the screen

Let us start with the first task.

The MFC Framework supplies coordinates of the mouse click in the client space. The charts' data points are in the data space. We cannot work in the client space because the conversion from the `double`s of the data space to the `float`s in the client space could result in loss of precision. E.g., float(1.0e-234) = 0.0f. So we have to work in the data space. To map the mouse point into the data space point, we use the function (*ChartDef.h*):

```
PointD MatrixD::TransformToPntD(double locScY, const Gdiplus::PointF& pntF)
{
    ENSURE(m_scX*m_scY != 0.0);  // The matrix must be invertible
    MatrixD* matrixDI = Clone(); // Do not change the original
    matrixDI->Invert();
    // Map back into data space
    double X = pntF.X*matrixDI->m_scX + matrixDI->m_offsX;
    double Y = pntF.Y*matrixDI->m_scY + matrixDI->m_offsY;
    Y /= locScY;                // Correct for the local scaleY

    delete matrixDI;
    return PointD(X, Y);
}
```

Many interesting things are going here.

Obviously, to map back from the screen to the data space, we have to invert the transform matrix that was used to transform the data space into the client space. The first line in the body of the function checks whether the matrix is invertible. Second, the `Invert` function calculates the data members of the inverted matrix. Because we are using the direct matrix in every drawing procedure, we have to operate on the clone of it.

As you remember, before the direct transform, the X-coordinates of the chart's data points are multiplied by the chart's local scale Y to change the Y-size of the chart's curve on the screen. So after inversion, we must divide the Y-coordinates by `locScY`.

The coordinates of the given request point in the data space will not change. If it was X = 4.5, it will be X = 4.5 until the next click or request. In the client space, the position of the given request point could change due to zooming, panning, or if the container window changes its size. So let us to keep the coordinates of the request point in the data space in the data member `CChartContainer::m_dataLegPntD`. We will use this data member to track the request point on the screen.

Task: Select data points to show

We will collect all chart data points closest to X_0 into `std::multimap CChartContainer::MAP_SELPNTSD m_mapSelPntsD`. We have to use the multimap because some charts might have several data points with the same X (e.g., the rectangle wave in the demo app).

What does "closest" mean? Obviously, at the moment of selection, 'closest' means "visually close" to the place of mouse click. In this chart control, "visually close" is defined as the data point located in a six-pixel interval centered on X_0. Again, the image of this interval in the data space is a constant for the lifetime of the point of request. We save it in the data member `CChartContainer::m_epsX`. The value of `m_epsX` in the client space is always six pixels at the moment of the click, but it should follow to the size of the X-extent of the container window. E.g., the six-pixel interval will be twelve pixels wide if the container X-extent is made a half of the previous extent. For the new click in this new X-extent, the new interval is six pixels again.

The six-pixel interval might cover a multitude of the chart's data points or cover no data points at all. (In the demo app, it might be 12 or more data points per pixel at X-extent -10.0...10.0.) Therefore, the "closest" chart's data point should be in `m_epsX` and has the minimal distance from X_0. In addition, if several data points satisfy this condition (multi-valued data series), we have to select all of them.

The function `GetNearestPoint` selects the points:

```
PAIR_ITNEAREST CChart::GetNearestPointD(const PointD& origPntD, double dist, PointD&
selPntD)
{
```

```
    V_CHARTDATAD::iterator it = m_vDataPnts.begin(),
    itE = m_vDataPnts.end();
    int nmbMultPntsD = 0;
    double leftX = origPntD.X - dist/2.0;
    double rightX = origPntD.X + dist/2.0;
    // Find the first point in vicinity of the origPntsD.X, if it exists
    it = find_if(it, itE,
    coord_in_range<double, false>(leftX, rightX));
    if (it != itE)  // Than find closest to origPntD.X
    {
        it = find_nearest(it, itE,
        nearest_to<double, false>(origPntD, dist));
        if (it != itE)
        // Always true; will return
        // found_if result at least
        {
            // Now get the number of multi-valued points (the same X's, different Y's)
            selPntD = *it;
            nmbMultPntsD = count_if(m_vDataPnts.begin(), it,
              count_in_range<double, false>(selPntD.X, selPntD.X));
            return make_pair(it, nmbMultPntsD + 1);
        }
    }
    return make_pair(itE, 0);
}
```

In the first element of `PAIR_ITNEAREST`, the function returns the iterator to the first of the nearest to `origPntD.X`data points of the chart's data vector. The number of data points with the same X-coordinate is returned in the second element of the pair.

The function uses three STL algorithms.

First, it applies the algorithm `std::find_if` to the entire data vector to search for the first data point with X-coordinate in the interval `origPntD.X ± dist/2.0`. The parameter `dist` defines the interval in the data space where the "closest" points might be. Usually, it is the same six-pixel "visual close" interval transformed into the data space.

Next is the search for the data point with the minimal distance from X_0. The search starts from the iterator returned by `std::find_if`. The search is conducted in the same interval `origPntD.X ±dist/2.0`. This algorithm is custom-made with the custom predicate:

```
//Find closest to X or Y coord of some point of origin; apply to sorted
// sequences only
template<class _InIt, class _Pr>
inline _InIt find_nearest(_InIt _First, _InIt _Last, _Pr _Pred)
{
     _DEBUG_RANGE(_First, _Last);
     _DEBUG_POINTER(_Pred);

     _InIt _NearestIt = _First; // Find first _InIt satisfying _Pred
     for (; _First != _Last; ++_First)
```

```
    {
      if (_Pred(*_First))
           break;
     _NearestIt = _First; // Continue
    }
   return (_NearestIt);
}
```

The predicate `_Pred` compares the distance between adjacent points `fabs(dataPntD.X - origPntD.X)` with the previous value stored in the predicate's data member. The data vector is sorted by X, so the distance will always decrease when we are moving to `origPnt.X`, might decrease on the move to the first point after `origPnt.X`, and always increase after that. On the first iteration that yields increased distance, the predicate will return `true`. The algorithm immediately breaks the loop and returns the iterator to the previous point that is the closest to `origPntD.X`. Again, we use the template definition for the search for Y-coordinates, and a partial specialization for the search for X.

Finally, the algorithm `std::count_if` counts the number of data points with the same X-coordinates as the closest point has. Keep in mind that only the first search operates on the entire data vector. The second search and counting are performed in the tiny interval `dist` around the point of request.

All selected data points, visible or not, are stored in the multimap `CChartContainer::m_mapSelPntsD`. For the lifetime of the given point of request, the multimap does not change if the container does not add or removes charts, or changes the charts data vectors. When the "closest" points are selected, we can discuss how to render them.

Task: Show the data label

To render the selected points, we have to show the request point, the visible selected points, and the X and Y values of them together with the names of the charts they belong to, and names of charts Y-values. It is convenient to show the request point as a vertical data line (should I say the line of request?) that goes through `origPntD.X`. We will show the visible selected data points as circles around the corresponding data points. It results in the window of a very irregular shape: a line from the top to the bottom of the container client area, a set of circles around the selected data points, and a rectangle with the text strings.

The better way to do it is to use the layered window. Unfortunately, the layered window cannot be a child of the other window, for example, the chart container's window. (It can be owned.) It means that all the work that the Operating System and the MFC Framework are doing when the owner is moving, resizing, or is under the other windows, you have to do by yourself.

Using the child window with the style `WS_EX_TRANSPARENT`, which covers all area as it is shown on the picture above brings problems with handling the mouse events that occurs over the child, but should

be transferred to the container for handling, and other complications.

So I decided to do a little hack: I have left the drawing of the data line and the selected points to the container, and delegated the drawing of the strings with data point info to the container's child. It frees me of basic housekeeping (the child moves, hides, and closes together with its parent). As a price, I accepted the task to remember to draw both in the container client area and in the child client area in the same time, and to refresh both windows when appropriate.

The class `CDataWnd` is responsible for drawing the data point's info. The container keeps the pointer to `CDataWnd* m_pDataWnd` as its data member. The container allocates the memory on the heap for this pointer, and passes to `m_pDataWnd` info strings related to the visible selected data points. After receiving this info, `m_pDataWnd` is ready to draw the data label.

What does this data point info consist of?

To make identification of the data points easy, we display the info string in the color of its chart. The short line before the chart's name has the same color, dash style, and pen width as the chart curve has. It means that the container has to pass to `m_pDataWnd` not only the strings of the values of the data point coordinates, but also the chart and X- and Y-value names, and visual data like color, dash style, and pen width.

This info is passed as a tuple (*ChartDef.h*):

C++

```
typedef std::tuple<string_t, string_t,string_t, string_t,
        string_t, Gdiplus::Color, Gdiplus::DashStyle, float> TUPLE_LABEL;
```

Why do we pass five strings instead of one? It is because, for a pretty picture, we want to display the data point info in columns: chart names, X-values name (the same for all charts), formatted X-values, Y-values names for each chart, and corresponding formatted Y-values.

I have a confession to make. At first, I took tuples as an unnecessary gimmick that ISO C++ committee invented to make life harder for us, Joe programmers. I thought that to have structures is enough. But later, I began to appreciate how easy and uniform access to tuple elements might be. I am using enums and the function `std::get<>(...)` to do this. Compare the verbose access to the members of a structure to this:

C++

```
enum TUPLE_LIDX {IDX_LNAME, IDX_LNAMEX, IDX_LX, IDX_LNAMEY, IDX_LY, IDX_LCOLOR, IDX_LDASH,
IDX_LPEN};
TUPLE_LABEL tuple_label;
get<IDX_LNAME>(tuple_label) = string_t(_T("SineWave_0"));
Gdiplus::Color chartCol     = get<IDX_COLOR>(tuple_label);
```

To get the tuples, the container on each visible point from `m_mapSelPntsD`, call the function:

C++

```
// Formats string and prepares chart visuals for the screen

TUPLE_LABEL CChart::GetSelValString(const PointD selPntD, string_t nameX,
            int precision, val_label_str_fn pLabValXStrFnPtr)
{
  TUPLE_LABEL tuple_label;
  get<IDX_LNAME>(tuple_label)  = m_label;
  get<IDX_LNAMEX>(tuple_label) = nameX;
  bool bAddEqSign = nameX.empty() ? false : true;
  get<IDX_LX>(tuple_label)     = pLabValXStrFnPtr(selPntD.X, precision, bAddEqSign);
  get<IDX_LNAMEY>(tuple_label) = m_labelY;
  bAddEqSign = m_labelY.empty() ? false : true;
  get<IDX_LY>(tuple_label)     = m_pLabYValStrFn(selPntD.Y, m_precisionY, bAddEqSign);

  int alpha = max(m_colChart.GetAlpha(), 128);  // TODO: Use definition instead of number
  Color labCol = SetAlpha(m_colChart, alpha);
  get<IDX_LCOLOR>(tuple_label) = labCol;

  get<IDX_LDASH>(tuple_label) = m_dashStyle;
  get<IDX_LPEN>(tuple_label)  = m_fPenWidth;

  return tuple_label;
}
```

Let us talk about precision. It is a container precision, set by the user or by an external application. The function just passes it to the pointer to the containers formatting function, `pLabValXStrFnPtr`:

```
get<IDX_LX>(tuple_label) = pLabValXStrFnPtr(selPntD.X, precision, bAddEqSign);
```

The container packs the tuples in `multimap std::multimap<int, TUPLE_LABEL>`, and passes the `multimap` to its member `CDataWnd* CChartContainer::m_dataWnd`. The multimap keys are the chart IDs. We use the multimap because the chart data vector might have multiple data points with the same X and different or the same Y coordinates (think about a rectangle wave).

The chart container fills this `multimap` with tuples for the data point's info for visible points of the charts only. So the `multimap` of the selected data points might have less elements than `CDataWnd m_mapLabs, or have no entries at all.`.

After receiving the `multimap`, `m_dataWnd`can start to render itself.

The drawing itself is straightforward. First, we need to attach a window to `m_pDataWnd`, if it was not done before. We call `CDataWnd::CreateLegend(CWnd* pParent, CPoint origPnt, bool bData)` to do that. It is a wrapper around the MFC function `CreateEx`.

Obviously, the parent is the chart container. The flag `bool bData` specifies the type of the child: whether it is the data or the names label.

We do not know beforehand the selected points and values of the points' X and Y, nor do we know for all the time the set of visible charts and points. It means the size of the label window is also unknown beforehand. To calculate the label window rectangle, we need the `paintDC` (more correctly, the `Graphics` object). Therefore, `CreateEx` is called on zero x, y, width, and height. After creation, we can get the `Graphics` object from the window's DC, calculate the rectangle and the window position, and move the window to this position. But first, we have to calculate the text rectangle that envelops all strings to be displayed.

We do this iterating over `m_mapLabs` of the `m_pDataWnd`. We search separately for the longest chart name, the longest X value, the longest Y name, and the longest Y value strings using the function `Gdiplus::MeasureString`. Unfortunately, the fonts with the fixed character width are looking not very nice on the screen, so I had to use the font with the variable character width. It means that the MeasureString should be applied to each string, not to the string with the greatest length. It does not matter for the data and names labels because there are only 10-20 charts in the container, but when we have to calculate layout for the data view with the thousands of the data points, there might be a visible delay in displaying the view. The delay is still acceptable for the 1000 - 5000 data points. For the bigger vectors we are displaying the message box "Calculating..." Again, this problem exists for the big data vectors in the data view window only.

The width of the text rectangle is the sum of the maximal widths of the bounding rectangles, returned by `MeasureString`. The height is the height of the bounding rectangle times the size of `CDataWnd::m_mapLabs`. The total width should include additional spacing.

Finally, we have to decide how to place the label in reference to the request point. As a rule, we place the label to the right of the request point if this point is in the left half of the container window, and to the left, if the point is to the right half. If there is not enough space to the left of the request point, we will place the left border of the label close to the left border of the client rectangle. The similar is true for the right borders.

Task: Tracking the data label

The position of the request point and the interval it is centered on are constant in the data space for the lifetime of the given request. The `multimap` of the selected points is also constant. So we do not need to search for the closest points again.

What changes are the position of the request point and the value of the interval in the client space. For example, assume the X-extension of the container is -10.0...10.0, and the X-coordinate of the point of request is 0.0. Then in the client space, this coordinate is mapped to `X = 0.5*clientRect.Width`. Let us zoom in the container to the extent -4.0...1.0. Now 0.0 is mapped to `0.8*clientRect.Width`.

So we need to map the boundaries of the container's client rectangle into the data space, and pass the selected points that fit into this transformed rectangle and are visible to `m_pDataWnd`. Actually, only Y boundaries of the client rectangle should be mapped into the data space. The X boundary are always equal to the container's X extent. We also have to take into account for the local scaleY of every visible chart. To update the data window, we use the function:

```
size_t CChartContainer::UpdateDataLegend(MAP_SELPNTSD& mapSelPntsD, MAP_LABSTR& mapLabStr)
{
  mapLabStr.clear();
  if (!mapSelPntsD.empty()&& in_range(m_startX, m_endX, m_dataLegPntD.X))
  {
    CRect clRect;
    GetClientRect(&clRect);
    CPoint pntLimYL(0, clRect.bottom);
    CPoint pntLimYR(0, clRect.top);
    PointD pntLimYLD, pntLimYRD;
    MousePntToPntD(pntLimYL, pntLimYLD, m_pMatrixD);
    MousePntToPntD(pntLimYR, pntLimYRD, m_pMatrixD);

    MAP_SELPNTSD::iterator itSel = mapSelPntsD.begin();
    MAP_SELPNTSD::iterator itSelE = mapSelPntsD.end();
    while(itSel != itSelE)
    {
      int chartIdx = itSel->first;
      CChart* chartPtr = GetChart(chartIdx);
      if (chartPtr != NULL)
      {
        if (chartPtr->IsChartVisible())
        {
          PointD selPntD = itSel->second;
          if (in_range(m_startX, m_endX, selPntD.X)&&
            in_range(pntLimYLD.Y, pntLimYRD.Y, selPntD.Y*chartPtr->GetLocScaleY()))
          {
            TUPLE_LABEL tuple_res = chartPtr->GetSelValString(
                        selPntD, m_labelX, m_precision, m_pLabValStrFnPtr);
            mapLabStr.insert(MAP_LABSTR::value_type(chartIdx, tuple_res));
          }
        }
        ++itSel;
      }
      else
       itSel = mapSelPntsD.erase(itSel);
    }
  }

  CPoint origPnt(-1, -1); // Not used on empty mapLabStr
  if (!mapLabStr.empty())
  {
    PointF origPntF = m_pMatrixD->TransformToPntF(1.0, m_dataLegPntD);
    origPnt = CPointFromPntF(origPntF);
  }
// Recalc dataWnd window rects and show data wnd or will hide it
  m_pDataWnd->UpdateDataLegend(mapLabStr, this, origPnt);
  return mapLabStr.size();
}
```

This function iterates over `mapSelPntsD`. The map element's key is the chart ID, the value is the selected data point. If the chart is visible and the selected data point is in the client rectangle, the function calls `GetSelValString` for this chart and adds the result to `mapLabs`. Note that the selected

point must be in the client rectangle, not in the `epsX` interval. The interval was used before, in search for neighboring points.

(If `mapSelPntsD` is about being changed, it is cheaper to set `mapSelPntsD` from scratch using `CChartContainer::PrepareDataLegend(PointD origPntD, double epsX, MAP_LABSTR& mapLabels, MAP_SELPNTSD& mapSelPntsD, bool bChangeMatrix)` and `m_dataLegPntD` and `m_epsX`.)

Task: Show chart names

To show chart names, we use the same technique and the same `CDataWnd` class. The container's data member is a pointer to the instance of this class, `m_pLegWnd`. The name string consists of a short line to show color, dash style, and pen width of the chart, and a chart name. The chart names window is a child of the container, and is always located in the upper right corner of the container's window.

Task: Zooming and panning along X-axis (keeping X-history)

The zooming and panning along X-axis themselves are mundane jobs. You just set the container's new X-extension `m_startX`, `m_endX` and ask the container to update its image on the screen. Matter that is more complicated is how to keep history records. We need the history records to undo zooming/panning. We are keeping separate history records for X- and Y- axes. Here we are discussing the X-history.

We are storing the history records as pairs of old `m_startX`, `m_endX` in the vector `m_vHistX`, the `CChartContainer` data member. We just `push_back()` the old pair of `m_startX`,`m_endX` before we set the new `m_startX`, `m_endX`. To undo the action, we will use the saved values to reset `m_startX`, `m_endX`.

Things get more interesting when we change the full X-extent of the container. It might happen when we add charts, append, or truncate the charts' data vectors, delete charts, or simply change the X-extent.

To understand the problem, let us consider the situation when you want to analyze some part of the chart's curve. You have zoomed in the container and are looking at the curve, when, all of sudden, the application decides to append the chunk of data points to some chart. If the container would update its X-extent immediately, the picture you were so busy analyzing will go down the drain. If it would not update, you will lose the new extent.

The full X-extent of the container is always saved in the first element of the history vector. Therefore, the solution to this problem is to update the first element of the vector and not change the current values of `m_startX`, `m_endX`.

The function `CChartContainer::UpdateExtX` does exactly that:

C++

```
void CChartContainer::UpdateExtX(double minExtX, double maxExtX, bool bRedraw)
{
```

```
  if (maxExtX < minExtX)                    // Possible if from app
    return;

  double initStartX = GetInitialStartX();   // Old initial m_startX, m_endX
  double initEndX   = GetInitialEndX();

  double startX, endX;

  if (initStartX > initEndX)  // The container is empty
  {
    startX = minExtX;
    endX   = maxExtX;
  }
  else
  {
    startX = min(minExtX, initStartX);
    endX   = max(maxExtX, initEndX);
  }

  if (startX == endX)
  {
    endX += fabs(startX*0.01);;
  }

  if (m_vHistX.size() > 0)   // Was zoomed or panned
    m_vHistX.front() = make_pair(startX, endX);
  else                       // Has no history
  {
   m_startX = startX;
   m_endX   = endX;
  }

  if (bRedraw)
  {
    if (m_bTracking&& IsLabWndExist(true))
      UpdateDataLegend(false);
    else
      RefreshWnd();
  }
}
```

Pay attention to this piece of the code:

C++

```
if (startX == endX)
{
    endX += fabs(startX*0.01);;
}
```

If we have only charts with one data point each, and these data points have the same X - coordinates, the `startX = endX`. To draw the container, we need some non-zero X - extension. So we artificially set the `endX` 1% apart from the `startX`.

The application should decide how and when to notify the user about the X-extent changes if these changes are hidden by zoom or pan modes.

Task: Zooning and panning along the Y-axis (keeping Y-history)

It turned out that designing and coding the vertical zooming and panning is much more complicated than ones for horizontal zoom/panning .

To begin with, we perceive the horizontal and vertical dimensions of a picture differently. Think about a picture of a family reunion: we would forgive a little cropping of the picture from the left or the right, but we implicitly request and expect some clear space above heads of our relatives.

I took it into consideration: initially chart curves fill only 0.8 of the client rectangle height. A position of this drawing space in the client rectangle depends on the position of the X-axis. It is simple: you just calculate the Y-scale as `0.8*clientRect.Height/Yextent`.

Now enters the vertical zoom. You delineate zoom borders, and you want the picture to fill entire vertical space, entire client rectangle height. So now you have to calculate the Y-scale as `clientRect.Height/Yextent.`

Meanwhile a vertical panning must only shift the drawing space it got from previous operation.

So vertical zooming/panning uses the function:

C++

```
void CChartContainer::UpdateExtY(double minExtY, double maxExtY, bool bRedraw)
{
  if (maxExtY < minExtY)                    // Possible if from app
    return;

  double initMinY = GetInitialMinExtY();    // Old initial m_startX, m_endX
  double initMaxY = GetInitialMaxExtY();

  double startY, endY;

  if (initMinY > initMaxY)  // The container is empty
  {
    startY = minExtY;
    endY   = maxExtY;
  }
  else
  {
    startY = min(minExtY, initMinY);
    endY   = max(maxExtY, initMaxY);
  }

  if (startY == endY)
  {
    double delta = fabs(startY*0.01);
    startY -= delta*4.0;
    endY   += delta;
```

```
  }

  if (m_vHistY.size() > 0)   // Was zoomed or panned
    m_vHistY.front() = make_pair(startY, endY);
  else                       // Has no history
  {
   m_minExtY = startY;
   m_maxExtY = endY;
  }

  if (bRedraw)
  {
    if (m_bTracking&& IsLabWndExist(false))
      UpdateDataLegend(true);
    else
      RefreshWnd();
  }
}
```

but it does the trick with the vertical scale in the function:

C++

```
PAIR_DBLS CChartContainer::UpdateScales(const RectF drawRectF,
                                  double startX, double endX, double minY, double maxY)
{
  if (m_mapCharts.empty())
    return make_pair(1.0, 1.0);

  RectF dRF = drawRectF;
  if ((m_chModeY == MODE_FULLY)||(m_chModeY == MODE_MOVEDY)||(m_chModeY == MODE_MOVEY))
    dRF.Inflate(0.0f, -0.1f*drawRectF.Height); // Reserve 20% to beautify full picture
  double scX = UpdateScaleX(dRF.Width, startX, endX);
  double scY = UpdateScaleY(dRF.Height, minY, maxY);
  return make_pair(scX, scY);
}
```

Again, pay attention to correction for `startY == endY`.

We are not done yet with drawing spaces: we have to tackle problems with undoing vertical zooming/panning. What is the problem? Suppose we have restored previous `m_minExtY, m_maxExtY` from the history vector `m_vHistY`. What vertical drawing space we have to use to calculate the `scaleY`? If we are undoing a chain of actions MoveY1 - ZoomY1 - MoveY2 - ZoomY2 - MoveY3, obviously, up to the ZoomY1 we have to work with full client rectangle height, and return to 0.8H upon undoing ZoomY1. Fortunately, it is easy to pick out moves from zooms: if we pan, values of changes of `startY` and `endY` are equal.

So, there is the function:

```
void CChartContainer::UndoHistStepY(bool bRedraw)
{
```

```
  if (m_vHistY.empty())
    return;

  PAIR_POS zh = m_vHistY.back();
  m_minExtY = zh.first;
  m_maxExtY = zh.second;

  if (m_vHistY.size() > 1)    // Must check whether it is moves only
  {
    auto itZ = adjacent_find(m_vHistY.rbegin(), m_vHistY.rend(),
      [](const PAIR_POS& lhs, const PAIR_POS& rhs) ->bool
          {return (fabs(1.0 - fabs((rhs.first - lhs.first)/(rhs.second - lhs.second))) >
                                                                  4.0*DBL_EPSILON);});
    if (itZ == m_vHistY.rend())
       m_chModeY = MODE_MOVEDY;
    else
      m_chModeY = MODE_ZOOMEDY;
  }
  else
    m_chModeY = MODE_FULLY;

  m_vHistY.pop_back();

  if (bRedraw && IsWindow(m_hWnd) && IsWindowVisible())
  {
    if (m_bTracking && (m_pDataWnd != NULL))
      UpdateDataLegend(true);
    else
      RefreshWnd();
  }
}
```

There we are using `std::adjasent_find` with lambda expression. The expression returns `true` if changes in `minY` and `maxY` are not equal. This algorithm starts from the end of the history vector and returns when it found the `zoomY`. If there are no zooms saved, you have to work with 0.8H drawing space.

See the measure of equality: it is the difference between 1.0 and the ratio of the difference between two adjacent `minY` to the difference between two adjacent `maxY`. The criteria is 4*`DBL_EPSILON`, the smallest such that 1.0+ `DBL_EPSILON` !=1.0. I cannot use the differences alone because of quirks of floating-point arithmetic.

Finally, we have to decide what to do with zooming/panning of empty space. Obviously, it makes no sense to zoom a space without any visible data points, but what about panning? If you are panning along X-axis, there is a chance you are in some valley and will see some data points hidden from view now. But for Y-panning if you do not see any data points in the container's window, you will not see them if you continue in the same direction. So the zooming/panning along Y-axis is blocked, if the new container's extension does not have any visible data point.

Task: Show the chart data

The chart data view displays the data vector of the selected chart as a table. You call the data view for the selected chart from the container's popup menu or programmatically.

It might take many rows to display the entire table, so I choose a page structure to display one page at a time against a choice of scrolling. To save the screen's real estate, I squeeze into one page as many rows and columns as possible.

To navigate between pages and print the data, we need buttons. It would be nice to have buttons with bitmaps, but you cannot embed resource files, external icons, and bitmaps in MFC static libraries (see here). So the data view builds the bitmap buttons at run-time (for the same reason, the container's popup menu is also built at request time, upon mouse right click).

All functionality of the data view is implemented in the class `CChartDataView`. The class is derived from `CWnd`.

In response to the request to display the data view, the container calls:

C++

```
bool CChartContainer::ShowDataView(CChart* chartPtr, bool bClearMap, bool bRefresh)
{
  if (m_pChartDataView == NULL)
    m_pChartDataView = new CChartDataView;

  if (m_pChartDataView != NULL)
  {
    if (!IsWindow(m_pChartDataView->m_hWnd))
    {
      CRect parentWndRect;
      GetParent()->GetWindowRect(&parentWndRect); // App main dlg window

      CRect workRect;
      SystemParametersInfo(SPI_GETWORKAREA, NULL, &workRect, 0);

      int leftX  = parentWndRect.right + DV_SPACE;
      int rightX = leftX + DV_RECTW;
      int topY   = parentWndRect.top - DV_SPACE;
      int bottomY = topY + DV_RECTH;

      CRect dataViewRect(leftX, topY, rightX, bottomY);
      CRect interRect;
      interRect.IntersectRect(&dataViewRect, workRect);
      if (interRect != dataViewRect)
      {
        dataViewRect.right = workRect.right - DV_SPACE;
        dataViewRect.left = max(dataViewRect.right - DV_RECTW, workRect.left + DV_SPACE);
        dataViewRect.top = workRect.top + DV_SPACE;
        dataViewRect.bottom = min(dataViewRect.top + DV_RECTH, workRect.bottom - DV_SPACE);
      }
      BOOL bRes =  m_pChartDataView->CreateEx(0,
                                AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW|CS_SAVEBITS),
                                _T("Chart Data View"),
                                WS_POPUPWINDOW|WS_CAPTION|WS_MINIMIZEBOX|WS_VISIBLE,
```

```
                                dataViewRect.left, dataViewRect.top,
                                dataViewRect.Width(), dataViewRect.Height(),
                                NULL,
                                NULL,
                                NULL);
      if (!bRes)
      {
        delete m_pChartDataView;
        m_pChartDataView = NULL;
        return false;
      }
    }
    else if (m_pChartDataView->IsIconic())
      m_pChartDataView->ShowWindow(SW_RESTORE);

    int chartIdx = chartPtr->GetChartIdx();
    m_pChartDataView->ShowWaitMessage(chartIdx, chartPtr->m_vDataPnts.size());
    m_pChartDataView->InitParams(chartPtr, bClearMap, this);

    if (m_dataViewChartIdx != chartIdx)
    {
      m_dataViewChartIdx = chartIdx;
      bClearMap =true;
    }

    if (bClearMap)
    {
      m_mapDataViewPntsD.clear();
      if (bRefresh)
        RefreshWnd();
    }
  }
  return true;
}
```

The interesting points there are calculation of the view's location, creation of the controls in the data view, and communication between the data view and the container.

I wanted to set the size of the data view window close to letter format, 8.5"x11", to get WYSIWYG printing. However, this format is too big for most monitors. I chose dimensions `DV_RECTW = 710` and `DV_RECTH = 874` pixels. At 96 pixels per inch, it equals 7.4"x9.1".

With the size defined, I try to place the data view rectangle 50 pixels to the right and above the application's main window, which is the parent of the container. Next, I use `SystemParametersInfo` with `SPI_GETWORKAREA` to get the working area of the display. (VS Help: "The work area is the portion of the screen not obscured by the system taskbar or by the application desktop toolbars".) If the intersection of the working area and the newly minted data view rectangle were less than this rectangle, I would move the rectangle to the left and adjust its vertical position. So if there is enough space, the data view window does not overlap the app main window.

The data view window is created as a popup window to allow some leeway in positioning it on the screen.

After view creation, the container calls the function to pass the chart's data to the view.

If the window is already created and at some moment was minimized before a new call to `ShowWindow`, we have a problem: the minimized window has empty window rectangle. It will crash the function `CalcLayout` in `InitParams`. So there is the line in `ShowWindow`:

```
else if (m_pChartDataView->IsIconic())
      m_pChartDataView->ShowWindow(SW_RESTORE);
```

The function `CChartDataView::InitParams` initializes the data view:

```
void CChartDataView::InitParams(const CChart* chartPtr, bool bClearMap, const
CChartContainer* pHost)
{
  m_chartIdx      = chartPtr->GetChartIdx();
  m_precision     = pHost->GetContainerPrecisionX();
  m_precisionY    = chartPtr->GetPrecisionY();
  m_label         = chartPtr->GetChartName();
  string_t tmpStr = pHost->GetAxisXName();
  m_labelX        = tmpStr.empty() ? string_t(_T("X")) : tmpStr;
  tmpStr          = chartPtr->GetAxisYName();
  m_labelY        = tmpStr.empty() ? string_t(_T("Y")) : tmpStr;
  m_pXLabelStrFn  = pHost->GetLabXValStrFnPtr();
  m_pYLabelStrFn  = chartPtr->GetLabYValStrFnPtr();
  m_vDataPnts     = chartPtr->m_vDataPnts;

  m_vStrX.resize(m_vDataPnts.size());
  transform(m_vDataPnts.begin(), m_vDataPnts.end(), m_vStrX.begin(),
                  nmb_to_string<double, false>(m_precision, m_pXLabelStrFn));
  m_vStrY.resize(m_vDataPnts.size());
  transform(m_vDataPnts.begin(), m_vDataPnts.end(), m_vStrY.begin(),
                  nmb_to_string<double, true>(m_precisionY, m_pYLabelStrFn));
  m_currPageID = 0;

  SetOwner((CWnd*)pHost);
  m_vRows.clear();
  if (bClearMap)
    m_mapSelCells.clear();
  else
    UpdateDataIdx();

  CalcLayout();
  m_header = GetTableHeader();   // Set the header string

  CreateChildren();

  bool bEnableLeft = m_currPageID == 0 ? false : true;
  bool bEnableRight = m_nPages == 1 ? false : true;

  m_leftEnd.EnableWindow(bEnableLeft ? TRUE:FALSE);
  m_leftArr.EnableWindow(bEnableLeft ? TRUE:FALSE);
```

```
  m_rightArr.EnableWindow(bEnableRight ? TRUE:FALSE);
  m_rightEnd.EnableWindow(bEnableRight ? TRUE:FALSE);

  if (IsWindow(m_hWnd)&&IsWindowVisible())
    RedrawWindow(NULL, NULL, RDW_INVALIDATE |
                 RDW_UPDATENOW | RDW_NOERASE|RDW_ALLCHILDREN);
}
```

The container is set as an owner of the data view, so the view will automatically hide, set visible, and close with the container.

To accelerate the drawing, we provide two auxiliary vectors of strings, `m_vStrX` for the X values, and `m_vStrY` for the Y values of the chart data vector. We use the algorithm `std::transform` with the custom-made predicate template `<typename T, bool bY> struct nmb_to_string` (see *Util.h*).

The navigation buttons are instances of the class `CPageCtrl : piblic CButton`. The buttons are created as children of `m_pDataView`. The drawing of the buttons' bitmaps is embedded into `CPageCtrl::OnPaint` (see *DataView.cpp* for details).

Now let us go to communication between the data view and the container. We need to inform the container when we select/deselect a cell in the table. The container then will show/hide the data point, selected in the data view, on the chart's curve in the container window. In addition, the data view needs info to modify the data view if the container name, the chart's data vector, or/and X- and Y-axes names, precision, or/and formatting functions are changed in the container.

The data view has a copy of the data vector of the chart it displays, `CDataView::m_vDataPnts`.

It also keeps the data points for all selected cells in the map `CDataView::m_mapSelCells`. The key of the map element is the cell's ID. The data view updates the map when the selection changes.

The container has the copy of this map in `CChartContainer::m_mapDataViewPntsD`. After change of the selection in the data view, the data view calls the container's function:

```
CChartContainer* pContainer = static_cast<CChartContainer*>(GetOwner());
pContainer->UpdateDataViewPnts(m_chartIdx, dataID, dataPntD, bAdd)
```

The container uses `m_mapDataViewPntsD` to draw circles around data points selected in the data view data points. It enables you to see exactly where a particular point sits on the chart's curve.

Obviously, changes in chart attributes such as the name of Y-values, the Y-precision, and the Y-formatting function might change the data view layout. The same is true for the name of X-values and X-formatting function, and changes of the chart's data vector (e.g. appended or truncated). Changes of the chart and container names influence page headers only. We found more convenient to recalculate only affected parts of the layout. To do this we use the function `CChartContainer::UpdateDataView`. This function calls `CDataView::UpdateParams`:

```
bool CChartDataView::UpdateParams(const CChart* chartPtr, int flagsData)
{
  bool bRes = false;
  int flags = 0;
  size_t dataOffset = 0;

  int chartIdx = chartPtr->GetChartIdx();
  if (chartIdx == m_chartIdx)
  {
    CChartContainer* pHost = dynamic_cast<CChartContainer*>(GetOwner());
    ENSURE(pHost != NULL);
    if (!chartPtr->HasData())
    {
      pHost->DestroyChartDataView();
      return true;
    }

    m_label = chartPtr->GetChartName();    // Page header must be changed with new date
stamp

    int precisionX = pHost->GetContainerPrecisionX();
    if (m_precision != precisionX)        // PrecisionX: entire X column is changing
    {
      m_precision = precisionX;
      flags |= F_VALX;
    }

    int precisionY = chartPtr->GetPrecisionY();
    if (m_precisionY != precisionY)       // PrecisionY: entire X column is changing
    {
      m_precisionY = precisionY;
      flags |= F_VALY;
    }

    string_t tmpStr = pHost->GetAxisXName();
    string_t labelX = tmpStr.empty() ? string_t(_T("X")) : tmpStr;
    if (m_labelX != labelX)               // X-axis name: column header and width might
change
    {
      m_labelX = labelX;
      flags |= F_NAMEX;
    }

    tmpStr          = chartPtr->GetAxisYName();
    string_t labelY = tmpStr.empty() ? string_t(_T("Y")) : tmpStr;
    if (m_labelY != labelY)               // Y-axis name: column header and width might
change
    {
      m_labelY = labelY;
      flags |= F_NAMEY;
    }

    val_label_str_fn pXLabelStrFn  = pHost->GetLabXValStrFnPtr();
    if (m_pXLabelStrFn != pXLabelStrFn)   // Entire X-column should be changed
    {
```

```
  m_pXLabelStrFn = pXLabelStrFn;
  flags |= F_VALX;
}

val_label_str_fn pYLabelStrFn = chartPtr->GetLabYValStrFnPtr();
if (m_pYLabelStrFn != pYLabelStrFn)   // Entire Y-column should be changed
{
  m_pYLabelStrFn = pYLabelStrFn;
  flags |= F_VALY;
}

if (flagsData != F_NODATACHANGE)
{
  size_t endOffs = 0;
  switch (flagsData)
  {
  case F_APPEND:
    endOffs = OnChartAppended(chartPtr->m_vDataPnts);
    if (!(flags & (F_VALX|F_VALY|F_DSIZE)))
    {
      dataOffset = endOffs;
    }
    flags |= (F_VALX|F_VALY|F_DSIZE);
    break;
  case F_TRUNCATE:
    endOffs = OnChartTruncated(chartPtr->m_vDataPnts);
    if (!(flags & (F_VALX|F_VALY|F_DSIZE)))
    {
      dataOffset = 0;
    }
    flags |= (F_VALX|F_VALY|F_DSIZE);
    break;
  case F_REPLACE:
  case F_REPLACE|F_HASCELLSMAP:
   dataOffset = OnChartDataReplaced(
        chartPtr->m_vDataPnts, flags&F_HASCELLSMAP ? true : false);
   flags |= (F_VALX|F_VALY|F_DSIZE);
   break;
  }
}
else
{
  if (flags & F_VALX)
  {
    transform(m_vDataPnts.begin() + dataOffset, m_vDataPnts.end(),
          m_vStrX.begin() + dataOffset, nmb_to_string<double,
          false>(m_precision, m_pXLabelStrFn));
  }

  if (flags & F_VALY)
  {
    transform(m_vDataPnts.begin() + dataOffset, m_vDataPnts.end(),
          m_vStrY.begin() + dataOffset, nmb_to_string<double,
          true>(m_precisionY, m_pYLabelStrFn));
  }
}
```

```
    if (IsIconic())
      ShowWindow(SW_RESTORE);

// Set the header string
    m_header = GetTableHeader();

    m_vRows.clear();
    if ((flags != 0)&&(dataOffset != m_vDataPnts.size()))
      CalcLayout(flags, dataOffset);

    if (flagsData != F_NODATACHANGE)
    {
      if (flagsData & F_TRUNCATE)
      {
        if (m_nPages <= m_currPageID)
          m_currPageID = 0;
      }
      else if ((flagsData & F_APPEND) == 0)
        m_currPageID = 0;
    }
    else
      m_currPageID = 0;


    bool bEnableLeft = m_currPageID == 0 ? false : true;
    bool bEnableRight = (m_currPageID == (m_nPages - 1)) ? false : true;

    m_leftEnd.EnableWindow(bEnableLeft ? TRUE:FALSE);
    m_leftArr.EnableWindow(bEnableLeft ? TRUE:FALSE);

    m_rightArr.EnableWindow(bEnableRight ? TRUE:FALSE);
    m_rightEnd.EnableWindow(bEnableRight ? TRUE:FALSE);

    if (IsWindow(m_hWnd)&&IsWindowVisible())
      RedrawWindow(NULL, NULL, RDW_INVALIDATE | RDW_UPDATENOW |
                   RDW_NOERASE|RDW_ALLCHILDREN);
    bRes = true;
  }
  return bRes;
}
```

This function looks for the changed chart's attributes and sets appropriate flags. The flags control the tasks to be performed by the data view to reflect the changes. The `flagsData` that are passed to the function as a parameter are informing the function about changes in the chart data vector. This information is used to calculate the page the data view will show after the data view is updated. It might be the old page if the old page still keeps some data points, or the first page if the old page was truncated. I refer you to the *ChartDataView.cpp* for farther details.

Task: Printing

You can print the container window from the container's popup menu or programmatically. You can also print the chart data tables from the data view window.

Let us begin with the container.

First, let me say that what we are going to print is not WYSIWYG. If the user decides to print only one chart, we will print only one selected chart. Otherwise, we will print all visible charts. Second, on the screen, to get to details, we always can move charts, zoom in, hide the data and name labels, etc. The printout is forever. So to not obscure chart curves, we will not show the data and names windows. Instead, we will print the chart info below the container window. To make measurements and calculations with the printout possible, we will include the Y-scale value in the chart info, and always print the X-axis labels. Third, the body of the printing is implemented as a static function `CChartContainer::PrintCharts`. We did so to allow printing from a working thread.

In the article KB133275, Microsoft explains how to print from a class other than an MFC `CView`. In addition, there is a tutorial on GDI Printing, GDI+ Printing on Internet. The tutorial is overcomplicated; Microsoft does not mention GDI+.

Still, I followed the framework of the Microsoft sample code.

The code for printing is in the function `CChartContainer::PrintCharts(CChartContaner* pContainer, float dpiRatioX, HDC printDC)` (*ChartContainer.cpp*).

The application must prepare parameters and pass them to the function. The code should look like:

```
.............................................
   int scrDpiX = GetScreenDpi();
   SendNotification(CODE_PRINTING);
   CChartContainer* pContainer = CloneChartContainer(string_t(_T("")), true);
   SendNotification(CODE_PRINTED);
   PrintCharts(pContainer, scrDpiX, printDlg.GetPrinterDC());
   delete pContainer;
```

First, we clone the container. The clone inherits the name and the state of the ancestor. No window is attached to the clone: we do not need it. Before and after cloning we send notifications to the container's parent. It can use them in multithreading environment or might ignore them altogether.

We use the clone because the size of the printing area (page) is different from the size of the ancestor's client rectangle. Our drawing functions use the transform matrix of the container, so we have to recalculate the container's transform matrix for printing. We also are going to change the state of the clone to allow printing the X-axis labels.

Second, we calculate the ancestor's screen resolution in dots per inch, calling:

```
int CChartContainer::GetScreenDpi(void)
{
    CPaintDC containerDC(this);
    int scrDpiX = containerDC.GetDeviceCaps(LOGPIXELSX);
```

```
    int scrDpiY = containerDC.GetDeviceCaps(LOGPIXELSY);
    ENSURE(scrDpiX == scrDpiY);
    return scrDpiX;
}
```

I will explain why we need it for printing in a moment.

Third, we need the printer DC.

If we started with the MFC dialog `CPrintDialog`, after the printer is selected, and the OK button is clicked, the handle to the printer DC is:

```
HDC printDC = printDlg.GetPrinterDC();
```

Now we can call `PrintCharts`.

We get the pointer to the CDC and attach the `printerDC` to it, following KB133275:

```
CDC* pDC = new CDC;
pDC->Attach(printDC);
```

We create a `Gdiplus::Graphics` object and set the document units:

C++

```
Graphics* grPtr = new Graphics(printDC);
grPtr->SetPageUnit(UnitDocument);
```

This mode displays 300 DPI per inch.

After page units are set, all GDI+ functions will understand any value passed to them as the `UnitDocument` value. For example, if we are setting a pen's width to two, it is inch on the screen and inch on the paper. So we have to correct values of all literals used in printing.

We use the `scrDpiX` parameter to get `dpiRatioX = 300.0f/scrDpiX`. For the screen, this ratio is 1.0, so if we need to adjust the pen width for printing, we should write `pen.SetWIdth(width*dpiRatio)`.

And the last preparation job: get the client rectangle:

```
RectF rGdiF;
grPtr->GetVisibleClipBounds(&rGdiF);
// The same as the clip rect
```

Finally, begin printing:

```
pDC->StartDoc(pContainer->m_name.c_str());
// MFC functions

pDC->StartPage();
```

I have mentioned earlier that the chart info strings are printed below the container window. The chart info consists of the chart name, vertical scale for this chart in data space (Y units per screen inch), X-axis name, X value string, Y-axis names, and Y value strings of the data points displayed in the data label in the ancestor. If there is no selected points, or the selected point is out of view (as a result of zooming/panning) instead of the X and Y names and value strings we print charts' minimal and maximal Y values. The short line before the info string has the same color, dash style, and pen width as the chart has. It helps to identify the charts easily.

If there are too many charts in the container, the chart info lines might continue to the next page. Every page has a header: a name of the container, and a time when the printing started.

In drawing functions for the printing headers and chart info, we use the points unit to set the font size. Because we have set the page unit to `UnitDocument`, the font size is the same, no matter what the printer resolution is.

After we have done with the printing, we should clean up:

```
// End printing
pDC->EndPage();
pDC->EndDoc();

delete grPtr;
pDC->Detach();
delete pDC;
```

Note: Sometimes you might get circles around the data points that are not visible on the screen because the printed page size is greater than the container client window. If you do not need them, hide them using the container popup menu or call the function `CChartContainer::ShowChartPoints(int chartIdx, bool bShow, bool bRedraw)` with `chartIdx = -1` and `bShow = false` before printing. After printing restore the `ShowChartPoints` state.

The printing of the data view is similar.

Task: Saving the chart data

You can save the chart's data vector. You also can save the selected or all visible charts or all chatrts together with their visual attributes and data series into an XML file.

To get the chart data vector, you call one of the overloads of the function:

```
CChartContainer::ExportChartData(string_t chartName, V_CHARTDATAD& vDataPnts);
```

The overloads substitute `std::vector<std::pair<double, double> >` or a pair of vectors `std::vector<double>& vX`, `std::vector<double>& vY` instead of a vector of data points `V_CHARTDATAD& vDataPnts`. Because the chart ID is an internal parameter of the chart control, we select the container's chart by name.

To save the data into an XML file, we use the function `HRESULT CChartContainer::SaveChartData(string_t pathName, bool bAll)` .

Actually, all functionality related to the conversion to and from XML files is placed in the class `CChartsXMLSerializer`.

The static function `CChartsXMLSerializer::ChartDataToXML(pathName.c_str(), pContainer, chartName, bAll)` converts chart attributes and data vectors to XML. `SaveChartData` provides parameters for this function.

First of all, it processes the `pathName`. If the name is an empty string, the user is presented with the MFC `CFileDialog` to set the path and the XML file name.

Second, the `SaveCharts` clones the container to make multithreading possible. Before and after cloning the function sends notification the the container's parent. We clone the container because conversion to XML could take long time for big data vectors. XML converter is working with the clone, not with the container itself.

Finally, the function looks at the container charts to pass the chart names to the serializer. The parameter `bAll` tells what charts are to be saved. If `bAll = true`, the chart names must be the names of the visible charts. If the chart is not visible, nothing will be saved. If `bAll = false`, the chart names might be names of any charts, visible or not. If the `chartName` is an empty string, all visible charts or all container's chart will be saved, depending on the parameter `bA<code>ll`; it the name of the existing chart is passed, only this chart will be saved. The `SaveChart` looks for a selected chart. If there is one, its name is passed to the converter function and only it will be saved. On this occasion `bAll` make no difference, because only visible chart can be selected.

You can call the `SaveChartData` from the container's popup menu or directly from your application. The popup menu automatically calls the function with empty `pathName` and `bAll = false`. Your application can use any combination of these parameters. The empty `pathName` means that the user will be presenter with an instance of the `MFC CFileDialog` to choice the path and file name.

Remember, `bAll = true` means that all charts in the container will saved if there is no selected chart; `bAll = false` means only visible chart(s) are for saving.

The user or the programmer can control the selection of charts indirectly, by selecting the chart to save or make visible all charts he wants to save.

I used MSXML6 to do the job. The structure of the XML file is shown above. Note that the XML schema in the version 1.1 is changed. The container cannot load XML file saved in the version 1.0 into version 1.1.

Task: Load the XML file

To load charts from an XML file, you have to call the function:

C++

```
HRESULT CChartContainer::LoadCharts(LPCTSTR fileName, const MAP_CHARTCOLS& mapContent)
{
    HRESULT hr = CChartsXMLSerializer::XMLToCharts(fileName, this, mapContent);

    if (hr == S_OK)
    {
       if (IsLabWndExist(false))
         PrepareDataLegend(m_dataLegPntD, m_epsX,
                           m_pDataWnd->m_mapLabs,
                           m_mapSelPntsD, NULL);

       UpdateContainerWnds();
    }
    return hr;
```

It calls:

C++

```
HRESULTULT CChartsXMLSerializer::XMLToCharts(LPCTSTR fileName,
           CChartContainer* pContainer, const MAP_CHARTCOLS& mapContent)
```

`XMLToCharts` reads the XML file using MSXML6, and adds the chart(s) to the container.

We have a couple of problems here. First, the XML file might keep several charts, but we might not want to load all of them. Second, the names and visuals of the charts being loaded might mix up with the names and visuals of the charts already in the container. All we know from outset is the name of the XML file.

To get more information about charts in the file, use the functions:

```
HRESULT CChartContainer::GetChartNamesFromXMLFile(LPCTSTR fileName, MAP_CHARTCOLS&
mapContent)
HRESULT CChartContainer::GetChartNamesFromXMLFile(LPCTSTR fileName, MAP_NAMES& mapNames)
```

The last function was introduced in the version 1.1.

The functions are wrappers around the functions from `CChartsXMLSerializer` with the same names and signatures. The wrapping spares you from including an additional header, *ChartXMLSerializer.h*, in your project.

The firs function, `GetChartNamesFromXMLFile`, retrieves the chart names and colors from the file, and stores them in the `MAP_CHRTCOLS`. Map keys are the chart names, values are chart colors. Given the map, you can erase the unwanted charts from the map, and change the colors of the charts you decided to load into the container. After the map is adjusted, you pass it to `LoadCharts`. Of course, you can fill the map manually if you know the chart's names and colors. You also can change the chart colors after loading the charts. The chart name could be automatically changed inside `LoadCharts` if the container already has the chart with the same name. The name in the XML file will not change.

The second function retrieves names: the chart names, the names of the X- and Y-axes, and the samples of the formatted X and Y value strings. It gives you opportunity to write and include in your application appropriate formatting functions and register them with your chart container.

If you are loading charts into an already populated container, and the container is in tracking mode, you have to update the data and name labels. It turns out that it is computationally cheaper to prepare the new data legend from scratch, than to change the existing map of the selected data points.

The function `UpdateContainerWnds` resets the container window and labels to their current state.

If you want just replace the container's charts with the charts from an XML file, just use the function `HRESULT CChartContainer::ReplaceContainerCharts(LPCTSTR fileName)`. No problems with colors, names, etc.

Task: Saving chart as image

The general schema of things is very simple: produce a bitmap with charts drawn into it, and save the bitmap in any picture format your OS supports using `Gdiplus::Save(const WCHAR* filename, const CLSID* clsidEncoder, const EncoderParameters* encoderParams)`. As I have mentioned above, all drawing in the container's function `OnPaint()` is done into memory bitmap first to avoid flickering, so this part of the task can be done using the code from `OnPaint()`. Nevertheless, there is a problem: the name and data labels are displayed as the container children. The child windows have their own `OnPaint()` and will not be shown in the parent's bitmap. The solution is to use the code from the childs bitmaps. Draw the children into the main bitmap, but carefully position the children layout rectangles on the main bitmap. To see details, look into `CChartContainer::DrawContainerToBmp(Graphics*rGdi, Bitmap& bmp)` in *ChartContainer.cpp*.

I think enumeration of the supported picture formats is also interesting. Here is the code (before `if (pathName.empty())`:

C++

```
Status CChartContainer::SaveContainerImage(string_t pathName)
{
  if (!HasChartWithData(-1,true))   // Repeat to provide for standalone use
    return GenericError;

  Status status = Aborted;
  UINT  num;        // number of image encoders
  UINT  size;       // size, in bytes, of the image encoder array

// How many encoders are there? How big (in bytes) is the array of all ImageCodecInfo
objects?
  GetImageEncodersSize(&num, &size);
// Create a buffer large enough to hold the array of ImageCodecInfo objects
// that will be returned by GetImageEncoders.
  ImageCodecInfo* pImageCodecInfo = (ImageCodecInfo*)(malloc(size));;
// GetImageEncoders creates an array of ImageCodecInfo objects
// and copies that array into a previously allocated buffer.
  GetImageEncoders(num, size, pImageCodecInfo);
// Get filter string
  sstream_t stream_t;
  string_t str_t, tmp_t;
  string_t szFilter;
  CLSID clsID;
  typedef std::map<string_t, CLSID> MAP_CLSID;
  typedef MAP_CLSID::value_type TYPE_VALCLSID;
  typedef MAP_CLSID::iterator IT_CLSID;

  MAP_CLSID mapCLSID;

  for(UINT j = 0; j < num; ++j)
  {
    stream_t << pImageCodecInfo[j].MimeType <<_T("\n");
    getline(stream_t, str_t);
    size_t delPos = str_t.find(TCHAR('/'), 0);
    str_t.erase(0, delPos + 1);
    clsID = pImageCodecInfo[j].Clsid;
    mapCLSID.insert(TYPE_VALCLSID(str_t, clsID));
    tmp_t = str_t;
    std::transform(tmp_t.begin(), tmp_t.end(), tmp_t.begin(),
         [](const TCHAR&tch) ->TCHAR {return (TCHAR)toupper(tch);});
    szFilter += tmp_t + string_t(_T(" File|*.")) + str_t + string_t(_T("|"));
  }
  szFilter += string_t(_T("|"));
  free(pImageCodecInfo);

  if (pathName.empty())     // Let the user choose
  {
    TCHAR szWorkDirPath[255];
    GetModuleFileName(NULL, szWorkDirPath, 255);
    PathRemoveFileSpec(szWorkDirPath);
```

```
    string_t dirStr(szWorkDirPath);
    size_t lastSlash = dirStr.find_last_of(_T("\\")) + 1;
    dirStr.erase(lastSlash, dirStr.size() - lastSlash);
    dirStr += string_t(_T("Images"));
    szFilter += string_t(_T("|"));
    CFileDialog fileDlg(FALSE, _T("BMP File"), _T("*.bmp"),
        OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT|OFN_NOCHANGEDIR|OFN_EXPLORER,
        szFilter.c_str(), this);

    fileDlg.m_ofn.lpstrInitialDir = dirStr.c_str();
    fileDlg.m_ofn.lpstrTitle = _T("Save As Image");

    string_t strTitle(_T("Save "));

    if (fileDlg.DoModal() == IDOK)
    {
      pathName = string_t(fileDlg.GetPathName());
    }
    else
      return Ok;
  }

  if (pathName.empty())
    return InvalidParameter;

  size_t pos = pathName.find(_T("."));
  if (pos == string_t::npos)
    return GenericError;

  string_t szExt = pathName.substr(pos);
  pos = szFilter.find(szExt);
  if (pos == string_t::npos)
    return UnknownImageFormat;
  szExt.erase(0, 1);

  IT_CLSID it = mapCLSID.find(szExt);
  if (it != mapCLSID.end())
  {
    SendNotification(CODE_SAVEIMAGE);
    CRect clR;
    GetClientRect(&clR);
    Rect rGdi = CRectToGdiRect(clR);
    Bitmap bmp(rGdi.Width, rGdi.Height);
    DrawContainerToBmp(rGdi, bmp);

    clsID = it->second;
    status = bmp.Save(pathName.c_str(), &clsID);
    SendNotification(CODE_SAVEDIMAGE);
  }
  else status = UnknownImageFormat;
  return status;
}
```

Like saving charts to XML file, the function receives path to a file to save the image. The path might be an empty string; it it is the case, the `CFileDialog` is displayed to ask the user to select the path and the file name. I had problems using `_tupr` functions, so I took the `transform` algorithm with a lambda

expression to convert the picture format string to upper case. Of course, only what is visible in the container's window is saved.

Task: Changing the container size

When the size of the container's parent window is changing, the container's window might not receive the `WM_SIZE` message. E.g., it happens in the dialog-based application.

You have to call `CChartContainer::OnChangedSize(int cx, int cy)` from the appropriate handler of the parent or the owner of the container to change the container window size (see the "Clone Container" in the demo).

There is a little trick in `OnChangedSize`. When the parent window is changing its size, it is not a continuous process: here and there, the user unintentionally interrupts the smooth movement of the mouse. If the data and/or name labels are displayed, they will flicker. So upon every call, `OnChangedSize` hides the visible labels, redraws the container, and restarts the timer. The timer's delay is 50 ms, small enough to be not an annoyance, but big enough to keep the timer running between `WM_SIZE` interrupts. Finally, 50 ms after the sizing ends, the timer procedure redraws the labels on the screen.

In my practice, sometimes I was using the ChartCtrl (a clone of it) without a parent, as a poppup window with a resizable borders. On this occasion, the clone was receiving and must process `WM_SIZE` messages. So now ths chart container has it own `OnSize` handler. But remember, this handler is not called if the chart container is a child of the `CDialog` parent.

Application Programmer Interface: Charts interface

All API functions are in the *ChartContainer.h* file. I am going to mention only the most important of them.

First, there are chart interface functions: `AddChart`, `AppendChartData`, `ReplaceChartData`, `TruncateChart`, and `RemoveChart`.

I have discussed `AddChart`, `AppendChartData`, and `TruncateChart` in the chapter "Add charts to the chart container" above.

The function:

C++

```
bool ReplaceChartData(int chartIdx, V_CHARTDATAD& vData, bool bClip = false,
                      bool bUpdate = false,bool bVerbose = false, bool bRedraw = false);
```

replaces the old chart data with the new data vector. The parameter `bUpdate` defines whether the container X- and Y- extensions should be recalculates. If `bClip == true`, only data points inside the

chart's old X-range are copied from `vData`. If `bVerbose == true`, the warning about loss of old data points is displayed. The function returns `true` on success.

There are three overloads for the three other types of data: the time series, the vector of `std::pair<double, double>`, and for two vectors of X and Y values. For the time series overload you should provide the start X coordinate and X step.

The chart data vector could be replaced with empty vector. It makes possible to start anew with the same chart visuals.

The function:

```
bool RemoveChart(int chartIdx, bool bCorrectMinMax, bool bRedraw)
```

does just that: removes the chart from the container. If `bCorrectMinMax == true`, the container calculates and sets new `m_startX` and `m_endX` boundaries of the X-axis. The function returns `true` on success.

Application Programmer Interface: Access to chart attributes

Access to chart data members is permitted only via container member functions. Mostly, you have to pass to these functions the chart's ID. The charts are known outside the container by their names. To get the chart ID, you should use the container member function:

```
int GetChartIdx(string_t chartName)
```

or store and remember the value returned by `AddChart`.

Here `string_t` is an alias of `std::basic_string<TCHAR>`.

The chart ID cannot be a negative value. The ID value -1 has a special meaning: if it is returned by a "`Get`" function, it means failure (e.g., chart does not exist). When it is passed to a "`Set`" function, it means "All charts in this container" or "All visible charts".

The functions that change the appearance of the container often have a parameter `bool bRedraw`. If it is set to true, the container will be redrawn.

Unfortunately, there are too many member functions in the container to discuss all of them. Please look at *ChartContainer.h* and *ChartContainer.cpp*.

Application Programmer Interface: Notifications

As I mentioned before, the container sends notifications to its parent when the user changes the container's X-extension or chart's "Show/Hide data points" flag or/and visibility from the popup menu. It makes possible for the parent to react to these actions. The notification is a standard MFC/Win32 process. The container sends `WM_NOTIFY` message to the parent:

```
LRESULT CChartContainer::SendNotification(UINT code, int chartIdx)
{
  NMCHART nmchart;
  nmchart.hdr.hwndFrom = m_hWnd;
  nmchart.hdr.idFrom = GetDlgCtrlID();
  nmchart.hdr.code = code;
  nmchart.chartIdx = chartIdx;

  switch (code)
  {
  case CODE_VISIBILITY: nmchart.bState = IsChartVisible(chartIdx);            break;
  case CODE_SHOWPNTS:   nmchart.bState = AreChartPntsAllowed(chartIdx).first; break;
  case CODE_EXTX:
  case CODE_EXTY:
                        nmchart.minX   = GetStartX();
                        nmchart.maxX   = GetEndX();
                        nmchart.minY   = GetMinY();
                        nmchart.maxY   = GetMaxY();
                        break;
  case CODE_REFRESH:    nmchart.minX   = GetInitialStartX();
                        nmchart.maxX   = GetInitialEndX();
                        nmchart.minY   = GetInitialMinExtY();
                        nmchart.maxY   = GetInitialMaxExtY();
                        break;
  case CODE_SAVEIMAGE:
  case CODE_SAVEDIMAGE:
  case CODE_SAVECHARTS:
  case CODE_SAVEDCHARTS:
  case CODE_PRINTING:
  case CODE_PRINTED:
  case CODE_SCY:        break;
  case CODE_TRACKING:   nmchart.bState = m_bTracking;
                        break;
  default:              return 0;
  }

  CWnd* parentPtr = (CWnd*)GetParent();
  if (parentPtr != NULL)
    return parentPtr->SendMessage(WM_NOTIFY, WPARAM(nmchart.hdr.hwndFrom),
LPARAM(&nmchart));
  return 0;
}
```

The notification codes and extension of the `NMHDR` structure are defined in *ChartDef.h*:

```
typedef struct tagNMCHART
{
  NMHDR hdr;
  int chartIdx;
  bool bState;
  double minX;
  double maxX;
  double minY;
  double maxY;
} NMCHART, *PNMCHART;


// Codes: Toggle Visibility
#define CODE_VISIBILITY  1U
// Show points
#define CODE_SHOWPNTS    2U
// Ext X was changed
#define CODE_EXTX        3U
#define CODE_EXTY        4u
#define CODE_REFRESH     5U
// Save (to use in multithreading)
#define CODE_SAVEIMAGE   6U
#define CODE_SAVEDIMAGE  7U
#define CODE_SAVECHARTS  8U
#define CODE_SAVEDCHARTS 9U
// Printing
#define CODE_PRINTING   10U
#define CODE_PRINTED    11U
// Scale change
#define CODE_SCY        12U
// For enabling tracking from popup menu
#define CODE_TRACKING   14U
```

As usual, the parent should implement the MFC handler for notification. For example:

```
BEGIN_MESSAGE_MAP(CChartCtrlDemoDlg, CDialogEx)
  ..............................
  ON_NOTIFY(CODE_VISIBILITY, IDC_STCHARTCONTAINER, OnChartVisibilityChanged)
  ..............................
END_MESSAGE_MAP()

afx_msg void OnChartVisibilityChanged(NMHDR*, LRESULT*);
```

The version information

In an application (.exe), all version information is in the version resource in the .rc file. As we already know, resource files cannot be included in the static library file. So I used stand-ins: according to MS recommendations I placed at the beginning of *ChartDef.h* definitions:

```
// Version
//

#define FILEVER            2,0,1,1
#define PRODUCTVER         2,0,1,1
#define STRFILEVER         _T("2.0.1.1")
#define STRPRODUCTVER      _T("2.0.1.1")
#define STRCOMPNAME        _T("geoyar")
#define STRFILEDESCRIPTION _T("ChartCtrlLib")
#define STRFILENAME        _T("ChartCtrlLib.lib")
#define STRPRODNAME        _T("ChartCtrlLib")In the same file, I have defined access
functions:
```

```
// Retrieve version info
inline string_t GetLibFileVersion(void){return string_t(STRFILEVER);}
inline string_t GetLibProductVersion(void) {return string_t(STRPRODUCTVER);}
inline string_t GetLibCompName(void) {return string_t(STRCOMPNAME);}
inline string_t GetLibFileDescr(void) {return string_t(STRFILEDESCRIPTION);}
inline string_t GetLibFileName(void) {return string_t(STRFILENAME );}
inline string_t GetLibProdName(void) {return string_t(STRPRODNAME);}
```

So at first glance on *ChartDef.h* you will get information about the version of the *ChartCtrlLib.lib*. Your application can use the version access functions to do the same.

The demo application

It is a dialog-based application. The chart container is a control in the main application dialog.

All controls to manipulate the container (add/change/remove/append/truncate/delete/load from an XML file) are in a tab control to the right of the container window. The tabs of the control are shown below.

The application has data generators to generate the sinusoid, sin(x)/x, exponent, rectangle wave (multi-valued function), and random data series.

.

Tab 1 is the "Add Chart" tab. It is the default tab. You will see it first when you start the demo.

The "Add Chart" tab has edit boxes to enter the chart name and/or Y value name, controls to set visual attributes, chart's X-extent, and a number of data points in the data series. There is a slide to set Y-precision. The Y-multiplier slider sets the order of magnitude of the Y-coordinates of the data series. E.g., if the slider is set to -2, all Y-coordinates will be multiplied by 10^{-2}. If you do not enter the chart name, the application would generate the names for you.

Tab 0 is the "Container Properties" tab.

The control group "Set Colors" is enabled only if the container is empty. The "Precision" and "Set Range X" sliders and "X-Axis Name" edit box are enabled only if the container has at least one chart in it. Of course, in real life, you could change the colors of the container elements at any time. However, in the demo, I decided to allow the change of the colors only on an empty container to make it easy to set the right chart colors later when you already know the container colors. The X-axis name edit control allows changing the X-axis name. The name is the same for all charts in the container. The default name is "X".

When you are on this tab, the user input to the container is blocked: you can zoom/pan, invoke the popup menu, etc. I did it to make possible to undo the changes you have applied when you clicked on the button "Apply". You can undo one step or go to the state you had when you have open the tab.

The user input is unblocked when you quit this tab.

Tab 2 is for changing attributes of charts already in the container. You select the chart from the list box and, using tab controls, set the names of chart and its Y-values, Y-precision, and visuals: color, dash style, the pen width, and tension. You can undo all changes you did by selecting the chart in the list box and clicking the button "Undo" while you are staying on this tab. Switch to any other tab will clear the change history. The chart name must be unique for the given session. The chart and Y-name must have less than 28 characters. If you would violate these rules, the container "Set" function will truncate the entered names or/and add suffixes to them.

Tab 3 is the "Append Chart" tab. The list box control is shown for info only. For this demo, you cannot select one or several charts to append; all charts in the container will be appended. There is the checkbox "Animate". If it is checked, the "Append" command imitates an oscilloscope (sort of). You can discard the changes to the container with "Undo Append".

Tab 4 is the "Truncate Chart" tab. Select the chart, the start and end X coordinates, and truncate the chart. If the checkbox "Recalc Scales" is checked, the container will be forced to recalculate its X- and Y-scales to shrink to the new maximal ranges of its X and Y extensions. The checkbox "Keep Range X" can be activated only after the first successful truncation. If it is checked, it will lock the new X-extension to truncate all other charts to the same X-extension. You can select the chart and restore it to the initial state using the button "Undo".

Tab 5 is the "Remove Chart" tab. Just select and remove (delete) the chart. Again, "Recalculate Scales" updates the container's X- and Y-extents.

Tab 6 is the "Clone/Load" tab.

The "Clone" button copies the container in the new popup window with the resizable border. The owner of that window is the source container in the main application dialog.

The controls in the "Load from XML file" group are doing just that: they help to select and load chart(s) from the XML file. When you select the file, the chart names from the file are displayed in the multi-selection list box. Select the charts. Use the list box below and the color button to change the chart colors, if it is needed, and click "Apply Load".

In the tabs, I use the `SliderGdiCtrl` controls as sliders. To position the slider's thumb exactly where you want, left click on the slider to set the focus to it, and use arrow keys to move the thumb.

If the data view window is visible or minimized when the user changes chart attributes, appends, truncates, or remove chart, the data view will be updated automatically.

I suggest the following scenario to play:

1. Add 4-5 charts to the container. Use different dash styles, pen widths, curve types, and number of data points.
2. Play with the container: zoom in along X- and Y-axes, pane, invoke the data and name labels. Try all items of the popup menu. Do not forget to save the container to an XML file (see User's Manual chapter above).
3. Save the container image in any format you like (use the container's popup menu.)
4. Clone the container and try to resize the clone's window (use tab 6).
5. Append the charts in the container (tab 3).
6. Truncate one or all charts (tab 4).
7. Remove all charts from the container (tab 5).
8. Change the background color and adjust the colors of other elements of the container (tab 0).
9. Load the charts from the saved XML file. Before loading, adjust the colors of the charts you selected (use tab 5).
10. Add to the container the chart with the number of data points 630 and the order of Y-magnitude -1. Select this chart and change its Y-scale with the mouse wheel or the up/down arrow keys.
11. Select this chart in the container and click the "Show chart data" item in the popup menu. The data view window will popup on the screen to the right of the main demo dialog box. Navigate over the data view with data view buttons.
12. Invoke the names and data legends on the screen. The names legend is called from the context menu, the data legend is called by the middle button click to enable the tracking mode first. You will see the cursor change to the cross shape. After that click on any place inside the container window to invoke the data legend. Go to the tab 2, "Change Chart Attributes". From the list box, select the same chart that is selected in the container window and its data are shown in the data view. Change any attribute or combination of the attributes and click "Apply". Observe the changes in the labels and the data view.
13. Invent your own scenarios.

The demo source code can be used as a reference design:

Task	**Reference Header**	**Reference source file**
Change colors of cont. elements	*DlgGenProp.h*	*DlgGenProp.cpp*

Task	Reference Header	Reference source file
Set precision and X-extent	*DlgGenProp.h*	*DlgGenProp.cpp*
Add charts	*DlgAddChart.h*	*DlgAddChart.cpp*
Change chart attributes	*DlgChangeChart.h*	*DlgChangeChart.cpp*
Append charts	*DlgAppendChart.h*	*DlgAppendChart.cpp*
Truncate chart	*DlgTruncate.h*	*DlgTruncate.cpp*
Remove chart	*DlgRemoveChart.h*	*DlgRemoveChart.cpp*
Load chart from XML file	*DlgMisc.h*	*DlgMisc.cpp*
Clone container	*DlgMisc.h*, *DlgCharts.h*	*DlgMisc.cpp*, *DlgCharts.cpp*

The source code and demo projects

The file *ChartCtrlLib.zip* includes all source files for the ChartCtrlLib static library. It includes:

- *ChartCtrlLibSource.zip* - all source files for the static library ChartCtrlLib.
- *ChartCtrlDemoSource.zip* - all source files for the demo application.
- *ChartCtrlLibKit.zip* - files *ChartDef.h*, *ChartContainer.h*, and the compiled libraries *ChartCtrlLibD.lib* and *ChartCtrlLib.lib*. It includes everything you need to use the chart control in your application.
- *ChartCtrlLibKitVS2012.zip* - files *ChartDef.h*, *ChartContainer.h*, and the compiled libraries *ChartCtrlLibD2012.lib* and *ChartCtrlLib2012.lib* to use with VS 2012 VC++ 11 projects.
- *ChartCtrlLibDoxigen.zip* - HTML files with documentation for the ChartCtrlLib classes. Note that to use the links to the source files, you must first extract them into the folder *C:/VS2010/Projects/317712/ChartCtrlLib*.
- *ChartCtrlDemo.exe* - release version of the demo application. It was compiled and linked with the static MFC libraries.

History

- 01/20/2012: Initial version.
- 04/28/2912: Version 1.1.

  Changes and additions:

  - Compiler options related to optimization, /GL and /LTCG, are removed.
  - The new curve style, that draws the chart data points as disconnected crosses, is added to the chart dash styles.
  - The user can set the X-axis name according to his/her choice instead of default "X".
  - The user can set the Y-axis name for each chart individually, instead of default "Y".
  - Y-precision can be set individually for each chart.
  - The user can supply formatting functions for X-values and for each chart's Y-values.
  - The "Set" functions for charts now accept -1 as a chart Idx. It means "All visible charts".
  - The `ChartContainer` now sends the notification messages to its parent when the charts' visibility, data points presentation, or the container's X-extension are changed.
  - The version info definitions and access functions are included.
- 01/26/2013: Version 1.2

  Changes and additions

  - The new feature to save charts as image in any of Windows supporter picture formats..
  - The new feature to programmatically equalize the visible vertical size of the charts.
  - The new feature to block user access to make the container "read only."
  - The signature of the `CChartContainer::SaveChartData` was changed to allow to save all charts in the container, visible and not visible.
  - The constraint `pntNmb >= 3` is removed from the functions `AddChart` and `AppendChartData`.
  - The signatures of the overloaded functions `AddChart`, `AppendChartData`, and `ReplaceChartData` for time series are changed to allow the programmer to set the time origin and time step of the time date series..
  - The functions `CChartContainer::SetChartVisibility` and `CChartContainer::GetChart` now accept the parameter `chartIdx` = -1.
  - The notification with code `CODE_REFRESH` is added.
  - The library port to VS 2012 VC++ 11 (*ChartCtrlLibKitVS2012.zip*) is added.
- 02/28/2013: Version 1.2.1. Bug introduced in v. 1.2.0 in the function `DrawLabel(..)` is fixed (file *DataLabel.cpp*).
- 06/15/2013: Version 2.0.

  Changes and additions:

- The new feature: Now container accepts charts without data and charts with one and two data points.
- The new feature: Zooming and panning along the Y-axis.
- The signatures of `SaveChartData` and `SaveContainerImage` are changed to make these functions callable from the container's parent.
- Improved functionality of many functions.
- Added notification codes for cloning container, and other events (see above in article and in *ChartDef.h*).
- Fixed bugs I and readers have found.

License

Written By

geoyar

Software Developer Verizon Internet Services
United States

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.

Comments and Discussions