KLARÄLVDALENS DATAKONSULT AB

Programmer's Manual

KD CHART 1.1

Table of Contents

List of Figures

Chapter 1. Introduction

KD Chart is Klarälvdalens Datakonsult AB's charting package for Qt applications. It features a large number of different chart types and literally hundreds of configuration options for tailoring a chart to your needs. Since all configuration settings have reasonable defaults you can usually get by with setting only a handful of parameters and relying on the defaults for the rest.

This is the KD Chart Programmer's Manual. It will get you started with creating your charts and provides lots of pointers to advanced features. Besides this manual, there are two other documents:

- Depending on your KD Chart version, you will find different `INSTALL` files that explain how to install KD Chart on your platform or how to build it from sources.
- KD Chart comes with an extensive Reference Manual that is generated directly from the source code itself.

You should refer to these in conjunction with this Programmer's Manual.

What You Should Know

You should be familiar with writing Qt applications, as well as have a working C++ knowledge. When you are in doubt about how a Qt class mentioned in this Programmer's Guide works, please check the Qt reference documentation or a good book about Qt.

The Structure of This Manual

This manual starts with an introduction about how to create a simple chart in Chapter 2, *Three Steps to Your Chart*. You will learn there how to specify the data that will be visualized by KD Chart and how to create a set of parameters that describes how the chart is drawn.

The following Chapter 3, *Specifying the Chart Type* lists each of the supported chart types together with information about how to configure them, as well as which subtypes exist for each chart type.

The subsequent chapters contain more advanced material like how to specify fonts, colors, and sizes if you are dissatisfied with KD Chart's default settings, how to create and display legends, headers and footers, how to add custom boxes, how to use external graphics, and how to make KD Chart use your data structures instead of passing your values into the tables of KD Chart. The manual closes with information about how to make your charts interactive, zoom and scroll them or combine several chart widgets.

You will find lots of sample code, together with screen shots that show in which display the sample code results in. We recommend our readers to try and run the sample code (which is provided electronically in the KD Chart distribution) and experiment with the various settings.

What's next

In the next chapter you will learn about the three basic steps to create a chart...

Chapter 2. Three Steps to Your Chart

This chapter shows you the basic steps you will take to make KD Chart display your chart. After presenting the procedure in theory we will look into the source code of a small sample program and learn some details about the underlying principles.

The Procedure

These three steps let you display your chart:

1. Create a `KDChartParams` object that contains all the settings of your chart. Usually, you will be able to accept the default settings in this object for most cases and only change those typically few settings that apply particularly to your application. For example, if the default axis labels are just fine for you, you never need to bother about setting up axes; KD Chart will do that for you automatically.

2. Create a `KDChartTableData` object and fill it with the data you want to visualize in a chart. This can either be done manually by setting the individual fields to their respective values or by importing the data from a `QTable` object in one fell swoop. Future versions of KD Chart will also enable you to visualize data stored in a Relational Database.

3. Let KD Chart draw the actual chart. There are two ways of doing this, a simple one and a flexible one. With the simple method that we will explore first, you create an object of class `KDChartWidget` and pass it a `KDChartTableData` and a `KDChartParams` object. The widget then takes care of drawing and repainting the chart where necessary and is maintenance-free. However, if you want the full flexibility, which includes advanced features like rotating and scaling, you can also use the static method `KDChart::paint()` and pass it, besides the aforementioned objects, a pointer to a `QPainter` which you have configured to your needs; in either case be sure to read the notes on Printing Your Chart at the end of this chapter to optimize your chart for screen or printer output respectively.

Note

The `KDChartWidget` constructor does *not copy* the `KDChartParams` object *nor* the `KDChartTableData` object: it just stores their addresses.

This means that your application is responsible for keeping these two objects alive while using that `KDChartWidget` instance *and* for deleting them when they are no longer needed.

On the other side you gain the advantage of being able to modify your parameters and/or your data at any time: all you have to do is redrawing the widget after your modification.

Example

Let us make this somewhat abstract description more concrete by looking at an actual piece of code. kdchart_step01.h and kdchart_step01.cpp show a simple demo application that follows these three steps and this shows a chart.

```
/* -*- Mode: C++ -*-
   KDChart - a multi-platform charting engine
*/

/****************************************************************************
** Copyright (C) 2001-2003 Klar#lvdalens Datakonsult AB.  All rights reserved.
**
** This file is part of the KDChart library.
**
** This file may be distributed and/or modified under the terms of the
** GNU General Public License version 2 as published by the Free Software
** Foundation and appearing in the file LICENSE.GPL included in the
** packaging of this file.
**
** Licensees holding valid commercial KDChart licenses may use this file in
** accordance with the KDChart Commercial License Agreement provided with
** the Software.
**
** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
**
** See http://www.klaralvdalens-datakonsult.se/Public/products/ for
**   information about KDChart Commercial License Agreements.
**
** Contact info@klaralvdalens-datakonsult.se if any conditions of this
** licensing are not clear to you.
**********************************************************************/
#ifndef KDCHART_STEP01_H
#define KDCHART_STEP01_H

#include <qwidget.h>

class TutorialWidget : public QWidget
{
    Q_OBJECT
public:

   TutorialWidget(QSize s, QWidget * parent=0, const char * name=0, WFlags f=0)
     : QWidget( parent, name, f )
     , mS( s ) {}
    ~TutorialWidget();

   QSize sizeHint() const
   {
     return mS;
   }

public slots:
    virtual void saveAndClose();

private:

   QSize mS;

};

#endif
```

The header file does not contain anything particular. We just create a subclass of `QWidget`, named `TutorialWidget`, for sizing and documenting purposes. KD Chart is not even mentioned here, all respective `#include` statements are specified in the implementation file.

The implementation file listed below uses an instance of the `TutorialWidget` as main widget.

```
 1 /* -*- Mode: C++ -*-
      KDChart - a multi-platform charting engine
   */

 5 /****************************************************************************
   ** Copyright (C) 2001-2003 Klar#lvdalens Datakonsult AB.  All rights reserved.
   **
   ** This file is part of the KDChart library.
   **
10 ** This file may be distributed and/or modified under the terms of the
   ** GNU General Public License version 2 as published by the Free Software
   ** Foundation and appearing in the file LICENSE.GPL included in the
   ** packaging of this file.
   **
15 ** Licensees holding valid commercial KDChart licenses may use this file in
   ** accordance with the KDChart Commercial License Agreement provided with
   ** the Software.
   **
   ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
20 ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
   **
   ** See http://www.klaralvdalens-datakonsult.se/?page=products for
   **   information about KDChart Commercial License Agreements.
   **
25 ** Contact info@klaralvdalens-datakonsult.se if any conditions of this
   ** licensing are not clear to you.
   **
   **********************************************************************/
   #include <iostream>
30 #include "../kdchart_tutorial_widget.h"


   #include <qapplication.h>
   #include <qtimer.h>
35 #include <qlayout.h>
   #include <qlabel.h>
   #include <qpixmap.h>

   #include <KDChartWidget.h>
40 #include <KDChartTable.h>
   #include <KDChartParams.h>
   #include <KDChartAxisParams.h>

   int main ( int argc, char* argv[] )
45 {
       QApplication a ( argc, argv );

       // declare the main widget before processing the command line
       // parameters, so we can connect a timer shot to the widget...
50     TutorialWidget* mainWidget = new TutorialWidget( QSize(500,500) );

       // process our parameters to enable screenshot mode possibly
       bool bScreenshotMode = false;
       for ( int i = 1; i < a.argc(); ++i ){
55         QString s = QString(argv[i]).upper();
           if( s == "-S" ){
               if( !bScreenshotMode ){
                   // we make a screenshot of the widget's content and quit
                   QTimer::singleShot(2000,mainWidget,SLOT(saveAndClose()));
```

```
60                bScreenshotMode = true;
              }
          }else if( s == "-?" || s == "/?" || s == "-H" || s == "/H" ||
                    s == "-HELP" || s == "/HELP" || s == "--HELP" ){
              // we show the help text and end the program
65            std::cout << "\n\n" << argv[0]
              << " -s      saves the widget as file image.png and quits\n";
              return 0;
          }
      }
70
      // ************************************************************************
      // Main Widget Layout
      // ************************************************************************
          // make it compile on windows using qt232
75        #if QT_VERSION >= 200 && QT_VERSION < 300
          mainWidget->setPalette( bScreenshotMode
                                                          ? Qt::white
                                                          : Qt::lightGray );
          #else
80    mainWidget->setPaletteBackgroundColor( bScreenshotMode
                                            ? Qt::white
                                            : Qt::lightGray );
          #endif
      mainWidget->setCaption( "KDChart Tutorial - Step 1" );
85    QVBoxLayout theLayout( mainWidget );
      theLayout.setMargin( 16);
      theLayout.setSpacing(16);
      if( !bScreenshotMode ){
          QLabel* label1 = new QLabel("a basic chart, default type:", mainWidget);
90            // make it compile on windows using qt232
              #if QT_VERSION >= 200 && QT_VERSION < 300
              label1->setPalette( Qt::white );
              #else
              label1->setPaletteBackgroundColor( Qt::white );
95            #endif
          theLayout.addWidget( label1, 0 );
      }

      // ************************************************************************
100   // Chart Params
      // ************************************************************************
      KDChartParams p;❶

      // increase the font size to be used for the data value texts
105   p.setPrintDataValuesFontRelSize( 0, 29 );

      // ************************************************************************
      // set Chart Table Data
      // ************************************************************************
110   KDChartTableData d( 3, 5 );❷
      // 1st series
      d.setCell( 0, 0,    17.5   );
      d.setCell( 0, 1,   110.4   );  // highest value
      d.setCell( 0, 2,     6.67  );  // lowest value
115   d.setCell( 0, 3,    33.333 );
      d.setCell( 0, 4,    30     );
      // 2nd series
      d.setCell( 1, 0,    40     );
      d.setCell( 1, 1,    40     );
120   d.setCell( 1, 2,    45.5   );
      d.setCell( 1, 3,    45     );
      d.setCell( 1, 4,    35     );
      // 3rd series
      d.setCell( 2, 0,    20     );
125   // missing value: d.setCell( 2, 1,   25 );
      d.setCell( 2, 2,    30     );
      d.setCell( 2, 3,    45     );
      d.setCell( 2, 4,    40     );
```

```
130     // ************************************************************************
        // Create Chart Widget, and add to layout
        // ************************************************************************
        KDChartWidget *chartWidget = new KDChartWidget(&p, &d, mainWidget);❸

135         // make it compiles on windows using qt232
            #if QT_VERSION >= 200 && QT_VERSION < 300
            chartWidget->setPalette( Qt::white );
            #else
        chartWidget->setPaletteBackgroundColor( Qt::white );
140         #endif
        theLayout.addWidget( chartWidget, 1 );

        // this is for the tutorial's screenshot function:
        mainWidget->registerWidget( chartWidget );
145
        a.setMainWidget( mainWidget );

        mainWidget->show();

150     return a.exec();
    }
```

❶ Calling the default constructor we create an instance of the `KDChartParams` class without specifying any parameters. For the moment we ignore that this sample file calls the `KDChartParams::setPrintDataValuesFontRelSize()` method since font size optimization is described below, see chapter Customizing Your Chart / Fonts.

❷ Data input is done in two steps: first we instantiate the `KDChartTableData` class to obtain a table for storing three data series with five cells each, then we put our data into this table using the first of the `KDChartTableData::setCell()` functions. To demonstrate that it is not necessary to fill all cells we skip cell (2,1).

❸ Drawing goes the usual way: in the main widget's layout we insert an instance of `KDChartWidget`, this instance receives pointers to the params, to the data and to the main widget, its parent.

Resulting from this basic setup we get a bar chart showing three series (using the default per-data series colors) and five items per series—with the exception of series #2 having only four items. Each item is represented by a separate bar, and their respective values are written in 45 degree rotated way above the bars, KD Chart recognizes our empty cell and shows a gap instead drawing of a bar for it.

This simple example allows us to see a very basic feature of KD Chart: automatic adaption to the widget size. Just resize the program window to see the font sizes and bar widths adapt accordingly, see the Fonts chapter for more information on this feature.

Printing Your Chart

To obtain best results when printing you should prevent KD Chart from optimizing your chart for the screen, as this is the default setting. There are two ways to do that:

1. Either turn off screen optimization by calling `setOptimizeOutputForScreen( false )` on your `KDChartParams` object before printing—and make sure to set it

back afterwards.

2. Or use the `KDChartWidget` convenience function `print(QPainter& painter, const QRect* rect)` and let it take care for (re)setting the optimization flag instead of doing that yourself.

Using either method produces equal printout results, KD Chart just behaves according to the status of the optimization flag, so there normally is no reason for calling `setOptimizeOutputForScreen()` yourself, at least if you *have* a `KDChartWidget`—if not, make sure to set the flag on your `KDChartParams` before passing them to `KDChart::paint()`.

Have a look at tutorial file `kdchart_step02.cpp` to see how these techniques are used in function `TutorialWidget::slotPrintChart()`. Running this program you can compare printouts of the chart in high quality mode to printing the window's raw screen representation.

What's next

In the next chapter you will learn how to use different chart types depending on the structure and purpose of your chart...

Chapter 3. Specifying the Chart Type

This chapter tells you how to change the chart type from the default (normal bar chart) to either one of the other bar chart subtypes or an entirely different chart type. All of the chart (sub)types provided by KD Chart are presented by some very basic sample programs and their screen shots.

To select the chart type just call the `KDChartParams` function `setChartType()` and/or specify the subtype using a method like `setBarChartSubType()`.

This chapter tells you which chart (sub)type might be appropriate for which purpose and it provides information on special features that are available for the respective chart (sub)types.

Bar Charts

Tip

Bar charts are the most common type of charts and can be used for visualizing almost any kind of data. Like the Line Charts the bar charts can be the ideal choice to compare multiple series of data.

A good example for using a bar chart would be a comparison of the sales figures in different departments, perhaps accompanied by a High/Low Chart showing each day's key figures.

KD Chart's default type is the bar chart so no command is necessary to get one, however after having used your `KDChartParams` to display another chart type you can reset the type by calling `setChartType( KDChartParams::Bar )`.

Bar charts typically have axes on which this manual provides extended information in the Axis Manipulation chapter.

What happens to missing data cells?

Missing data cells in a bar chart are represented by an empty space between the neighboring bars. While these gaps look similar to bars with zero value they can easily be distinguished from them since no Data Value Text is printed for missing cells: zero data cells are represented by a "0.0" value text by default.

Your bar chart can have one of the following subtypes explained below:

- Normal Bar Chart
- Stacked Bar Chart
- Percent Bar Chart
- Three-dimensional Multi-Row Bar Chart

Your bar chart can have one of the following subtypes explained below:

Note

Three-dimensional look of the bars is no special feature of the multi-row bar charts: you can enable it for the other subtypes (`BarNormal`, `BarStacked`, `BarPercent`) by calling the `KDChartParams` method

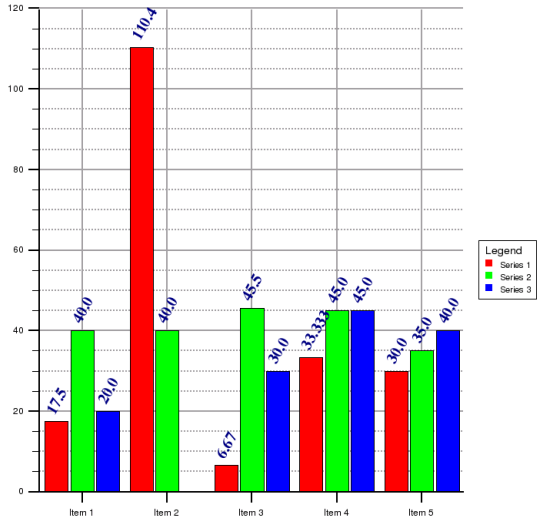
`setThreeDBars( true )`. By default the shadow colors for the three-dimensional look are calculated automatically, see the Colors chapter.

Normal Bar Chart

Tip

In a normal bar chart, each individual value is displayed as a bar by itself. This flexibility allows to compare both the values in one series and values of different series.

Figure 3.1. A Normal Bar Chart



KD Chart's default type is the normal bar chart so no method needs to be called to get one, however after having used your `KDChartParams` to display another bar chart subtype you can return to the normal one by calling `setBarChartSubType( KDChartParams::BarNormal )`.

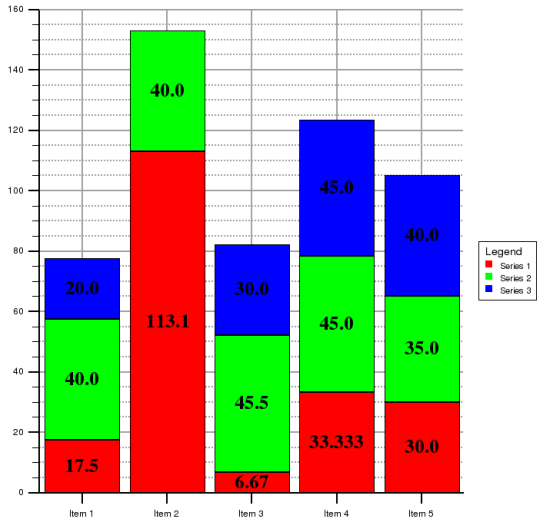
Compile and run the tutorial file `kdchart_step01.cpp` to see a normal bar chart of three datasets containing up to 5 items each (see Figure 3.1).

Stacked Bar Chart

Tip

Stacked bar charts focus on comparing the sums of the individual values in each data series, but also show how much each individual value contributes to its sum.

Figure 3.2. A Stacked Bar Chart



Stacked mode for bar charts is activated by calling the `KDChartParams` function `setBarChartSubType( KDChartParams::BarStacked )`.

See the tutorial file `kdchart_step01a.cpp` and Figure 3.2 for an example of a stacked bar chart. To achieve better readability of the Data Value Texts you might want to print them not rotated (or rotated by 270 degrees, resp.) and position them in the center of their respective segments—this can be achieved by calling the `KDChartParams` method `setPrintDataValues()`, the tutorial file shows the parameters needed.

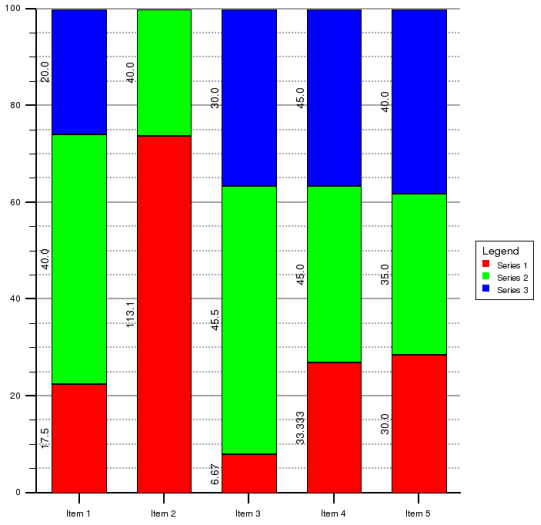
Please study the chapter on Data Value Texts for detailed information, e.g. you might use the special value `KDCHART_DATA_VALUE_AUTO_COLOR` to make KD Chart automatically calculate a good color for the texts inside instead of using a fixed one.

Percent Bar Chart

Tip

Unlike stacked bar charts, percent bar charts are not suitable for comparing the sums of the data series, but rather focus on the respective contributions of their individual values.

Figure 3.3. A Percent Bar Chart



Percentage mode for bar charts is activated by calling the `KDChartParams` function `setBarChartSubType( KDChartParams::BarPercent )`.

See the tutorial file `kdchart_step01b.cpp` and Figure 3.3 for an example of a percent bar chart showing the Data Value Texts written using special font and vertically next to their respective segments while the gap between the columns is increased to make things look better.

Three-dimensional Bar Charts in Multiple Rows

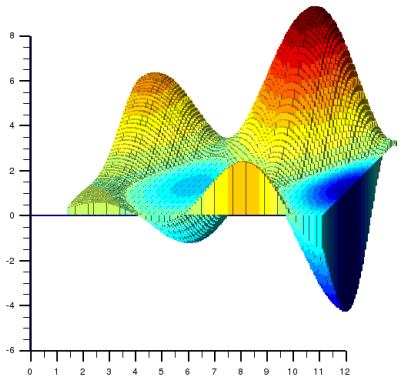
Tip

This type of bar chart allows to show lots of data values in limited space. It is less suited to compare data values with each other.

Note

Three-dimensional bar charts in multiple rows are an experimental feature that will be vastly improved in future versions. For example, no Z axis can be displayed yet.

Figure 3.4. A 3D Bar Chart With Multiple Rows



For having a look at this new feature just call the `KDChartParams` function `setBarChartSubType( KDChartParams::BarMultiRows )`.

See the tutorial file `kdchart_step01c.cpp` and Figure 3.4 for an example of a 3D bar chart with multiple rows: note that this chart type has no gaps between the bars. See the chapter on Cell specific Properties to learn how to specify the color of each bar individually.

Line Charts

Tip

Line charts usually show numerical values and their development in time. Like the Bar Charts they can be used to compare multiple series of data.

Examples might be the development of stock values over a longer period of time or the water level rise on several gauges.

KD Chart can generate line charts of different kind, you can activate line chart mode by the `KDChartParams` function `setChartType( KDChartParams::Line )`.

Line charts typically have axes on which this manual provides extended information in the Axis Manipulation chapter.

What happens to missing data cells?

Missing data cells in a line chart are not represented by an empty space but they are skipped: the respective line does not have a gap but it connects the neighboring cell's points.

Your line chart can have one of the following subtypes explained below:

- Normal Line Chart
- Stacked Line Chart
- Percent Line Chart

Your line chart can have one of the following subtypes explained below:

Note

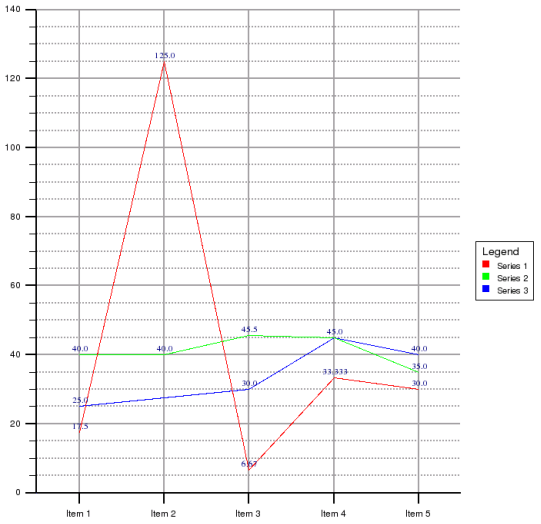
To enable three-dimensional look for any of the line chart subtypes just call the `KDChartParams` method `setThreeDLines( true )`, by default the shadow colors for the three-dimensional look are calculated automatically, see the chapter on Colors for details.

Normal Line Chart

Tip

Normal line charts are the most common type of line charts and are used when the datasets are compared to each other individually. For example, if you want to visualize the development of sales figures over time for each department separately, you might have one line per department.

Figure 3.5. A Normal Line Chart



KD Chart draws normal line charts by default when in line chart mode so no method needs to be called to get one, however after having used your `KDChartParams` to display another line chart subtype you can reset it by calling `setLineChartSubType( KDChartParams::LineNormal )`.

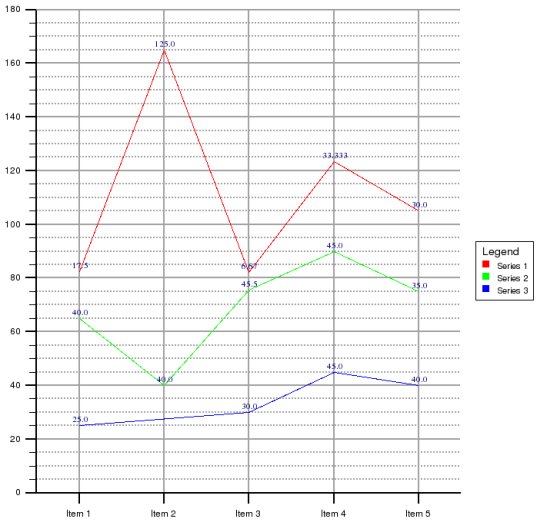
Compile and run the tutorial file `kdchart_step01d.cpp` and Figure 3.5 to see a normal line chart of three datasets containing different numbers of items: the missing item #2 in series #3 is just ignored, the line not interrupted.

Stacked Line Chart

Tip

Stacked line charts allow you to compare the development of a series of values summarized over all datasets. You could use this if you are only interested in the development of total sales figures in your company, but have the data split up by department.

Figure 3.6. A Stacked Line Chart



Stacked mode for line charts is activated by calling the `KDChartParams` function `setLineChartSubType( KDChartParams::LineStacked )`.

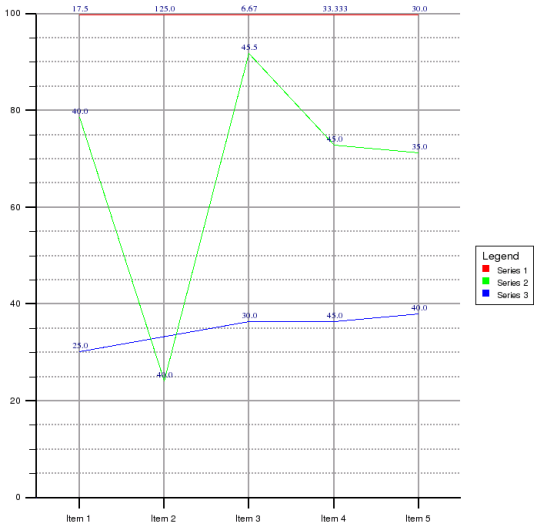
See the tutorial file `kdchart_step01e.cpp` and Figure 3.6 for an example of a stacked line chart.

Percent Line Chart

Tip

Percent line charts show how much each value contributes to the total sum, similar to percent bar charts.

Figure 3.7. A Percent Line Chart



Percentage mode for Line charts is activated by calling the `KDChartParams` function `setLineChartSubType( KDChartParams::LinePercent )`.

See the tutorial file `kdchart_step01f.cpp` and Figure 3.7 for an example of a percent line chart.

Line Charts with Markers

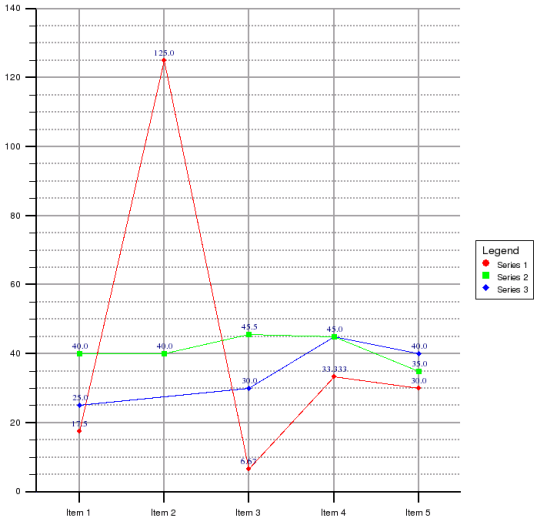
Line charts can be enhanced with so-called *markers*, little geometric shapes marking the position of each data point.

To enable these markers just call `KDChartParams::setLineMarker( true )`.

Tip

Markers in line charts can help emphasize the individual data points. If your line chart is Reporting Mouse Events you *should* enable the markers to give your users something to click on, otherwise the active regions will be some invisible squares of 3x3pt each centered at the data points.

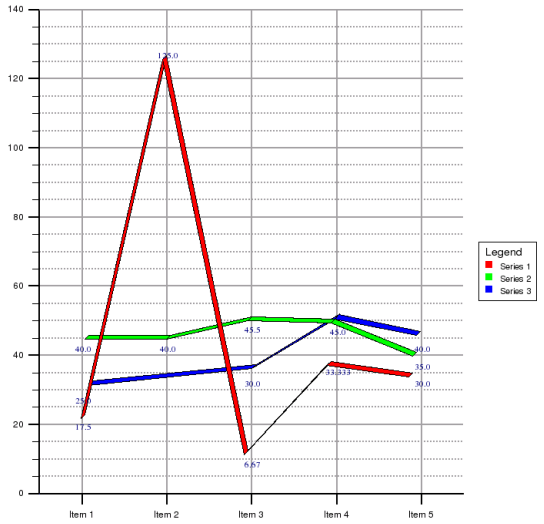
Figure 3.8. A Line Chart With Markers



Compile and run the tutorial file `kdchart_step01g.cpp` to see a normal line chart with additional markers using the default colors, shapes and sizes (see also Figure 3.8.

Line Charts with Three-D-Look

Figure 3.9. Line Charts With Three-D-Look



Line charts in Three-D-Look are just another way to display your two-dimensional data: one or more data series containing one or more cells each. You can enable Three-D-Look by calling the `KDChartParams` method `setThreeDLines( true )`.

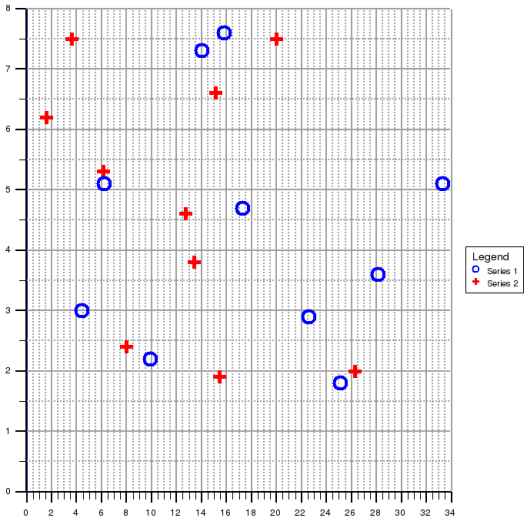
Compile and run the tutorial file `kdchart_step01h.cpp` to see a line chart in Three-D-Look using the default properties: here the data values are printed below their respective points by default (see Figure 3.9).

Point Charts

Tip

Point charts often are used to visualize a big number of data in one or several datasets. A well known point chart example is the historical first Herzsprung-Russel diagram from 1914 where circles represented stars with directly measured parallaxes and crosses were used for guessed values of stars from star clusters slightly similar to this simple chart (see Figure 3.10):

Figure 3.10. A Point Chart



Unlike the other chart types in KD Chart the point chart is not a type of its own but actually a special kind of Line Charts, so you activate point chart mode by calling the following three `KDChartParams` methods:

```
setChartType( KDChartParams::Line ); // switch to line chart mode
setLineMarker( true );               // make the markers visible
setLineWidth( 0 );                   // do not draw the lines
```

Compile and run the tutorial file `kdchart_step01z.cpp` to see a simple point chart (see also Figure 3.10): here no Data Value Texts are shown because we suppressed them by `setPrintDataValues( false )`.

Pie Charts

Tip

Pie charts can be used to visualize the relative values of a few data cells (typically 2..20 values). Larger amounts of items can be hard to distinguish in a pie chart, so a Percent Bar Chart might fit your needs better then. Pie charts are most suitable if one of the data elements covers at least one forth, preferably even more of the total area.

A good example is the distribution of market shares among products or vendors.

While pie charts are nice for displaying *one* dataset there is a complementary chart type you might choose to visualize several datasets: the Ring Chart.

Note

The Ring Charts chapter describes a circular *multi-dataset* chart type.

Pie charts typically consist of two or more pieces any number of which can be shown 'exploded' (shifted away from the center) at different amounts, starting position of the first pie can be specified and your pie chart can be drawn in three-D look. Activating the pie chart mode is done by calling the `KDChartParams` function `setChartType( KDChartParams::Pie )`.

What happens to missing data cells?

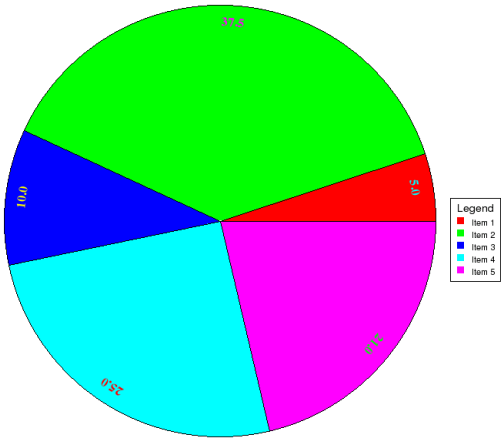
Missing data cells in a pie chart are not represented by an empty space but they are skipped: the respective piece just is not drawn nor counted.

Simple Pie Charts

Tip

A simple pie chart shows the data without emphasizing a special item.

Figure 3.11. A Simple Pie Chart



KD Chart by default draws two-dimensional pie charts when in pie chart mode so no method needs to be called to get one, however after having used your `KDChartParams` to display a three-dimensional pie chart you can reset it by calling `setThreeDPies( false )`.

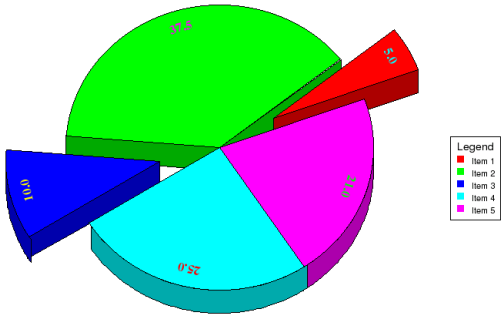
Compile and run the tutorial file `kdchart_step01i.cpp` to see a normal pie chart (see Figure 3.11.

Exploding Pie Charts with Three-D-Look

Tip

While the three-dimensional-look of the pie chart is more an esthetic feature, exploding individual segments allows to emphasize individual data.

Figure 3.12. An Exploding Pie Chart With Three-D-Look



Call `KDChartParams::setThreeDPies( true )` for three-dimensional pie charts.

To get a chart like the one above (having one or several of the pieces separated from the others in *exploded* mode) you would pass a `QValueList` containing the pieces' numbers to KD Chart and specify the explode factor, as shown in the following excerpt of the tutorial file `kdchart_step01j.cpp` (see also Figure 3.12):

```
p.setExplode( true );
p.setExplodeFactor( 0.2 );
QValueList<int> explodeList;
explodeList.clear();
explodeList << 0 << 2;
p.setExplodeValues( explodeList );
```

Other possible options for configuring your pie chart include using `setPieStart(int)` or `setThreeDPieHeight(bool)`, see the reference documentation of class `KDChartParams` for more on this.

Ring Charts

Tip

While a Pie Chart might be a good choice when displaying a single data series, using a ring chart might be ideal for visualizing a small amount of data cells stored in several datasets: ring charts can show both the relative values of the cells compared to their dataset's total value *and* the relation of the series totals compared to each other. This is done by using relative ring widths as shown below.

Ring charts (like Pie Charts) typically consist of two or more segments any number of which can be shown 'exploded' (shifted away from the center), activating the ring chart mode is done by calling the `KDChartParams` function `setChartType( KDChart-Params::Ring )`.

What happens to missing data cells?

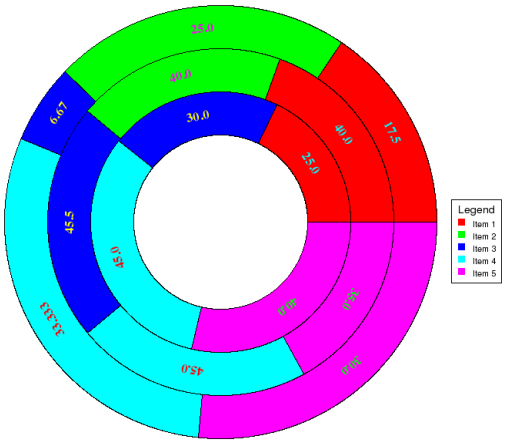
Missing data cells in a ring chart are not represented by an empty space but they are skipped: no segment is drawn for them.

Simple Ring Charts

Tip

If you do not care about the relative size of the sums of values in each datasets, simple ring charts are your chart of choice.

Figure 3.13. A Simple Ring Chart



KD Chart by default draws non-exploded rings with equal thickness when in ring chart mode so no methods need to be called to get such charts, however after having used your `KDChartParams` to display a more sophisticated ring chart you can reset from relative-thickness by calling `setRelativeRingThickness( false )` while explosion mode can be reset by `setExplode( false )`.

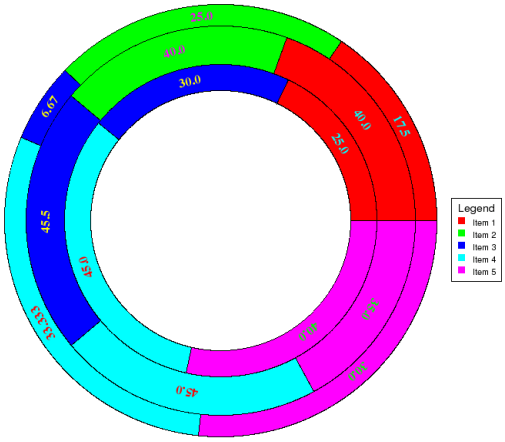
Compile and run the tutorial file `kdchart_step01k.cpp` to see a normal ring chart (see Figure 3.13).

Ring Charts with Relative Thickness

Tip

Looking similar to a tree's annual rings these charts might be a good choice to display several years-related volumes of data like sales numbers. The segments could stand for the different product lines in a sortiment of goods. Looking at a ring's thicknesses you then could see the change in sales for *all* of your goods while a segment's length would show you how much this product line has contributed to the respective year's total turnover—compared to your other products.

Figure 3.14. A Ring Chart With Relative Thickness



Relative thickness mode is activated by calling the `KDChartParams` function `setRelativeRingThickness( true )` where each ring represents one dataset and the ring widths show the relations of the dataset totals to each other.

Compile and run the tutorial file `kdchart_step01l.cpp` to see a ring chart featuring three datasets: the thickness of the middle ring shows clearly that this series represents the biggest total value (see Figure 3.14).

Ring Charts Exploding Segments

Tip

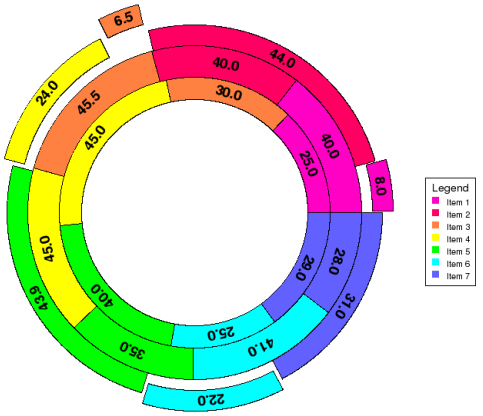
Explode one or several of the segments of your ring chart to emphasize the respective data cell(s). You might use this for drawing attention to sales figures below a critical level of for highlighting a very successful item.

To have one or several segments of your ring chart shown exploded you activate the exploded mode, store the respective segment numbers in a `QValueList` and pass this list to your `KDChartParams` class. Additionally, you can specify the width of the gap caused by the explosion, as shown in the following example:

```
// explode first and third segment by factor 0.1
p.setExplode( true );
p.setExplodeFactor( 0.1 );
QValueList<int> explodeList;
explodeList.clear();
explodeList << 0;
explodeList << 2;
p.setExplodeValues( explodeList );
```

In case you want to apply *different* explode factors to the segments just call the `KDChartParams::setExplodeFactors()` function and pass to it a `KDChartParams::ExplodeFactorsMap` with one `double` value for every segment. The following ring chart (see Figure 3.15) has exploded all segments with values less than their ring's average while the smallest value is exploded even more.

Figure 3.15. A Ring Chart With Exploding Segments



Compile and run the tutorial file `kdchart_step01m.cpp` to see a ring chart featuring both, relative thickness of the rings and differently exploded segments on the outer ring (see Figure 3.15). Have a look at the chapters on Colors and on Fonts for details on the other methods used in this program.

Note

Only segments that are located on the *outer* ring can be exploded.

Area Charts

Tip

Even more than a Line Chart (of which they are a special case) an area chart can give a good visual impression of different datasets and their relation to each other.

An area chart might be the best choice for showing how several sources contributed to increasing ozone values in a conurbation during a summer's months.

Area charts (similar to Line Charts) are based upon several points which are connected by lines—the difference to the line chart is that the area below a line is filled by the respective dataset's color. While giving a good impression of each dataset's relative values these filled areas can also be a disadvantage: as shown below the Simple Area Chart often makes it impossible to see all points since some are covered by another dataset's area. To still make it possible to use an area chart in such cases KD Chart offers the non-overlapping Stacked Area Chart and Percent Area Chart types.

You can activate the area chart mode by calling the `KDChartParams` function `setChartType( KDChartParams::Area )`.

Note

KD Chart uses the term "area" in two different ways which can be distinguished easily:

- In this chapter (like in `KDChartParams::Area`) it stands for a special chart type.
- In the explanations given in Chapter 5, *Areas of a Chart* it means the different (normally rectangular) parts of a chart like the *data area* or the *headers area*.

This varying usage of the word "area" should not cause a lack of clarity: In the context of this special chapter on *area charts* the word is clear, in the rest of the manual it just means a part of a chart.

What happens to missing data cells?

Missing data cells in area charts are not represented by an empty space but they

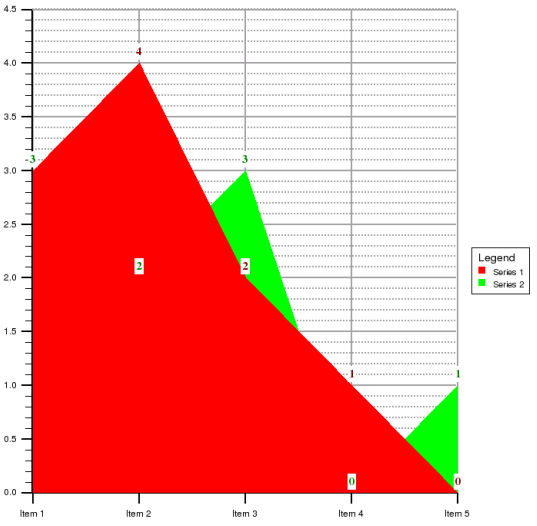
are skipped: the area (like the line in a Line Chart) reaches from the previous point to the next one. This can be exactly what you want but it might also be misleading: test it to find out if this chart type fits your needs, if not you might consider using another one, e.g. a Bar Chart would show the missing cells clearly.

Simple Area Chart

Tip

Simple area charts can be used for displaying not more than two datasets: otherwise the areas most likely will cover too many points. For visualizing more that two series you might might consider using a Stacked Area Chart or even a Percent Area Chart both of which do *not* cover any points behind the areas.

Figure 3.16. A Simple Area Chart



KD Chart by default draws simple (overlapping) areas when in area chart mode so no methods need to be called to get such a chart, however after having used your `KDChartParams` to display a Stacked Area Chart or a Percent Area Chart you can reset it by calling `setAreaChartSubType( KDChartParams::AreaNormal )`.

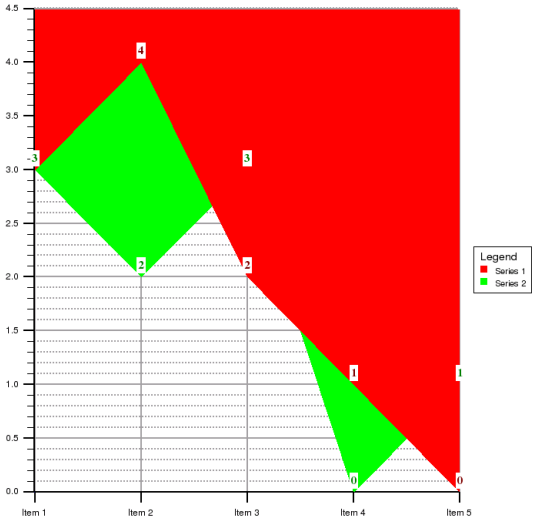
Compile and run the tutorial file `kdchart_step01n.cpp` to see a normal area chart (see Figure 3.16).

Note

Sometimes it is possible to achieve a better design or make your chart more clear by drawing the areas in *reversed* order: some points that were covered in normal mode might become visible when the space *above* the line is filled instead of the space below it.

Compile and run the tutorial file `kdchart_step01o.cpp` to see an area chart in reversed mode. The only difference from the previous file is the command `setAreaLocation( KDChartParams::AreaAbove )` overwriting the default `AreaBelow` setting—Figure 3.17 illustrates the results.

Figure 3.17. A Reversed Area Chart

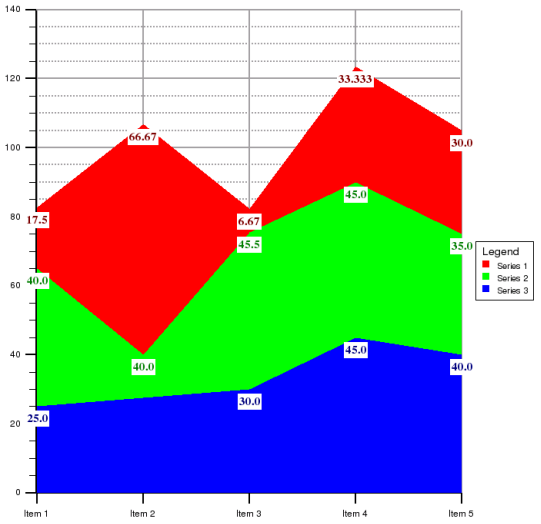


Stacked Area Charts

Tip

Stacked area charts (as well as the Percent Area Charts) can be a good choice for displaying *two or more datasets* since (unlike the Normal Area Charts) their filled areas *never cover* any points behind them.

Figure 3.18. A Stacked Area Chart



To get your area chart drawn in stacked mode just call your `KDChartParams` method `setAreaChartSubType( KDChartParams::AreaStacked )`.

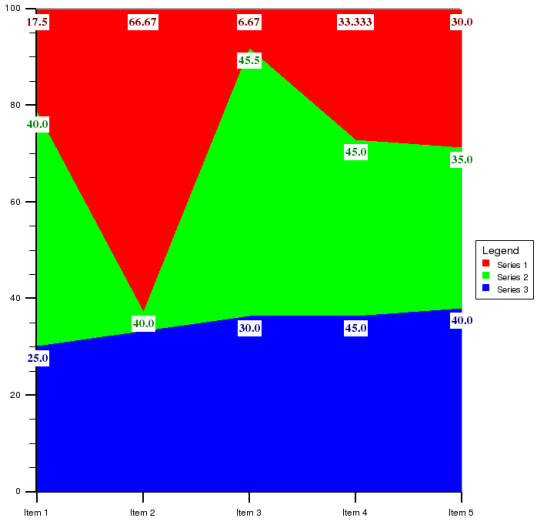
Compile and run the tutorial file `kdchart_step01p.cpp` to see a stacked area chart (see also Figure 3.18) visualizing three datasets: as described at the beginning of this Chapter there is no chance to see that item #2 in series #3 is *not 27.5* but a missing cell, see the special note on missing values (see Area Charts [31]) for the pros and cons of this feature.

Percentage Area Charts

Tip

Area charts in percentage mode (like the Stacked Area Charts) can be a good choice for displaying *two or more datasets* since (unlike the Normal Area Charts) their filled areas *never cover* any points behind them.

Figure 3.19. A Percent Area Chart



To get your area chart drawn in percentage mode just call your `KDChartParams` method `setAreaChartSubType( KDChartParams::AreaPercent )`.

Compile and run the tutorial file `kdchart_step01q.cpp` to see an area chart in percentage mode visualizing three datasets (see also Figure 3.19): as described at the beginning of this Chapter there is no chance to see that item #2 in series #3 is *not 27.5* but a missing cell, see the special note on missing values (see Area Charts [31]) for the pros and cons of this feature.

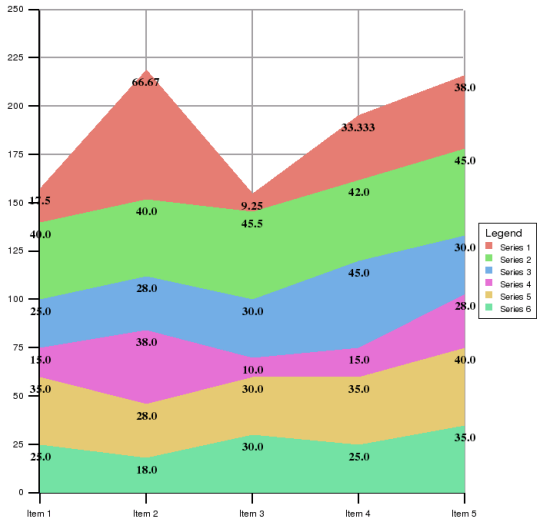
Area Charts In Subdued Colors

Tip

All of the above shown Area chart sub types share the same problem: their Data Value Texts are either hard to read (when using a `Qt::NoBrush` text background) or they are nuisance elements distracting the viewer and destroying the chart's look by their white background rectangles.

To solve this problem without removing the texts you could simply remove the reason for the extra background shown behind the texts: using subdued colors instead of the default color set enables you to show the data value texts without the disturbing background and still have them nicely readable:

Figure 3.20. Stacked Area Chart In Subdued Colors



Compile and run the tutorial file `kdchart_step04t.cpp` to see a nice stacked area chart with readable data value texts (see also Figure 3.20). The colors are set by a very short part of our program featuring the following `KDChartParams` function calls:

```
// activate a subdued color set
// (ideal for area charts with data value texts)
p.setDataSubduedColors();

// disable using a special background for the data value texts
// by re-setting the parameters to their default value
p.setDataValuesColors();
```

An illustration on KD Chart's color sets (see Figure 4.3) is provided by the Colors chapter, for details on `setDataValuesColors()` have a look at the Data Value Texts chapter.

High/Low Charts

Tip

High/low charts could be useful if your value's evolution during a time period should be visualized looking at small segments of this time. Often they are used to display a share's opening and closing value per day, as

well as each day's maximum and minimum. Other possible uses of high/low charts include the changing temperature during the months: open/close would stand for a month's first and last day then.

Typically high/low charts are used to display one value evolving over the time—for showing more than one value either consider using another chart type or use a second `KDChartWidget` positioned next to the first one. Another option might be to show a line chart for all of your values and draw special attention to one of them by adding an extra high/low chart featuring this value. The high/low chart could be positioned above the line chart—sharing its abscissa axis as shown below in the chapter Combining Two Charts.

High/low charts are the most simple ones of KD Chart's statistical chart types and (unlike the sophisticated Box&Whisker Charts) they do *not calculate* each dataset's high/low/open/close values but it is your obligation to pass these numbers into the data cells. This is done for optimization: most likely you have these figures anyway, if not it is extremely easy to determine them, see Figure 8.1 for an example. The respective source code in `kdchart_step08.cpp` shows that these values can even be determined *on the fly*: using the `KDChartTableData::expand()` method there is no need for a two-pass algorithm.

The advantage of not stuffing all your data values into KD Chart is that such very small datasets (holding just four cells each) enable your high/low charts to be displayed very quickly and even when representing a large number of days they will not consume too much memory.

You can activate the high/low chart mode by calling the `KDChartParams` function `setChartType( KDChartParams::HiLo )`.

What happens to missing data cells?

There can be two kinds of missing data problems in high/low charts, the serious one is having a *too small data table* holding not enough cells per dataset: by definition this is not allowed since you store each dataset's high and low values into the data cells and if you want your chart to show the close value (and the open value, resp.) you would store this values as well. Thus if KD Chart finds not enough data cells it does not draw anything but shows a respective error output on console (when in debug mode only).

The other kind of missing data is the usual *missing cells* in an otherwise correct data table, and there will be different results based upon the importance of the missing cell(s):

Missing *high* or *low* values make it impossible to visualize the respective dataset, so nothing is drawn at all for this dataset them.

Missing *open* or *close* values still give KD Chart the chance to draw the vertical high/low line, but the horizontal lines for open(or close, resp.) can not be shown then and will

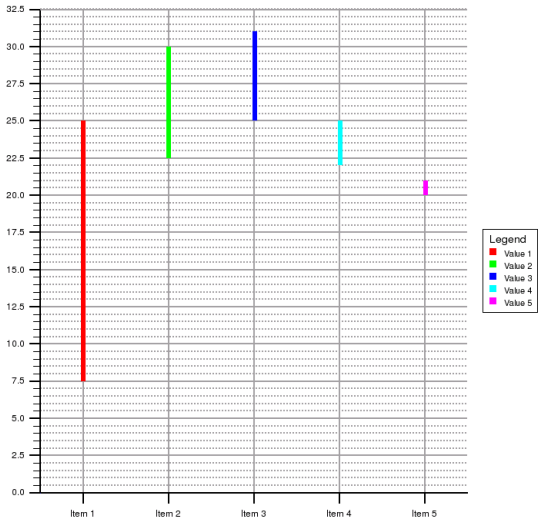
be omitted for this series, the same of course applies to any Data Value Texts written next to the line ends.

Simple High/Low Charts

Tip

Simple high/low charts (not showing open nor close values) can be useful to give an overview about an item's largest and smallest values per time-segment. You can use such a chart if your values beginning and end values are not interesting because the important information is represented by its minima and maxima: e.g. this might be the highest/lowest temperature reached in a year where your chart might show the evolution of these values during a span of some 100 years.

Figure 3.21. A Simple High/Low Chart



by default draws normal high/low charts without showing any open or close marks when in High/Low Chart mode so no methods need to be called to get such charts, however after having used your `KDChartParams` to display another high/low chart type you can call `setHiLoChartSubType( KDChartParams::HiLoNormal )` to return to the default subtype (the name `HiLoSimple` is a synonym for `HiLoNormal` and was defined for your convenience).

Compile and run the tutorial file `kdchart_step01r.cpp` to see a normal high/low

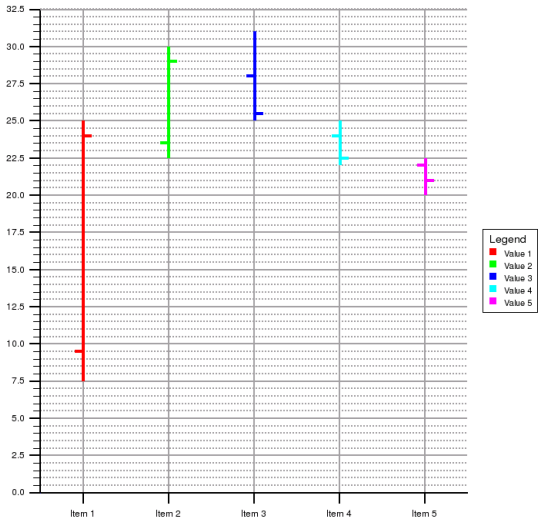
chart (see also Figure 3.21): a special modification there is a `setLineWidth( 6 )` used to get thick lines. We do this to make the high/low lines look better since by default this chart type shows no data labels and without such labels its lines are a bit difficult to see if only a few lines are shown, see the next but one chapter for a High/Low Chart with Data Labels.

Open/Close High/Low Charts

Tip

Extended open/close high/low charts provide a view on the start and end values as well as on the maximal and minimal values of a specific item per time-segment. This kind of chart is useful if your item's start and end value is of special importance too, e.g. typically such charts are used to visualize the evolution of per-day figures in a financial share value diagram displaying one or several weeks.

Figure 3.22. An Open/Close High/Low Chart



To make draw the open and close lines too you call the `KDChartParams` function `setHiLoChartSubType( KDChartParams::HiLoOpenClose )`. In case you are not interested in the open value (e.g. because this would always be identical to the respective previous close value) just use `HiLoClose` instead.

Compile and run the tutorial file `kdchart_step01s.cpp` to see an open/close high/

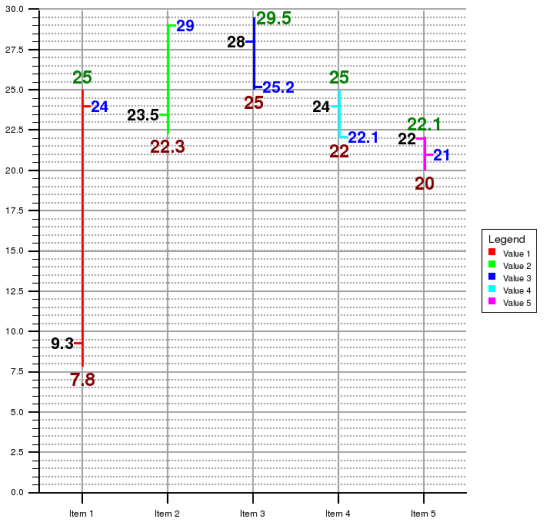
low chart—modified by an extra `setLineWidth( 4 )` function call used to get thick lines (see also Figure 3.22). We do this to make the lines look better since this chart type by default shows no data labels and without such labels its lines are a bit difficult to see if only a few lines are shown, see the next chapter for a High/Low Chart with Data Labels.

High/Low Charts with Custom Data Labels

Tip

Adding data labels to your High/Low chart can be useful if your chart is displaying a rather limited number items: otherwise it might be a better idea to draw the lines without showing the data labels but show a few labels instead, using Cell specific Properties for some of the cells—or you might add one or several separate Custom Text Boxes anchored at the most significant points.

Figure 3.23. A High/Low Chart With Custom Data Labels



To make print the value texts next to the respective line ends you call one or several of the following `KDChartParams` functions:

- `setHiLoChartPrintHighValues()` prints *high* values on top of the lines.
- `setHiLoChartPrintCloseValues()` prints *low* values below the lines.

- `setHiLoChartPrintOpenValues()` prints *open* values at the left of the lines.
- `setHiLoChartPrintCloseValues()` prints *close* values at the right of the lines.

Each of the statistical value types can be displayed using different font settings, e.g. you might specify the high value texts like this:

```
QFont fontHigh( "helvetica", 1, QFont::Bold );
QColor colorHigh( Qt::darkBlue );
p.setHiLoChartPrintHighValues(  true, &fontHigh, 38, &colorHigh );
```

The third parameter of this function (`size`) is ignored if not greater than zero, by setting it you make calculate the font size dynamically: based upon the widget size. Otherwise the fixed font size will be used that is set for the `font` parameter, see the Fonts chapter for more information on this.

Compile and run the tutorial file `kdchart_step01t.cpp` to see an open/close high/low chart featuring all possible value texts using custom font settings (see also Figure 3.23).

Note

The upper line's high value (`29.5`) shows a special feature of KD Chart's high/low charts:

If a line reaches too near to the data area's top edge the high value's text is not drawn above the line but at the right of its top—the same applies to the low value's text which would be shown at the left side of the line instead of touching the data area's lower border.

If you don't like this feature just make sure there is enough room for the value texts by setting the lower and/or the upper limit of the ordinate axis as shown in the Limiting/Reverting the Axes chapter.

Box&Whisker Charts

This chart is called box&whisker because of its look: a box in the middle and two lines looking like whiskers on each side. While the box surrounds the center half of your spreading values, the upper and lower whisker ends are framing all or most of the values, except from a few outliers indicated separately.

Box&whisker charts provide detailed descriptive statistics of a variable: The height of the box shows how close together the main part of your values are and the length of the whiskers indicate how far the other values spread.

Tip

Since box&whisker charts give an overview of your values distribution (plus their mean and average value) they can be used for a quick estimate without looking into your statistical tables. An example might be a comparison of the number of failures in a device, perhaps three datasets of some 30 computer chips each to compare their errors at three different temperatures. Each of the data cells containing the number of failures shown by one chip your chart would show three boxes: you might expect both a lower error number (with the box being drawn nearer to the abscissa axis) and less variation (smaller box and whiskers) at lower temperature.

Box&whisker charts are the most sophisticated ones of KD Chart's statistical chart types: unlike the simple High/Low Charts *they calculate* each dataset's statistical values themselves.

You can activate the box&whisker chart mode by calling the `KDChartParams` function `setChartType( KDChartParams::BoxWhisker )`.

What happens to missing data cells?

Missing values (empty data cells) will be ignored since KD Chart is able to calculate box&whisker chart statistics if there is at least *one* value per dataset, otherwise nothing will be drawn for this dataset: there will be a gap between the neighboring boxes but the invisible box will still be mentioned in the Legend indicating you that some data are missing here.

Note

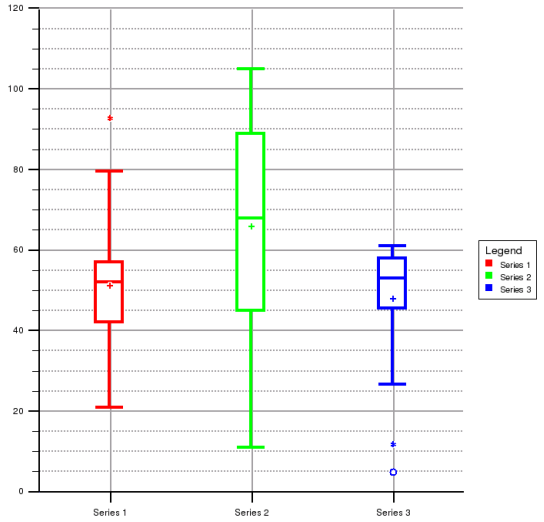
While the line colors of a box&whisker chart can be specified by the usual `KDChartParams::setDataColor()` the line *thickness* for these charts is calculated dynamically based upon the horizontal space available for one dataset. You can increase this line width by calling the `KDChartParams` function `setLineWidth()` which acts as a multiplier here: a width of `2` would make the lines twice as thick as normal.

Normal Box&Whisker Charts

Tip

Normal box&whisker charts are used to get a quick overview about your spreading data and see the show outliers.

Figure 3.24. A Normal Box&Whisker Chart



KD Chart 1.0 knows only one type of box&whisker charts, calling `setBWChartSubType( KDChartParams::BWNormal )` is possible but not necessary.

Compile and run the tutorial file `kdchart_step01u.cpp` to see a normal box&whisker chart (see also Figure 3.24): series #1 shows an *outlier* located in the *upper inner fence* marked by a star, series #3 has an *outlier* and an *extreme*: the later (a circle) is even below the *lower outer fence*. Lower fences are defined like this (upper ones accordingly):

The "Lower Inner Fence" is the interval between the following positions:

- the lower hinge (bottom quartile)
- the lower hinge minus 1.5 times the IQR (Interquartile Range)

The "Lower Outer Fence" is the interval between the following positions:

- the lower hinge minus 1.5 times the IQR
- the lower hinge minus 3 times the IQR

Note

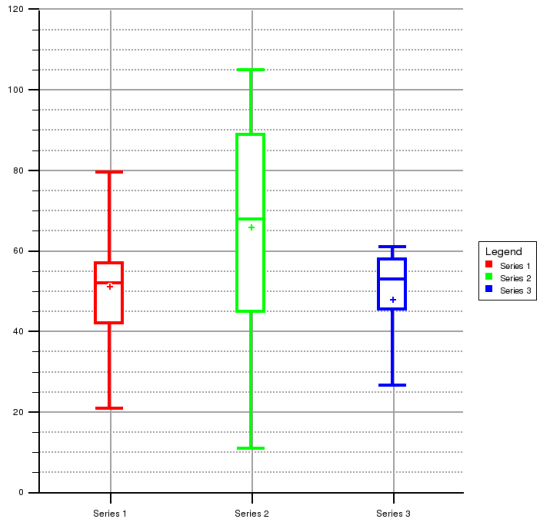
In case the build-in default factors (1.5 and 3.0) for calculation of the inner and outer fences do not match your needs you can adjust them by calling the `KDChartParams` function `setBWChartFences()`. It is possible to use different values for all of the four fence factors.

Box&Whisker Charts without Outliers

Tip

If outlier values would dilute the message of your chart or simply be visually unpleasant, you can turn them off as described here.

Figure 3.25. A Box&Whisker Chart Without Outliers



To suppress drawing of any markers for *outliers* or *extremes* just set their size to zero by calling `setBWChartOutValMarkerSize( 0 )` as demonstrated by the tutorial file `kdchart_step01v.cpp`—of course you could also use a value different from zero to set them to a specific size: `setBWChartOutValMarkerSize( -50 )` would increase their dynamic size by factor two since -25 is the default setting (a quarter of the box width), while `setBWChartOutValMarkerSize( 15 )` would set them to a fixed size of 15pt ignoring the size of the widget.

Note

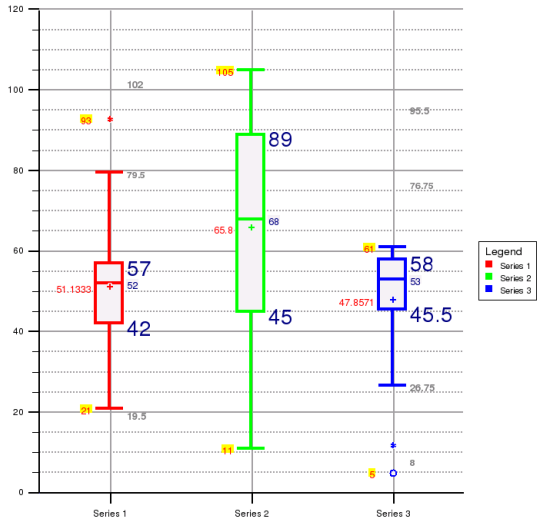
This special mode might require some additional explanation to tell your users that the outliers and extremes are suppressed: otherwise the chart might be interpreted wrongly since it looks like *there are no* outliers but all values are within the inner fences or the box area.

Box&Whisker Charts with Statistics

Tip

By showing statistical values in your box&whisker charts, you can make your charts even more expressive - at the possible expense of readability.

Figure 3.26. A Box&Whisker Chart With Statistical Values



Box&whisker charts can print up to ten different statistical figures next to the respective boxes—each of which can be shown using specific fonts, text color, and background colors.

Printing of a statistical value is enabled by passing one of the `KDChart-Params::BWStatVal` enum's descriptive names (`UpperOuterFence`, `UpperInner-Fence`, `Quartile3`, `Median`, `Quartile1`, `LowerInnerFence`, `LowerOuterFence`, `MaxValue`, `MeanValue`, `MinValue`) to the following function:

```
void KDChartParams::setBWChartPrintStatistics( BWStatVal statValue,
    bool active,
    QFont* font = 0,
    int size = 24,
    QColor* color = 0,
    QBrush* brush = 0 );
```

Parameters:

`statValue`	one of the enum values listed above
`active`	set to true to have the `statValue` printed using either the default fond and color or the settings specified by the following parameters
`font`	if not zero the font will be used for this statistical value
`size`	if not zero this value will be interpreted as percent of the actual box width: font size will be calculated dynamically then instead of using the `font` parameter's fixed size
`color`	the text color of this statistical value
`brush`	the color of the background of this statistical value, if set to Qt::NoBrush the background is *not* erased before the text is printed

Compile and run the tutorial file `kdchart_step01w.cpp` to see a box&whisker chart featuring all possible statistical value texts using custom font settings and a special background color for the boxes (see also Figure 3.26).

Note

No fence values are printed for the middle series in our sample (Figure 3.26) because they are outside of the chart's data area. This can occur if all values of a series are in the inner fences and the box. Since in this case there is no need to show the fence values the range of the ordinate axis is *not* extended but the chart uses the available space to have more room for displaying the other datasets which otherwise would have less vertical space to draw their boxes and whiskers.

Polar Charts

Tip

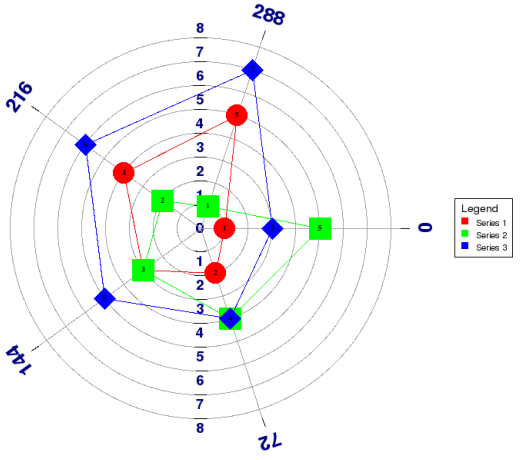
Polar charts got their name from displaying "polar coordinates" instead of Cartesian coordinates. Currently only normalized polar charts can be shown: all values advance by the same number of polar degrees and there is no way to specify a data cell's angle individually. While this is ideal for some situations it is not possible to display true world map data like this since you can not specify each cell's rotation angle. Transforming your coordinates to the Cartesian system and using a Point Chart might be a solution in such cases.

What happens to missing data cells?

Missing data cells in a polar charts are just skipped: the respective point is missing in its dataset's circular line.

You can activate the polar chart mode by calling the `KDChartParams` function `setChartType( KDChartParams::Polar )`.

Figure 3.27. A Normal Polar Chart



Just like the Line Charts to which they can be compared the polar chart type is divided into three sub types which can be activated calling `setPolarChartSubType( KDChartParams::PolarNormal )` or `PolarStacked` or `PolarPercent` respectively.

Compile and run the tutorial file `kdchart_step01x.cpp` to see a normal polar chart (see also Figure 3.27)—you can try the other sub types by removing the comment token (`//`) from the beginning of the respective source code line.

Note

Data Value Texts are shown by default in polar charts even if drawing of the markers is suppressed by `setPolarMarker( false )`, you can hide them by calling the `KDChartParams` function `setPrintDataValues( false )`.

What's next

In the next chapter you will learn how to modify some layout properties of your chart: colors, fonts, ...

Chapter 4. Customizing Your Chart

This chapter presents some further steps you might want to take for configuring your chart's layout. It is about adjusting the color settings to your needs, setting up the fonts to be used, specifying how the lines and/or markers are to be painted and adding or modifying three-dimensional effects.

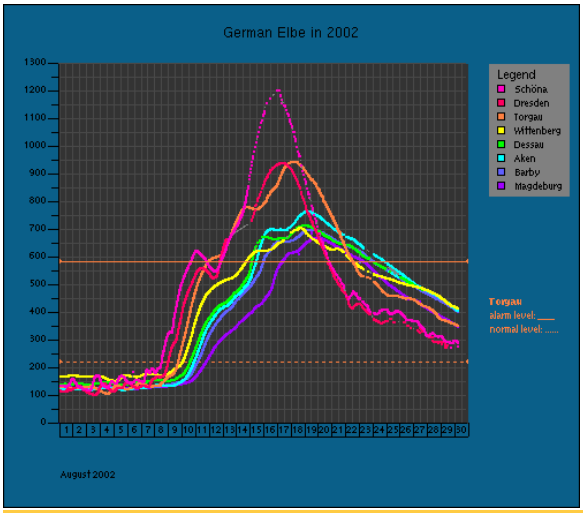
KD Chart offers a broad bandwidth of configuration options which are not fully covered by this chapter, for details please consult the Reference Documentation coming with your KD Chart package.

Colors

To specify a color in KD Chart you normally just pass a `QColor` to the respective function. Examples are the useful `KDChartParams` methods `setHeaderFooterColor()` (see the chapter on Headers and Footers) or `setDataColor()` used to specify the colors of one dataset's bars (or lines, pie slices, etc.)

Besides from this basic methods there is an extra `KDChartParams` function you might find nice to set a special series of colors to the datasets #0..#15 looking best on dark gray of black background: `setDataRainbowColors()`. See the following screen shot for an example:

Figure 4.1. A Chart with Rainbow Colors

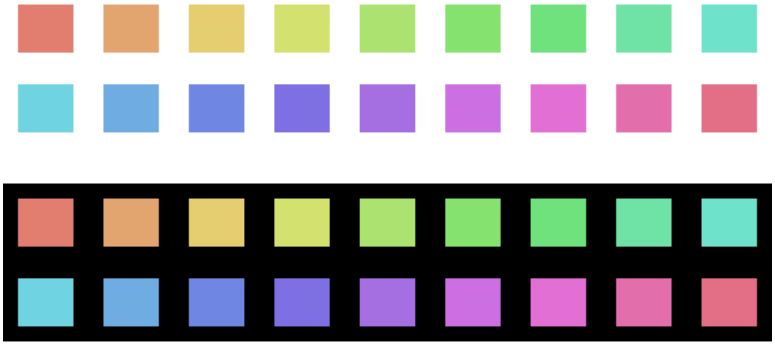


The rainbow colors are also shown by the color sets table Figure 4.3.

Another convenience function you might find useful in all situations where you want to print texts onto your data representations (e.g. when showing Data Value Texts texts on an Area Chart) is called `setDataSubduedColors()`.

By default this function sets eighteen subdued dataset colors in a way that the colors of neighboring datasets can be easily distinguished from each other (see the color sets table Figure 4.3). In case you prefer a *continuous* color effect just set the `ordered` parameter of this function to true, the colors would then appear in the order shown by the following illustration (see Figure 4.2):

Figure 4.2. The Subdued Color Set (ordered)



Have a look at the chapter on Area Charts In Subdued Colors featuring a sample area chart in subdued colors (see Figure 3.20).

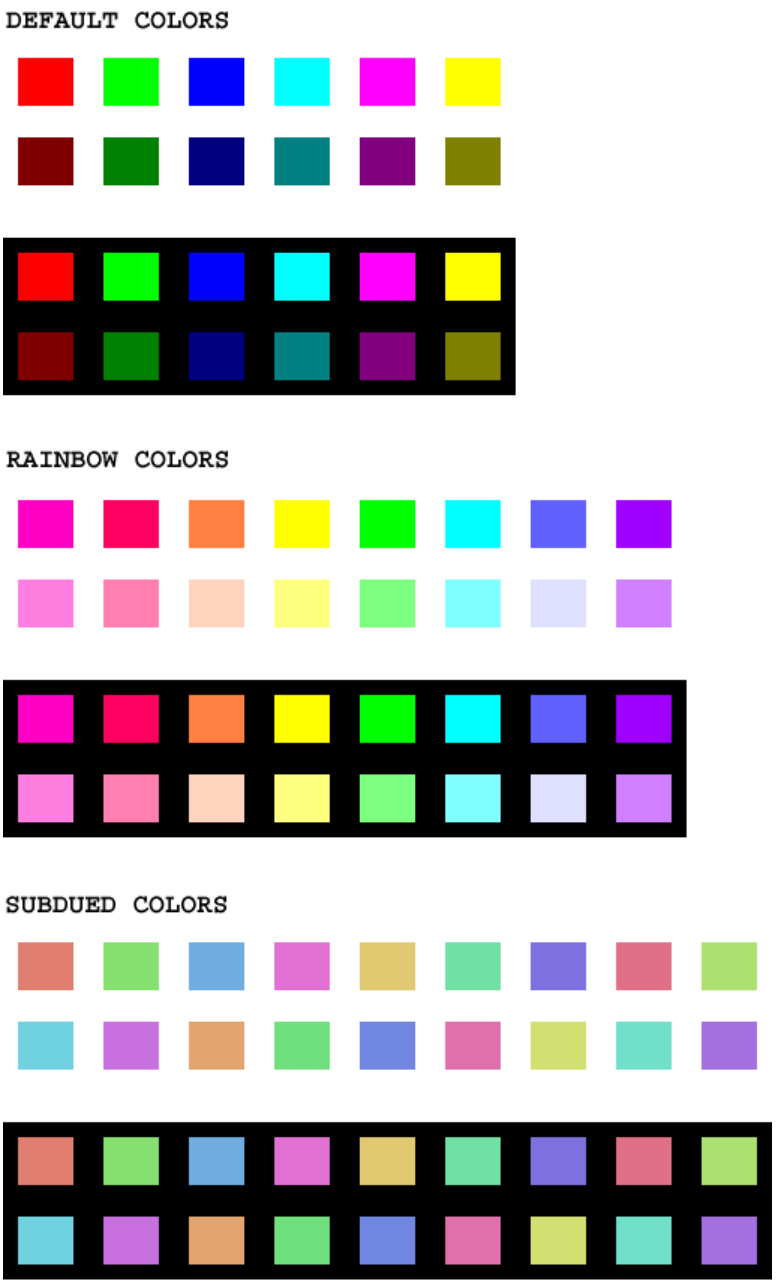
After changing the dataset colors you can restore the original settings by calling `setDataDefaultColors()`. This activates the default data colors (see Figure 4.3).

Tip

The Reference Documentation that is part of your KD Chart package provides additional information on the options for specifying the various colors used in your charts, e.g. the colors of the small Data Value Texts that can be written next to the bars (or line markers, etc.) can be adjusted by the `KDChartParams` function `setPrintDataValuesColor()` while you can change the color of your chart's horizontal grid lines by `KDChartAxisParams::setAxisGridColor()` for the left axis. It is even possible to fine tune the brightness of the shadowed areas shown when your bar or line chart is displayed in tree-dimensional look: see the Note at this chapter's end for details.

The following illustration (see Figure 4.3) shows how the colors look like on both light and dark background:

Figure 4.3. KD Chart's Color Sets



Of course, the different number of colors provided by the color sets shown above (see Figure 4.3) does *not* mean a limitation in the number of datasets that can be displayed: when more colors are needed than available KD Chart just starts from the respective color set's beginning and repeats the colors.

Note

By setting the dataset color you get the respective shading colors calculated by KD Chart: the shade colors of the three-dimensional bars in a Bar Chart (enabled by `setThreeDBars(true)`) or the respective colors for three-dimensional lines in a Line Chart (see `setThreeDLines(true)`) will be a bit darker than their base colors specified by calling `setDataColor()`.

This is done automatically unless you advise KD Chart to use the same colors by calling the `setThreeDBarsShadowColors( false )` function—or you might call `setShadowBrightnessFactor()` to fine-tune the shadow tones alternatively: by specifying values greater 1.0 you can lighten the shadows until the most intensive brightness is reached, while values smaller than 1.0 will darken the shadows until the respective colors are black.

Fonts

Font handling in KD Chart is done quite similar to the way you might be used from other Qt programs: just pass a `QFont` to the respective functions to specify the font settings. Examples are the often used `KDChartParams` methods `setHeaderFooterFont()` (see the chapter on Headers and Footers) or `setAxisLabelsFont()` (see Axis Label Font).

However there is one special thing about how KD Chart processes your font information: by default it uses *relative font sizes* where ever possible. Font sizes are adjusted to changing widget size to make your chart look good even when the user stretches or zooms her program window. Of course, this feature can be disabled: methods used for specifying a font normally also contain a parameter allowing to set the font size to a fixed or a relative value as desired.

Note

Relative sizes in KD Chart normally are calculated based upon the smaller one of the drawing area's extensions: so the font sizes will *not* change if the width of a chart is further increased which already was wider than heigh before. This is especially useful if you embed your `KDChartWidget` into a scrollable area. This might be a line chart displaying a large number of values shown in a horizontally scrolled part of your program: you would not want your font sizes to be adjusted to the large width.

Another useful configuration option available for many of KD Chart's font settings is *text rotation at will*: e.g. Data Value Texts or Axis Labels can be rotated using any angle—with abscissa axis labels even trying to rotate automatically if the available horizontal space is too narrow (of course this behavior can be disabled).

Some of the chart types show their Data Value Texts rotated by 45 degrees by default, e.g. the Bar Charts do this to make their texts fit into the available space trying to minimize overwriting of neighboring bars without forcing the user to turn the head by 90 degrees. You can disable this feature by calling the `KDChartParams` method `setPrintDataValues()` allowing (among other things) to specify the rotation angles for negative values and for positive values—for details see the Reference Documentation that came with your KD Chart package.

Using rotated texts has but one drawback: they are sometimes not looking good on screen, especially when a small font size is used. For compensation KD Chart uses its own output optimization method making rotated text look slightly better on screen but unfortunately this must be activated when your chart is to be printed since the optimized output only looks good at low resolutions: a printer's higher resolution is perfect for using Qt's default way of displaying rotated text. For your convenience there is a special `KDChartWidget::print()` method taking care for (re)setting the output optimization flag, see the notes on Printing Your Chart for detailed explanation.

Lines And Markers

In addition to offering the full bandwidth of the common `QPen` settings (line color, style, etc.) KD Chart also provides an elegant way of automatically adjusting the widths of the different lines to your widget's size: by default the axis lines, grid lines, outlines in a Box&Whisker Chart, etc., are drawn using different pen widths according to the size of the actual drawing area so there is no need to worry about how they might look when the widget is shrinked or enlarged considerably. This feature is called *dynamic size*.

Please consult KD Chart's Reference Documentation providing details on the several line specific functions—especially for the classes `KDChartParams` and `KDChartAxisParams`.

As shown in the chapters on Line Charts with Markers and Cell specific Properties KD Chart can also show various types of markers for drawing attention to a specific data point. There are two kinds of markers: the normal point markers used in Line Charts (or in Point Charts, resp.) and the *extra markers* which can be used in Bar Charts or in Line Charts together or instead of the *extra lines* as described below in the chapter on Cell specific Properties (see also the end markers of the two horizontal lines in Figure 4.1).

All of these markers by default share the same *dynamic size* feature but of course you can disable it at any time if you prefer using a fixed marker size: just fill the respective `QSize` parameter's width and height properties with positive values instead of the default negative figures.

Note

Negative size or width values in KD Chart normally are interpreted as *dynamic values*: the real value then is calculated based upon this dynamic value and the current widget size.

Three-dimensional Effects

Currently featuring three-dimensional look for Bar Charts, Line Charts and Pie Charts KD Chart provides the following `KDChartParams` functions to enable three-D look for the respective chart type: `setThreeDBars( true )`, `setThreeDLine( true )`, `setThreeDPies( true )`.

Shadowing

By default KD Chart uses shadow colors for displaying the parts of a three-D surface that are not oriented directly towards the user.

You can deactivate shadowing by calling the `KDChartParams` function `setThreeDShadowColors( false )`: the normal colors will then be used for all parts of the surface.

To adjust the brightness of the shadowed parts you can call the `KDChartParams` function `setShadowBrightnessFactor( double )`: choosing values greater 1.0 you can lighten the shadows until the most intensive brightness is reached, while values smaller than 1.0 will darken the shadows until the respective colors are black.

It is also possible to use KD Chart's shadow color calculation function separately, e.g. for using the shadow colors shown in the charts for your program's other widgets too. Just call `KDChartParams` function `calculateShadowColors()` to let KD Chart calculate two shadow values based upon a color specified by you.

Alternatively or in addition to adjusting the shadow colors you can specify which *pattern* is to be used for the shaded parts by calling the `KDChartParams` function `setShadowPattern()`, the default is a simple `Qt::SolidPattern`.

Depth or Height

To change the default three-dimensional look and reduce or increase the depth of a bar's three-D effect you can call the `KDChartParams` function `setThreeDBarDepth()`: this will make your bars look more flat or deeper.

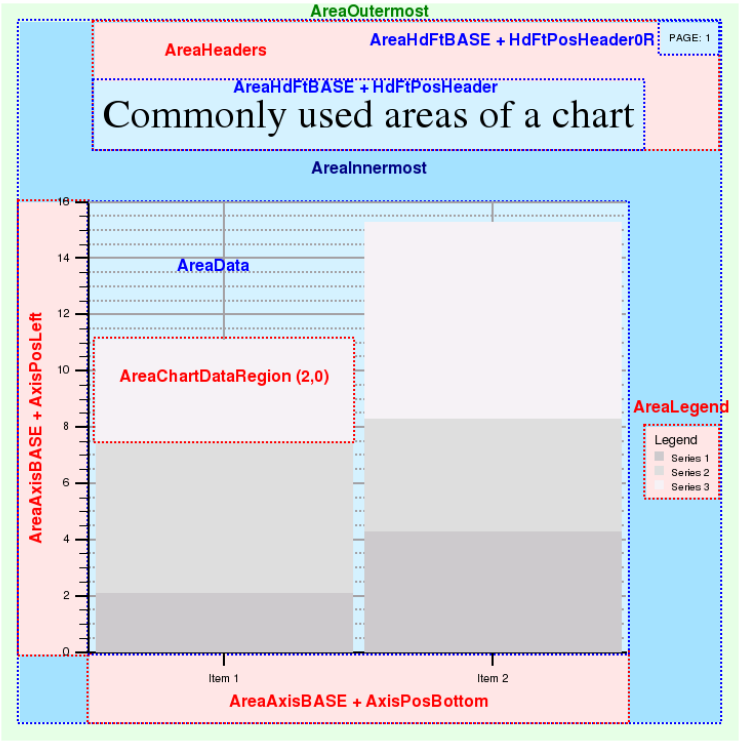
For your line chart the same is achieved by use the `setThreeDLineDepth()` function. Pie charts are viewed from above so the function is called `setThreeDPieHeight()`.

Chapter 5. Areas of a Chart

KD Chart's output is structured using the *Area* concept, an area is a (normally) contiguous and (most times) rectangular part of the screen containing one or several drawing elements that belong together in a logical way.

The following illustration (see Figure 5.1) shows the positions and names of the most commonly used areas in a chart:

Figure 5.1. Areas of a Chart



Compile and run the tutorial file `kdchart_step03.cpp` to see how this chart was created, read the explanations given below and study the chapters on Area Frames and on Adding Custom Text Boxes for details on the techniques used in this program.

These are the names defined by the `KDChartEnums::AreaName` enum:

`AreaData`	Covering the data area

`AreaAxes`	Covering the axes but leaving out the data area, so this is a non-contiguous area.
`AreaDataAxes`	Covering the data and axes areas.
`AreaLegend`	Covering the legend area
`AreaDataAxesLegend`	Covering the data, axes, and legend areas.
`AreaHeaders`	Covering the entire header area.
`AreaFooters`	Covering the entire footer area.
`AreaDataAxesLegendHeadersFooters`	AreaDataAxesLegendHeadersFooters: covering the data, axes, legend, header, and footer area.
`AreaInnermost`	Covering the complete drawing area but not covering the global left/top/right/bottom leading: you can specify the gaps between this area and the `AreaOutermost` by calling the `KDChartParams` function `setGlobalLeading()`.
`AreaOutermost`	Covering the complete drawing area including the global left/top/right/bottom leading
`AreaChartDataRegion`	Covering the area used to display one data entry (i.e. a point, a bar, a segment of a line, a pie slice, etc.) The respective data coordinates are specified by additional parameters, e.g. when calling the `KDChartCustomBox` constructor where you can use the `dataRow`, `dataCol`, `data3rd` parameters to indicate the respective data cell. Data region areas can also be accessed via `KDChartCustomBox::setDataRegionFrame()` in case you want to have a data entry by a (colored) line for special emphasis.
`AreaCustomBoxes`	Does not specify one area but rather many small areas covering all the custom boxes that you might have added to the chart, this is useful in case you want to have some default frame settings to be used for all custom boxes not having frame settings of their own.
`AreaAxisBASE`	Does not specify one area, must be used together with an Axis number to determine a specific axis area, e.g. the area of the left ordinate axis can be addressed by `AreaAxisBASE + AxisPosLeft`.
`AreaHdFtBASE`	Does not specify one area, must be used together with the number of an header (or footer, resp.) for specifying a specific header/footer area, e.g. the area of the main header can be addressed by `AreaHdFtBASE + HdFt-`

	`PosHeader`.
`AreaCustomBoxes-BASE`	Does not specify one area, must be used together with the number of a custom box you have added to the chart, e.g. if `boxIdx1` contains the ID of a `KDChartCustomBox` the area of this box can be addressed by `AreaCustBoxBASE + boxIdx1`.

Configuring the standard areas (or newly created ones when Adding your own Text Boxes) is an easy task. Headers, footers, and legend can be configured according to the user needs. Frames with colored lines, showing a nice background or presenting a picture can be easily added to the areas—the `KDChart_Presentation` program's bottom sample "nice full featured chart" might give you an impression of the many possibilities.

To add some precision concerning frames please note that they technically are no areas but they are always attached to one of the chart's area mentioned above. Frames not connected to a standard area must be assigned to a `KDChartCustomBox` object. We will describe how to create Frames in the last section of this chapter, for more on custom boxes, see the chapter Adding Custom Text Boxes.

We will now guide you through the procedure to follow in order to configure the different areas of your chart.

Headers and Footers

To discover the many configuration options for Headers and Footers, consult the documentation for the class `KDChartParams`.

Let us write together a small application, for you to have a reference about how to make these areas visible and how to configure them.

To create our own chart, we follow the procedure described in Chapter 2, *Three Steps to Your Chart* at the beginning of this manual.

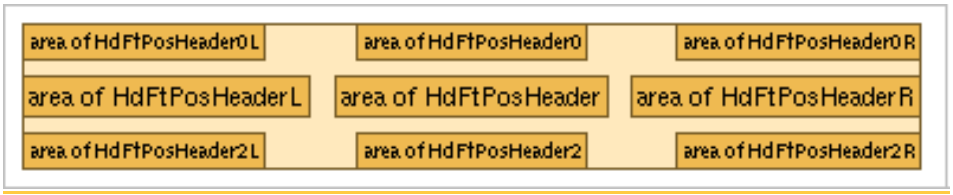
1. Create a `KDChartParams` object that contains all the settings of your chart.
2. Create a `KDChartTableData` object and fill it with the data you want to visualize in a chart
3. Create a `KDChartWidget` object and pass the `KDChartTableData` and `KDChartParams` to it.

We now add a header to our chart and configure the font to be displayed, by calling the `setHeaderFooterText()`, `setHeaderFooterFont()` and `setHeaderFooterColor()` functions of our `KDChartParams` object:

```
p.setHeaderFooterText( KDChartParams::HdFtPosHeader,
                       "My Bar chart" );

p.setHeaderFooterFont( KDChartParams::HdFtPosHeader,
                       QFont( "helvetica",24, QFont::Bold,true ),
                       true,
                       40 );

p.setHeaderFooterColor( KDChartParams::HdFtPosHeader,
                        Qt::blue );
```

`setHeaderFooterText()` specifies the text to be displayed in the header (or footer, resp.) section indicated by its first parameter. KD Chart offers up to nine header areas (and nine footers, resp.) and—although you most likely will not use all of them in one chart—see Figure 5.2 for their default positions and alignment:

Figure 5.2. Header Positions



For all about the different header or footer sections consult the explanations given on the `KDChartParams` enum `HdFtPos`, see section "Header and footer methods" in KD Chart's reference documentation.

To add a footer area just follow the procedure described above, replacing the position parameter of `setHeaderFooterText()` by the respective footer's enumeration value as declared by `KDChartParams::HdFtPos`.

Have a look at the tutorial file `kdchart_step04.cpp` to see our simple code at this stage.

The implementation file uses an instance of the `TutorialWidget` as main widget, as explained in Chapter 2, *Three Steps to Your Chart*.

To build our little sample program we add a project file (`*.pro`) for Qt's `qmake` tool:

```
1 HEADERS     = kdchart_step04.h
2 SOURCES     = kdchart_step04.cpp
3 TARGET      = step04
4
```

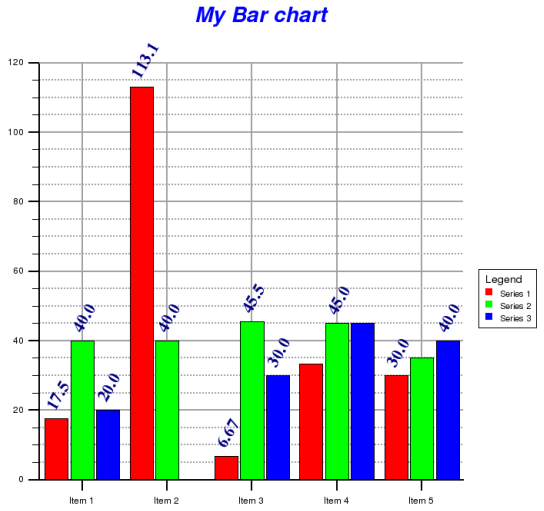
Next we run the usual commands.

1. `qmake -o Makefile myapp.pro`

2. `make` (or `nmake` for Windows)

We can now view the result of our sample code:

Figure 5.3. A Chart With A Header



Learn more about the many options to configure Headers and Footers by consulting the Reference Documentation of the `KDChartParams` class, see the `doc/reference/` directory of your KD Chart package.

Legends

Looking at the widget created in the Headers and Footers paragraph above, you can have noticed that KD Chart automatically added a legend to our chart. There are many ways to configure the legend and adapt its display to your needs.

Detailed information on the legend methods is given with the `KDChartParams` class in KD Chart's Reference Documentation.

We will now quickly overview those methods, describe how to set the texts and configure the legend display (position, text font and color, spacing, etc.)

We will then put in practice some of this, by adding a few lines of code to the application developed in the previous paragraph.

The legend configuration methods allow you to:

- Specify the legend position and orientation.
- Define the legend source (telling KD Chart where to get the legend texts from).
- Modify the legend texts.
- Configure the legend text and title (font, color..).
- Configure the legend display.

Additionally you can enframe your legend and/or have a colored background (or an image, resp.) behind it: the legend area is referred to by `AreaLegend`, see the description of `KDChartParams::setSimpleFrame()` in the Area Frames chapter for details on calling this method.

The Legend Source

There are different ways to pass the legend texts to KD Chart.

- Manual: Text is set with the `KDChartParams::setLegentText()` function.
- FirstColumn: The values stored in the first column of every row (== first cell of every dataset) will be used.
- Automatic: Will first try to use values from the first column, if there are no string values there, will try to use values set manually, and finally if there are no values set manually either, will resolve to standard texts like "Series1", "Series2", etc. (automatic mode is the default).

Let us enhance our sample application by defining the legend title and texts:

- Set the legend source to Manual mode.
- Specify the legend title.
- Specify the legend texts.

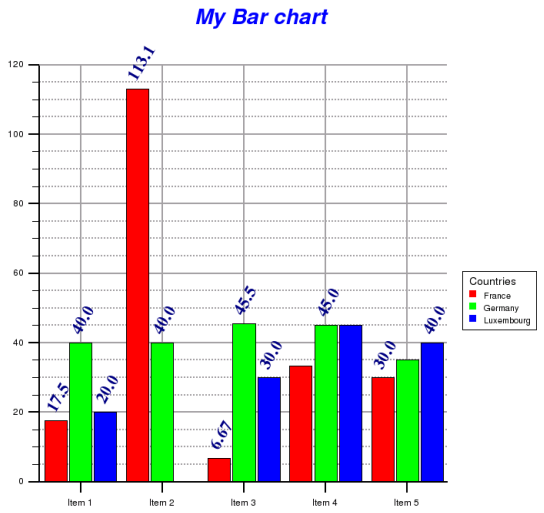
Add these code lines to your implementation file, p being your `KDChartParams` object:

```
p.setLegendSource( KDChartParams::LegendManual );
p.setLegendTitleText( "Countries" )
p.setLegendText( 0, "France" );
p.setLegendText( 1, "Germany" );
p.setLegendText( 2, "Luxembourg" );
```

The first parameter of `KDChartParams::setLegendText()` represent the dataset for which to set the legend text.

Compile and run the tutorial file `kdchart_step04a.cpp` to see a our sample program at this stage, your chart should look like in Figure 5.4.

Figure 5.4. Adding a Custom Legend



The Legend Position

KD Chart by default shows the legend with its standard texts (Series 1, Series 2, etc.) and title (Legend), at the right side of the chart. As for the text and the title, the position of the Legend can be configured easily.

Belonging to the inner parts of the chart the legend is always positioned between the headers area and the footers area and outside of the space occupied by the axis areas and by the data area.

The possible legend positions are defined in the `LegendPosition` enum of the `KDChartParams` class. Besides from specifying the place where the legend will go these values also control the way how the other inner parts of the chart (the axes and data area) make room for the legend:

`NoLegend`	No legend is displayed.

`LegendTop`	The legend is horizontally centered above the axes and data area which make room for it to the bottom.
`LegendBottom`	The legend is horizontally centered below the axes and data area which make room for it to the top.
`LegendLeft`	The legend is vertically centered at the left of the axes and data area which make room for it to the right.
`LegendRight`	The legend is vertically centered at the right of the axes and data area which make room for it to the left.
`LegendTopLeft`	The legend is near the upper left corner of the axes and data area which make room to the bottom and to the right.
`LegendTopLeftTop`	The legend is near the upper left corner of the axes and data area which make room to the bottom only.
`LegendTopLeftLeft`	The legend is near the upper left corner of the axes and data area which make room to the right only.
`LegendTopRight`	The legend is near the upper right corner of the axes and data area which make room to the bottom and to the left.
`LegendTopRightTop`	The legend is near the upper right corner of the axes and data area which make room to the bottom only.
`LegendTo-pRightRight`	The legend is near the upper right corner of the axes and data area which make room to the left only.
`LegendBottomLeft`	The legend is near the lower left corner of the axes and data area which make room to the top and to the right.
`LegendBottomLeft-Bottom`	The legend is near the lower left corner of the axes and data area which make room to the top only.
`LegendBottomLef-tLeft`	The legend is near the lower left corner of the axes and data area which make room to the right only.
`LegendBottomRight`	The legend is near the lower right corner of the axes and data area which make room to the top and to the left.
`LegendBottomRight-Bottom`	The legend is near the lower right corner of the axes and data area which make room to the top only.
`LegendBottom-RightRight`	The legend is near the lower right corner of the axes and data area which make room to the left only.

To specify the legend position call the `KDChartParams` function `setLegendPosition()`: it has only one parameter, which is the position value described above—using

`NoLegend` would hide the legend.

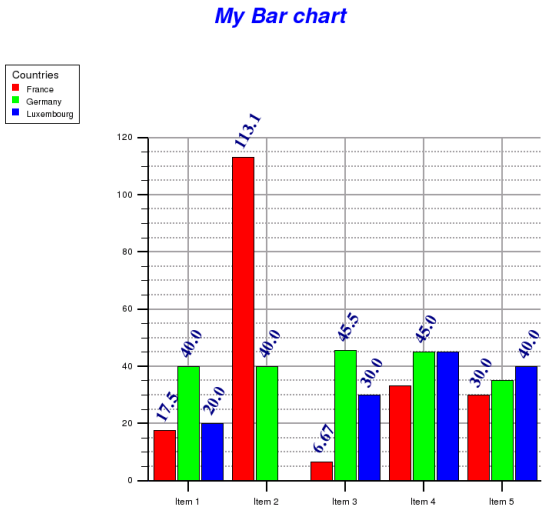
To put this in practice, we will now change the position of our legend by adding a piece of code to our example application.

Add the following line of code to your implementation file, p being your `KDChart-Params` object:

```
p.setLegendPosition(KDChartParams::LegendTopLeft);
```

Compile and run the tutorial file `kdchart_step04b.cpp` to see a our sample program at this stage, your chart should look like in Figure 5.5.

Figure 5.5. Changing The Legend Position



KD Chart offers so many ways of positioning for your chart's legend, just test them using the different enum values and find the layout that fit best your needs.

The Legend Orientation

KD Chart by default shows the legend title and the legend entries below each other, this is called *vertical orientation*.

To have the texts printed in one or several horizontal rows, call `KDChart-Params::setLegendOrientation( Qt::Horizontal )`.

Custom Legends

KD Chart allows you to completely ignore the automatic legend feature and specify your own custom legends by providing the static `KDChartPainter` method `drawMarker()`. Using this you can draw a marker of the desired size, style, color into a `QPainter`.

Note

Understanding this requires advanced knowledge—best aquired by studying the chapter on Advanced Charting, therefor it is described in this chapter, where you will also find a sample program illustrating the techniques to be applied.

see: Chapter 6. Advanced Charting / Detached Custom Legends

Frames and Backgrounds

Using frames you can highlight some parts of your chart and use colors or backgrounds to make things more understandable or just present your chart in a nicer way. Frames allow you to add notes, pictures, indicators in a very flexible way. Have a look at the `KDChart_Presentation` example (in the `KDChart_Presentation` directory of your package) to get an idea of the usability of frames in real life: see the bottom sample `full featured chart`.

For a complete overview about frames read the class reference documentation of `KDChartParams` and search for "Frame".

Framing a Data Region

We will now go quickly through the methods provided by KD Chart to implement and use frames, and then write a few line of codes to put the theory in practice as well as to give you some reference examples.

Normally you add a frame (or a background, resp.) to one of the areas explained in Chapter 5, *Areas of a Chart* by calling `KDChartParams::setSimpleFrame()`. This function will be described later in the Area Frames chapter, but first let us have a short look at an extra method that is to be used for the *data regions* which can *not* be accessed by `setSimpleFrame()`: instead they require calling the `KDChartParams` function `setDataRegionFrame()` allowing us to add a frame around a cell representation (a bar, a line marker, etc.). The style, color and width of this line can be specified.

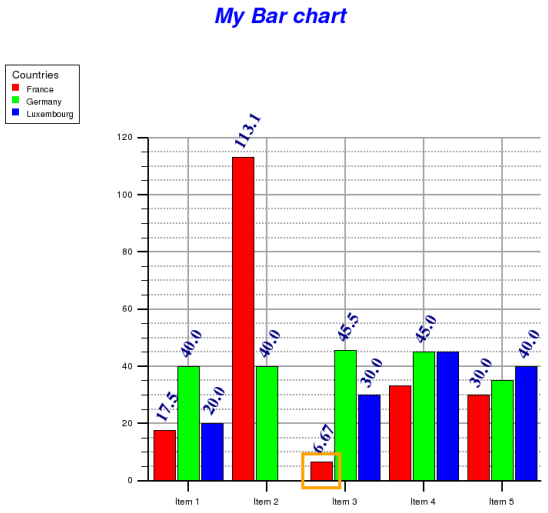
To see how this works let us add some code to the sample application we developed in the previous chapter. We will draw a colored line around one of the data bars to put the focus on it.

Load your implementation file and add the following piece of code, `p` being your `KDChartParams` object:

```
p.setDataRegionFrame( 0,2,0,
                      6,6,
                      true,true,
                      KDFrame::FrameFlat,
                      0,0,
                      QPen( QColor(0xff,0xa0,0),
                      4,
                      Qt::SolidLine ) );
```

Compile and run the tutorial file `kdchart_step04c.cpp` to see a our sample program at this stage, your chart should look like in Figure 5.6.

Figure 5.6. Framing A Single Bar



The first parameter of `KDChartParams::setDataRegionFrame()` is the dataset number, the second one indicates the item number. Have a look at the reference documentation of the `KDChartParams` class to learn more about this function.

We will now add some background to our chart, using a frame to make it nicer.

Area Frames

It is easy to add a frame and/or a background to one of the Areas of a Chart as KD Chart provides you with the powerful `KDChartParams` method `setSimpleFrame()`. Its parameters define:

`area`	The area to be surrounded by a frame.
`outerGap`	The distance between the frame and the surrounding part of the chart.
`innerGap`	The distance between the frame and the inner area.
`addFrameWidthToLayout`	For internal use—must be `true`.
`addFrameHeightToLayout`	For internal use—must be `true`.

`simpleFrame`	For internal use—must be KDFrame::FrameFlat.
`lineWidth`	For internal use—must be 1.
`midLineWidth`	For internal use—must be 0.
`pen`	The pen to be used for drawing the four frame border lines.
`background`	The brush to be used for painting the frame background.
`backPixmap`	The picture to be displayed as background image of the frame.
`backPixmapMode`	The way how backPixmap is to be modified to fill the frame(centered, scaled or stretched).
`shadowWidth`	For internal use—do not set this parameter or set it to 0.
`sunPos`	For internal use—do not set this parameter or set it to KD-Frame::CornerTopLeft.

Note that `setSimpleFrame()` lets us set a frame (and/or a background, resp.) to any area—with one exception: the several `AreaChartDataRegion` areas can *not* have an extra background, but you can either use a framing line (as shown above in Figure 5.6) or consult the chapter on Cell specific Properties providing some ideas how to highlight a data region.

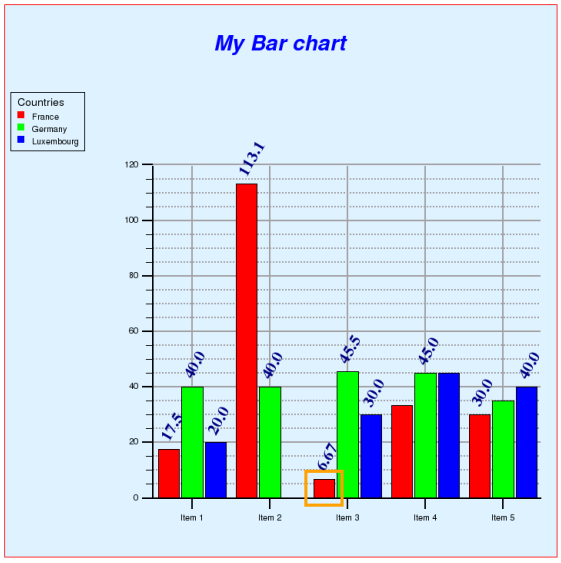
To frame our chart, show a nice background and have a small gap between its contents and the surroundings we add the following code to our implementation file:

```
p.setGlobalLeading(  7,7, 7,7 );
QColor myBackColor(218,240,255 );
p.setSimpleFrame( KDChartEnums::AreaInnermost,
                  0,0, 6,6,
                  true,
                  true,
                  KDFrame::FrameFlat,
                  1,
                  0,
                  QPen( Qt::red ),
                  QBrush( myBackColor ) );
```

Our chart now has a blue background and it is surrounded by a red line, also there is a gap between the chart and the frame, so the legend area does not touch the border.

Compile and run the tutorial file `kdchart_step04d.cpp` to see a our sample program at this stage, your chart should look like in Figure 5.7.

Figure 5.7. A Gap And a Frame Around The Chart



As mentioned above, the areas of your chart can be framed individually. You also have the possibility to paint a pixmap as background: either centered (using its original size) or scaled (preserving its x/y ratio) or stretched (covering the inner area completely).

Chapter 6. Advanced Charting

This part of the manual covers extended options for setting up your chart, explaining and showing you how to make the abscissa and the ordinate isometric to each other, how to configure your chart's axes until they meow, how to use several ordinate axes in the same chart, how to make two charts appear in one widget sharing the same abscissa axis, how to specify extra properties for some of your data cells, how to add custom text boxes to any area of your chart and how to enhance your chart by adding pictures in several ways.

While there normally might be *no need* to look into these chapters reading them could give you the power to get the best from KD Chart: charts that look *different* from Joe User's diagrams.

Axis Manipulation

KD Chart lets you access and manipulate the axis parameters of the chart by a large number of methods in the `KDChartParams` class and in the `KDChartAxisParams` class.

We will not review all the axis-related methods provided there and you should definitely look at the Reference Documentation for `KDChartParams` and `KDChartAxisParams`, to learn more about them and get an overview of their flexibility.

Using these functions you can modify the axis type, its area and size, the axis grid parameters, as well as the axis labels and text parameters and much more.

Note

All axis functions in the `KDChartParams` class are provided for your convenience only: they just wrap the respective `KDChartAxisParams` methods which often give you considerably more configuration options than the simple `KDChartParams` functions. An example are the Grid Functions: `KDChartParams::setAxisShowGrid()` lets you make the grid lines (in)visible while the `KDChartAxisParams` class offers detail functions like e.g. `setAxisGridStyle()` and `setAxisGridSubStyle()` allowing further specification.

The following schema (also demonstrated in the chapter on Isometric Coordinate Systems) gives you full access to the capabilities of the `KDChartAxisParams` class, supposing `p` is your chart's `KDChartParams` object:

```
// instantiate an axis parameters object
// calling the copy constructor for the left axis
KDChartAxisParams pa( p.axisParams( KDChartAxisParams::AxisPosLeft ) );

// manipulate the axis params, e.g. like this
pa.setAxisGridSubStyle( Qt::SolidLine );
```

```
// and more, see the Reference Documentation.

// make KD Chart use these settings for the left axis
p.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );
```

Keeping in mind the advanced options provided by the `KDChartAxisParams` class, let us now use some of the convenience methods provided by the `KDChartParams` class and write together a small sample application.

Titles

Before digging into the details of configuring your axes' labels let us have a quick look at two ways how to specify their *titles*.

1. Using the convenience methods provided by the `KDChartParams` class you can declare and set up an axis title with ease:

`setAxisTitle()`	declares the title string which can contain some simple rich text, just make sure that your string starts with "<qt>" and ends with "</qt>", an example might look like this: `"<qt>Revenue in <b>2003</b> <small>[ x 1000 Euro ]</small></qt>"`
`setAxisTitleColor()`	sets the color of this title text.
`setAxisTitleFont()`	specifies the font to be used.
`setAxisTitleFontRelSize()`	sets the dynamic font size: Otherwise the fixed size of the `QFont` that is passed to `setAxisTitleFont()` will be used, see the Fonts chapter for more information on this.

2. In addition (or alternatively) to using these methods you can enhance the axes by one or several Custom Text Boxes anchored at the any of the nine anchor points (see Figure 6.12) of the respective axis, using custom X and Y distances to the anchor, rotating the box at will.

   Actually the convenience functions described above do nothing else than adding and configuring such a custom box for you: For further access and additional setting up this implicitly added box you can retrieve its ID by calling the `KDChartParams` method `findFirstAxisCustomBoxID()` giving you the number of the first `KDChartCustomBox` that has been defined for the respective axis.

   See the special chapter on Adding Custom Text Boxes and the Reference Documentation for more information on the various features provided by the `KDChartCustomBox` class, including its background and framing options.

Labeling

In our example application we will first use the method `setAxisLabelString-Params()`.

This function specifies a `QStringList` from which the axis label texts will be taken, as well as other parameters as described below:

```
void KDChartParams::setAxisLabelStringParams(
    uint           n,
    QStringList*   axisLabelStringList,
    QStringList*   axisShortLabelStringList,
    const QString&  valueStart = QString::null,
    const QString&  valueEnd   = QString::null );
```

Parameters:

`n`	the ID of the axis
`axisLabelString-List`	points to the list of labels to be displayed (e.g. "Monday", "Tuesday", "Wednesday", etc.)
`axisShortLa-belStringList`	points to an alternative list of short names to be displayed if the long labels take too much place (e.g. "Mon", "Tue", "Wed", etc.)
`valueStart`	the label to begin with
`valueEnd`	the label to end with

Normally axis labeling starts with the list's first string and proceeds until its last string is reached: if there are more items than string iteration is restarted at the beginning accordingly. By specifying a start and an end value you make KD Chart repeat the strings between these limits only.

Let us add a piece of code to the sample program we wrote in the Area Frames chapter.

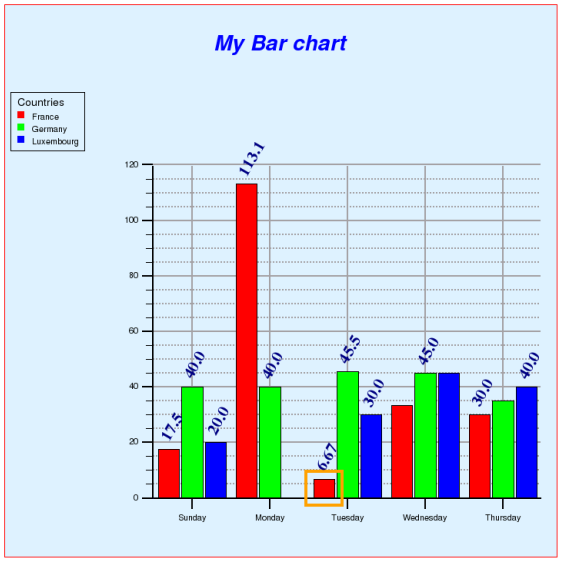
```
static QStringList abscissaNames;
if ( abscissaNames.empty() )
  abscissaNames << "Sunday" <<  "Monday"  << "Tuesday"
                << "Wednesday" << "Thursday"  << "Friday"
                << "Saturday"
static QStringList abscissaShortNames;
if ( abscissaShortNames.empty() )
  abscissaShortNames << "Sun"  << "Mon"  << "Tue"
                     << "Wed" << "Thu"  << "Fri"
                     << "Sat";
p.setAxisLabelStringParams( KDChartAxisParams::AxisPosBottom,
                            &abscissaNames,
                            &abscissaShortNames,
                            "Monday",
```

```
                                "Friday" );
```

Note that we are passing a valueStart and a valueEnd parameter.

Compile and run the tutorial file `kdchart_step04e.cpp` to see a our sample program at this stage, your chart should look like in Figure 6.1.

Figure 6.1. Changing The Axis Labels



Note

Axis labels can be formatted in various ways, see the reference documentation of the following `KDChartAxisParams` functions:

`setAxisLabelsCalc()`	Specifies the calculations to be applied to the axis labels: initial division by a power of ten and number of digits behind the comma. Will be ignored for non-numerical labels.
`setAxisLabelsRadix()`	specifies the 'radix character' (AKA decimal point) and the 'thousands point'. Will be ignored for non-numerical labels.
`setAxisLabelsFormat()`	specifies the way how the axis label strings will be formatted: prefix, postfix, padding. Will be ignored for non-numerical labels.

Fonts and Colors

Let us have a look at the fonts and change their configuration. To do that we can use `setAxisLabelsFont()`, which specifies the axis label font for one axis. Let us look at this method closer:

```
void KDChartParams::setAxisLabelsFont(
    uint   n,
    QFont  axisLabelsFont,
    int    axisLabelsFontSize = 0,
    QColor axisLabelsColor    = Qt::black );
```

Parameters:

`n`	the ID of the axis
`axisLabelsFont`	the font to be used for the labels
`axisLabelsFontSize`	the (fixed or relative) axis font size. If this value is less than zero the absolute value is per thousand of the actual printable area size. This will make the axis labels look the same even if scaled to very different size.
`axisLabelsColor`	the color of the labels

Note

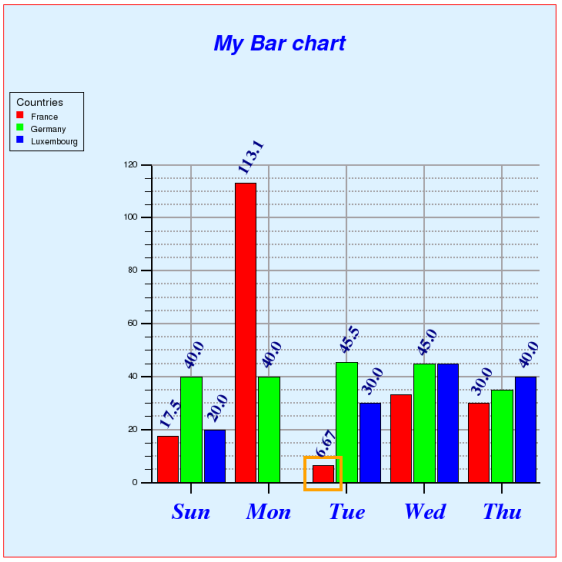
If `axisLabelsFontSize` is not zero the fixed size of `axisLabelsFont` is ignored but the font size is then calculated using the `axisLabelsFontSize`: either taking the value as fixed font size (if greater zero) or calculating the size dynamically based on the printable area's width or height—which ever is smaller.

To test this we add the following code to our source file:

```
p.setAxisLabelsFont( KDChartAxisParams::AxisPosBottom,
                     QFont( "times", 1, QFont::Bold, true ),
                     -44,
                     Qt::blue );
```

Compile and run the tutorial file `kdchart_step04f.cpp` to see a our sample program at this stage, your chart should look like in Figure 6.2.

Figure 6.2. Changing The Font Of The Axis Labels



By changing the widget's width we can make KD Chart use either the long or the short strings (see Figure 6.2).

Rotation/Shrinking

For optimal use of the space available below the data area KD Chart tries to rotate the abscissa axis labels if they do not fit into their fields. If rotating does not help the labels are *shrinked*: their font size is reduced to make them smaller.

This feature can be adjusted to your needs by calling one or both of the `KDChartAxisParams` functions `setAxisLabelsDontAutoRotate()` and `setAxisLabelsDontShrinkFont()`. Note that these functions require applying the axis parameters setting technique shown in the little sample code at the beginning of this chapter (see Axis Manipulation [69]).

Besides from automatic rotation, you can also specify a fixed rotation angle using the `KDChartAxisParams` function `setAxisLabelsRotation()`: in case auto-rotation *is* enabled the degrees value specified by you will be used as minimal rotation angle (so the labels actually might be rotated even more), if it is deactivated your value will be used as fixed rotation angle.

Note

It might happen that KD Chart is not successful in trying to find a good font size or rotation angle for the abscissa axis labels: in this case you

might consider specifying shorter labels or using two separate `QStringLists` to provide a set of long labels and one of short labels, as shown in the chapter at Beginning of this Chapter.

Limiting And Reverting the Axes

Sometimes you want to deactivate KD Chart's automatic axis range detection and determine the upper and/or lower limits of an axis yourself. Doing that is as easy as calling one or both of the following `KDChartAxisParams` functions:

- `setAxisValueEnd()` sets the axis range's upper limit
- `setAxisValueStart()` sets its lower limit.

  The value given by `setAxisValueStart()` will be used as the exact start value, unless you call `KDChartAxisParams::setAxisValueStartIsExact( false )` to tell KD Chart that you gave only an approximate value and it should look for a better value to start with.

  Setting this value to `false` might be a good idea in case you allow your users to specify random start values but still want to make sure that the start value makes some sense: e.g. it would not look good to start with a number like `75003.5` when the delta value (described in the following paragraph) is something like `50.0`.

In addition (or alternatively) to setting the limits you might want to specify the step width to be used by this axis which can be done by calling `setAxisValueDelta()`. Note that these functions require applying the axis parameters setting technique shown in the little sample code at the beginning of this chapter (see Axis Manipulation [69]).

For Line charts, you can also specify that your axis labels are printed in reverted order by calling `setAxisValuesDecreasing()` to have the lowest value printed at the top end of the ordinate axis (or at the right end of the abscissa axis, resp.).

Note that all of these functions require applying the axis parameters setting technique shown in the sample code at the beginning of this chapter (see Axis Manipulation [69]).

Note

While these functions are especially useful to configure the ordinate axis you might find the `KDChartParams` function `setAxisLabelStringParams()` interesting for specifying your abscissa axis texts and/or its limits, see the chapter on Labeling for details.

The chapter on Isometric Coordinate Systems shows how to use `KDChartAxisParams::setAxisValues( true )` to get your abscissa axis calculated the same way as an ordinate.

Grid Lines

All grid lines displayed by KD Chart are calculated based upon their respective axis' delimiter positions: vertical grid lines are positioned according to the horizontal axis (normally the bottom abscissa) while horizontal grid lines are specified by the respective vertical axis which by default is the left ordinate.

Thus all grid manipulation is done by either using the respective `KDChartAxisParams` functions listed below or just calling the simple `KDChartParams` convenience method `setAxisShowGrid()` allowing for quickly (de)activating a set of grid lines.

The `KDChartAxisParams` provides you with the following functions for specifying the grid lines controlled by the respective axis:

`setAxisShowGrid()`	enables/disables the grid lines.
`setAxisGridStyle()`	specifies the `PenStyle` to be used, its default is `Qt::SolidLine`.
`setAxisGridColor()`	sets the color of the grid lines. After previous calls of this function you can use `setAxisGridColor( KDCHART_DEFAULT_AXIS_GRID_COLOR )` to reset the color to its default value.
`setAxisGridLineWidth()`	modifies the grid line width. Normally you would *not* call this function since grid lines in most cases look best in their default width: the same as width of the axis line they belong to. However when combining multiple datasets using differently scaled ordinates (as shown in the chapter on Several Ordinates) you might want to reduce the line width of the respective grid lines and/or use different grid colors to show two grids in the same area. Doing this it might also be a good idea to call `setAxisShowSubDelimiters( false )` to avoid the dotted sub lines or to set their style to `Qt::NoPen` to get sub-delimiters on the axis but no grid sub lines. Using `setAxisGridLineWidth( KDCHART_AXIS_GRID_AUTO_LINEWIDTH )`, you can reset the value to its default and get the grid line width adjusted to the width of the axis line automatically.

In addition to these methods, you can call the following functions to set up the grid's so

called "sub lines": thin dotted lines shown between the normal grid lines if there is enough space.

`setAxisShowSubDelimiters()`	enables/disables both the small sub delimiter lines shown at the axis and their respective grid sub lines.
`setAxisGridSubStyle()`	specifies the `PenStyle` to be used, default is `Qt::DotLine`. Set the style to `Qt::NoPen` in case you want to see the axis' small sub delimiter lines but no grid sub lines.
`setAxisGridSubColor()`	sets the color of the grid sub lines, can be reset by `KDCHART_DEFAULT_AXIS_GRID_COLOR`.
`setAxisGridSubLineWidth()`	specifies the width of the grid sub lines, default is `KDCHART_AXIS_GRID_AUTO_LINEWIDTH`.

Note

Regardless of the previously described functions *no grid* is drawn for the as of yet experimental Three-dimensional Bar Charts in Multiple Rows.

Zero Lines

scope: A zero line is drawn between the Grid Lines if the zero value is not at the lower end of the axis but somewhere inside the visible range of the axis values.

The zero lines of your chart (just like the grid lines) are controlled by the respective axis: the horizontal zero line is depending on a vertical axis (normally the left ordinate) and the vertical zero line (if any) is based upon a horizontal axis (by default the bottom abscissa).

That's why zero line settings are specified by calling the appropriate function of their respective `KDChartAxisParams`. However as of yet KD Chart offers only one zero line method:

`setAxisZeroLineColor()`	specifies the color of the zero line, the default color is a dark blue, exactly: `QColor( 0x00, 0x00, 0x80 )`.

This method is called with either a specific color or with the respective grid's line color function, which would make the zero line look like a normal grid line, at least unless you have changed the grid line width: like the default line width of the grid lines (`KDCHART_AXIS_GRID_AUTO_LINEWIDTH`) the zero line is drawn using the line width of the respective axis.

So under normal circumstances the following code will make the zero line of your default ordinate axis look like a normal grid line:

```
KDChartAxisParams pa( p.axisParams( KDChartAxisParams::AxisPosLeft ) );
pa.setAxisZeroLineColor( pa.axisGridColor() );
p.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );
```

Touching the Edges

KD Chart uses reasonable default settings making sure that the axis labels and the contents of the data area are properly aligned. Thus—while you *may* call the `KDChartParams::setAxisLabelsTouchEdges()` function—you should be careful when doing this: overriding these build-in settings might result in bad looking charts, e.g. calling `setAxisLabelsTouchEdges( KDChartAxisParams::AxisPosBottom, true )` for a normal bar chart surely is *not* a good idea since the different bar groups will not be centered above their respective item labels anymore.

The sole purpose of this function is to provide you with a convenient option for setting up the axis layout in case you want to extend KD Chart by sub-classing the `KDChartAxisPainter` class for implementing your own chart types: this allows you to use the full range of the `KDChartAxisParams` capabilities without the need of writing your own axis painter methods.

Note

If you *still* wish to override the default settings of an axis by calling `setAxisLabelsTouchEdges()` make sure to call it again each time you have used `setChartType()` since your settings will be overwritten when changing the chart type.

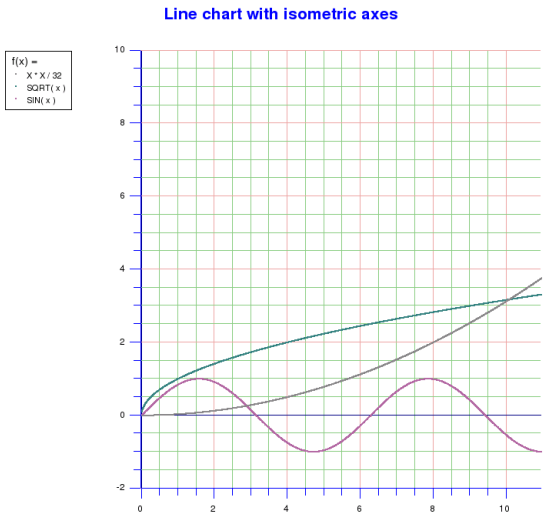
Isometric Coordinate Systems

Making your chart's abscissa axis and its ordinate axis isometric to each other is necessary for several kinds of *line or point charts*, e.g. you normally want this when displaying arithmetic function plots or when drawing real world maps like a plan showing subway routes or something like that.

To make our ordinate axis and abscissa axis use the same scaling width let us modify the abscissa's `KDChartAxisParams` settings, this `p` is our `KDChartParams` object:

```
KDChartAxisParams pa( p.axisParams( KDChartAxisParams::AxisPosBottom ) );

// specify that bottom axis labels must be CALCULATED
// (this initializes all calculation parameters)
pa.setAxisValues( true );

// make left and bottom axis use the same scaling width
pa.setIsometricReferenceAxis( KDChartAxisParams::AxisPosLeft );

p.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );
```

Figure 6.3. An Isometric Chart



Compile and run the tutorial file `kdchart_step04i.cpp` to see a simple line plot with isometric axes (see Figure 6.3): when increasing the widget's height you will see the Y axis adding more steps instead of changing its scale as it would do normally.

Multiple Axes

Offering up to eight different axes KD Chart provides you with a wide range of display options which let you replace the default axes by other ones (see Hiding/Showing Axes), have two ordinate axes facing each other or make them use different parts of your chart's vertical extent (covered by chapter Several Ordinates). You can even display two different chart types together—sharing the same abscissa axis as shown in the Combining Two Charts chapter below.

The following names address your chart's different axes all of which could be shown simultaneously, of course this would not make your chart look clearer:

`AxisPosBottom`	the default abscissa axis
`AxisPosLeft`	the default ordinate axis
`AxisPosTop`	an alternative abscissa axis
`AxisPosRight`	an alternative ordinate axis
`AxisPosBottom2`	an additional bottom abscissa axis
`AxisPosLeft2`	an additional left ordinate axis
`AxisPosTop2`	an additional top abscissa axis
`AxisPosRight2`	an additional right ordinate axis

While the above listed axis names can be used for all Cartesian axes you should use the following ones to address your Polar Chart's axes:

`AxisPosSagittal`	used to set up the degree texts which can be printed around the outer circle of a polar chart. By calling the `KDChartParams::polarZeroDegreePos()` function you can specify where the zero degree label is shown.
`AxisPosCircular`	used to specify the axis labels which can be printed at the circles of your polar chart. You can specify up to eight different directions to be followed when drawing these labels and print them horizontally or rotated, see the Reference Documentation on the `KDChartParams` functions `setPolarDelimsAndLabelsAtPos()` and `setPolarRotateCircularLabels()`.

The following chapters will concentrate on the Cartesian names to address the axes.

Hiding/Showing Axes

In case you just want to make an axis invisible you should bear in mind that most of the axis-based chart types must have an ordinate axis to be able to display their data: the reason for this is the ordinate not only showing the axis labels but also controlling the horizontal grid lines as well as the vertical position of the data points (or the heights of the bars, of the areas, etc.) So while hiding the bottom (abscissa) axis by calling the `KDChartParams` function `setAxisVisible( AxisPosBottom, false )` normally would be no problem you should *not* try to do make your *ordinate* invisible but consider taking less dramatic measures instead: e.g. you could suppress the labels of the ordinate by calling the `KDChartAxisParams::setAxisLabelsVisible( false )` or you might decide to use the right axis instead of the left one which we are going to do now.

Showing another ordinate or abscissa axis instead of the default left or bottom one is no problem and can be done in two simple steps shown here for replacing the default left axis by its right side counterpart (if `p` is your `KDChartParams` instance):

```
p.setAxisVisible(  KDChartAxisParams::AxisPosLeft,  false );
p.setAxisVisible(  KDChartAxisParams::AxisPosRight, true );
p.setAxisDatasets( KDChartAxisParams::AxisPosLeft,
                   KDCHART_NO_DATASET );
p.setAxisDatasets( KDChartAxisParams::AxisPosRight,
                   KDCHART_ALL_DATASETS );
```

After making the left axis invisible and enabling the right axis we tell KD Chart that the left axis is not responsible for any value calculations anymore but the right axis shall control all datasets now.

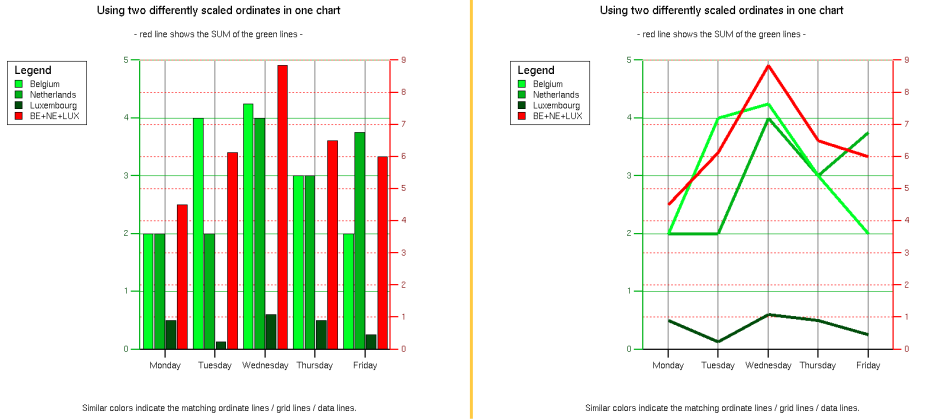
Note

Take care to perform *both* steps! It is *not* enough to deactivate one axis and activate another one: KD Chart would not know which datasets are to be calculated based on which axes since there can be complex situations where several different axes control a subset of the data series each. Thus changing the default axes requires specification of their dataset responsibilities too.

Several Ordinates

By using more than one ordinate axis in your chart, you can display datasets with different data ranges together. Of course, this *must be done with great care* to avoid confusion, a good idea might be to color your datasets according to the color of the ordinate axis they belong to and you might add some explanation to inform the viewer about the dataseries' different scaling, as shown in the following sample charts (see Figure 6.4).

Figure 6.4. Two Ordinates In One Chart



The multi-ordinate charts in Figure 6.4 basically show the same information as the normal Stacked Line Charts or Stacked Bar Charts: they visualize both the values of the single data and the total values of the datasets. However adding that extra summary dataset instead of stacking the lines (or bars, resp.) makes it much more easy to compare the values of the single data. While in theory this is possible with the stacked chart types too it is not very easy then since only the bottom lines/bars would start at the same height, the stacked ones usually start at different levels making it hard to compare their absolute values.

Furthermore we gain an additional benefit from displaying a differently scaled second ordinate for the summary dataset: now being able to use the full height of the left axis for the single data we are able to distinguish their values way better, this is especially true if one of the datasets has its values close together.

Setting up such a chart with two ordinates is very easy, see source file `kdchart_step05.cpp`. Besides from calling some simple color setting functions the most interesting methods used are the ones specifying which datasets belong to which ordinate axis:

```
p.setAxisDatasets( KDChartAxisParams::AxisPosLeft,  0, 2,   0 );
p.setAxisDatasets( KDChartAxisParams::AxisPosRight, 3, 3,   0 );
```

Let us have a short look at this important `KDChartParams` function:

```
void KDChartParams::setAxisDatasets( uint axis,
                                     uint dataset,
                                     uint dataset2,
                                     uint chart = 0 )
```

Parameters:

`axis` the ID of the axis.

`dataset` the number of the first dataset controlled by this axis.

Set this to `KDCHART_ALL_DATASETS` if *all* datasets shall be controlled by this axis: this is set by default for the left ordinate and for the bottom abscissa.

Set the parameter to `KDCHART_NO_DATASET` if no datasets are to be controlled by this axis, this is necessary in case you want to change the axis to be used, e.g. if for showing the right axis *instead of* the left one, as shown in the chapter on Hiding/Showing Axes.

`dataset2` the number of the last dataset controlled by this axis. This is ignored if `dataset` is either `KDCHART_ALL_DATASETS` or `KDCHART_NO_DATASET`.

`chart` the number of the chart this dataset—or this series of datasets, if the parameter `dataset2` is used too—belong(s) to.

We ignore this parameter for now and use its default zero value since we are using only one chart in this `KDChartWidget`, see the chapter on Combining Two Charts for more on this.

Note

As indicated by the parameters `dataset` and `dataset2` the datasets belonging together have to be in a contiguous series: there must be no gap between the datasets to be displayed by one axis.

Compile and run the tutorial file `kdchart_step05.cpp` to see a line chart with two differently scaled ordinate axes (see Figure 6.4): the series #0..#2 are controlled by the left ordinate and their lines are shown it green colors while series #3 is using red to match the right ordinate's color.

Using two differently scaled ordinates in the same chart is possible for Line Charts and for Bar Charts only. The right chart of the two samples shown above (see Figure 6.4) was produced using exactly the same program as the left one—we only changed the chart type, as you can see by looking at the source of the respective implementation file: `kdchart_step05.cpp`.

For testing the bar chart sample just change the source file to be used in the project file `step05.pro` by removing the # from the `#SOURCES = kdchart_step05_bar.cpp` and adding a # before the `SOURCES = kdchart_step05.cpp`.

Combining Two Charts

As shown in the chapter on Several Ordinates it is easy to display more than one ordinate axis at the same time. You can use this technique to either assign one or several of your chart's datasets to the other ordinate axis, or to display a second chart inside your chart. The primary chart and this additional chart will be sharing the same abscissa axis then.

Note

In case your charts shall *not* share the same abscissa axis but be completely independent from each other (using two separate `KDChartWidget` objects) you still can make them look belonging together by specifying some of KD Chart's layout parameters, an example showing the details is provided in Chapter 8, *Multiple Charts*.

Combining two charts sharing the same abscissa can be easily done like this:

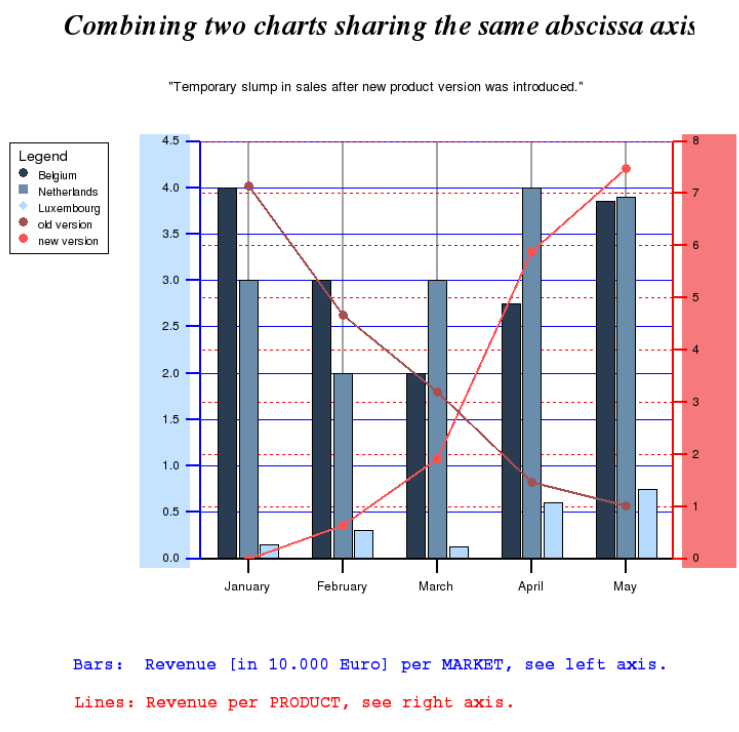
1. Specify an additional chart to be displayed together with the primary chart.
2. Declare which datasets belong to which chart.
3. Enable the second ordinate and assign the datasets to their respective axes.

Assuming `p` is your `KDChartParams` instance the following code would display a line chart and a bar chart, datasets #0..#2 belonging to the bar chart (controlled by the left axis) and datasets #3 and #4 displayed as lines and controlled by the right axis:

```
p.setChartType(           KDChartParams::Bar );
p.setAdditionalChartType( KDChartParams::Line );
p.setChartSourceMode( KDChartParams::DataEntry,     0, 2,   0 );
p.setChartSourceMode( KDChartParams::DataEntry,     3, 4,   1 );
p.setAxisDatasets( KDChartAxisParams::AxisPosLeft,  0, 2,   0 );
p.setAxisDatasets( KDChartAxisParams::AxisPosRight, 3, 4,   1 );
```

The charts resulting from this code are drawn onto the same part of the data area, using chart types like bar and line this might look very well as you can see in Figure 6.5.

Figure 6.5. Combining Two Charts: Lines in-line



Compile and run the tutorial file `kdchart_step05a.cpp` to learn how to combine two charts and have them drawn in-line.

Now let us see how this sample could be enhanced even further: e.g. you might want use *two* ordinates for the *primary* chart, so we modify our little sample code like this:

```
p.setChartType(           KDChartParams::Bar );
p.setAdditionalChartType( KDChartParams::Line );
p.setChartSourceMode( KDChartParams::DataEntry,     0, 3,   0 );
p.setChartSourceMode( KDChartParams::DataEntry,     4, 5,   1 );
p.setAxisDatasets( KDChartAxisParams::AxisPosLeft,  0, 2,   0 );
p.setAxisDatasets( KDChartAxisParams::AxisPosRight, 3, 3,   0 );
p.setAxisDatasets( KDChartAxisParams::AxisPosLeft2, 4, 5,   1 );
```

This would enable a *third* ordinate axis: the `AxisPosLeft2` axis would control our additional chart while `AxisPosLeft` and `AxisPosLeft2` would control the datasets displayed by the first chart.

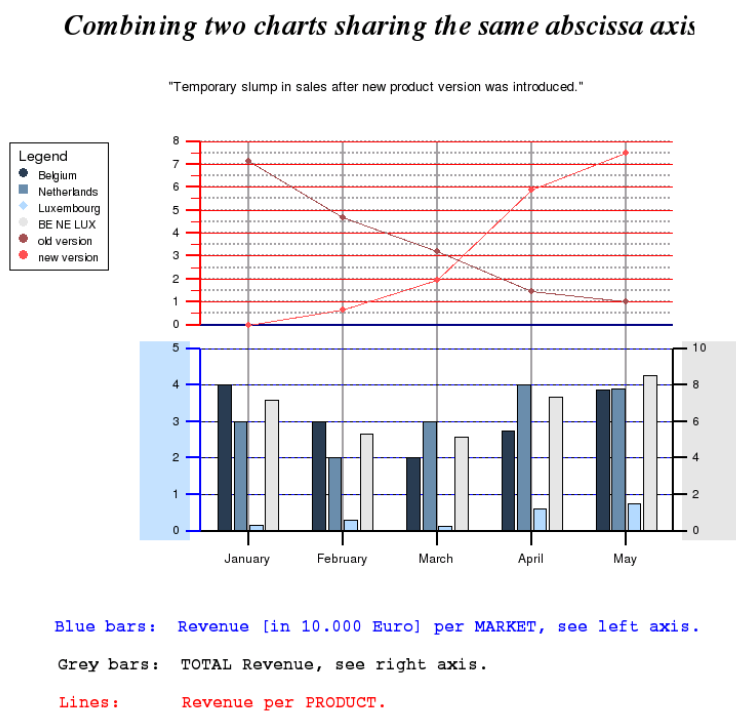
However using these three axes at the same level would it make quite difficult to understand our combined charts therefor we shift now the additional chart vertically by spe-

cifying which amount of the available space in the data area will be used by the respective axes:

```
// first left axis:
KDChartAxisParams pa(
    p.axisParams( KDChartAxisParams::AxisPosLeft ) );
// use the lower 47 percent of the available space for this axis
pa.setAxisUseAvailableSpace( 0, -469 );
p.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );

// right axis:
pa = p.axisParams( KDChartAxisParams::AxisPosRight );
pa.setAxisUseAvailableSpace( 0, -469 );
p.setAxisParams( KDChartAxisParams::AxisPosRight, pa );

// second left axis:
pa = p.axisParams( KDChartAxisParams::AxisPosLeft2 );
pa.setAxisUseAvailableSpace( -530, -1000 );  // the upper 47 percent
p.setAxisParams( KDChartAxisParams::AxisPosLeft2, pa );
```

Now our charts are separated nicely from each other since their left ordinates are not contiguous, see Figure 6.6.

Figure 6.6. Combining Two Charts: Lines on Top



Compile and run the tutorial file `kdchart_step05b.cpp` to see how easy it is to create such combined charts using several ordinates but one single abscissa.

Combining two charts with *different abscissa values* is also possible, as shown in the following sample displaying an area chart and a line chart in the same section of the screen: both the area and the lines are generated from data cells holding different abscissa axis values, so the points cannot be directly compared as in a simple chart using "Item 1", "Item 2", "Item 3" labels.

When an additional chart is specified KD Chart by default uses the last dataset for this one and the first datasets for the main chart. We don't want this here so we assign the first dataset to the main chart and all the remaining datasets to the additional (Line) chart:

```
// Area chart
p.setChartSourceMode( KDChartParams::DataEntry,   0, 0,   0 );
 // Line chart
p.setChartSourceMode( KDChartParams::DataEntry,   1, 2,   1 );
```

Another problem might occur due to KD Chart automatically calculating the ordinate axis limits based upon the datasets that are represented by the respective axis. To prevent the right and the left axis from being differently scaled we set their limits to equal values:

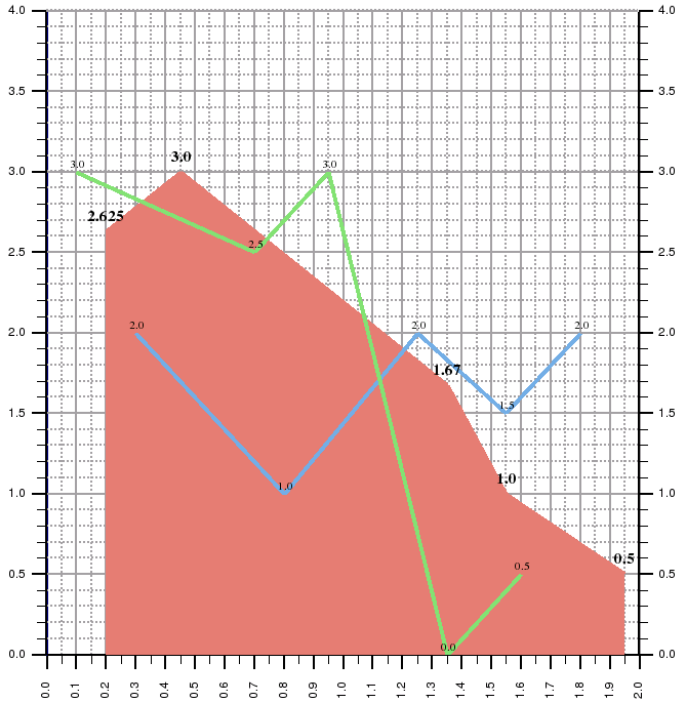
```
const double yMin = 0.0;
const double yMax = 4.0;

// Area chart
KDChartAxisParams pa(
    p.axisParams( KDChartAxisParams::AxisPosLeft ) );
pa.setAxisValueStart( yMin );
pa.setAxisValueEnd(   yMax );
p.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );

// Line chart
pa = p.axisParams( KDChartAxisParams::AxisPosRight );
pa.setAxisValueStart( yMin );
pa.setAxisValueEnd(   yMax );
p.setAxisParams( KDChartAxisParams::AxisPosRight, pa );
```

To improve the layout we now declare a fixed rotation angle for the abscissa axis labels:

```
pa = p.axisParams( KDChartAxisParams::AxisPosBottom );

// deactivate automatic rotation adjustment
pa.setAxisLabelsDontAutoRotate( true );

// specify fixed rotation angle
pa.setAxisLabelsRotation( 270 );
p.setAxisParams( KDChartAxisParams::AxisPosBottom, pa );
```

The resulting image shows the area and the lines together, see Figure 6.7. Note how the abscissa axis has numbers, not items, since it is calculated to match the different X coordinates of the data cells.

Figure 6.7. Combining Area and Lines: numerical abscissa axis



Compile and run the tutorial file `kdchart_step05c.cpp` to see how these combined charts were specified.

Data Value Texts

KD Chart by default prints the data cells' values next to their representations (bars, lines, etc) for all chart types except of the High/Low Charts and the Box & Whisker Charts which provide their own methods for enabling printing of all or several of the texts.

To suppress printing of data value texts you would just call the following `KDChart-Params` function: `setPrintDataValues( false )`.

This function can take a large number of parameters only the first of which is mandatory, it can be used in case you want to specify nearly all of the data value text settings in one go. Normally you might prefer calling the separate functions expecting only one or a few parameters each:

`setDataValuesCalc()`	takes `divPow10` (power of ten divisor, default `zero`) to by applied to each data value before printing it and `digitsBehindComma` specifying the number of digits that are to be shown behind the comma, default is `KDChartParams::DATA_VALUE_AUTO_DIGITS`.
`setDataValuesFont()`	takes `font*` pointing to the `QFont` to be used (no default, set to `zero` to leave the font unchanged) and `size` specifying the relative font size (if different from its default `UINT_MAX` value). Set the size to `zero` to use the fixed size of the `QFont` if you want to disable dynamic font size calculation, see chapter Customizing Your Chart / Fonts.
`setDataValuesPlacing()`	takes `position` which is one of the nine anchor points (see Figure 6.12) of the respective data area (the region covered by the bar, the pie slice, the ring segment, etc), `align` declaring how the text is to be aligned to this anchor point, `deltaX` and `deltaY` defining the distance from this anchor point, `rotation` specifying the number of degrees by which the text is to be rotated and a special flag `specifyingPositiveValues` to be set `false` in case you want to have all of these settings to be applied to *negative* values—by default they are used to specify the placing of data value texts that are greater or equal to zero. For charts with Cartesian axes, you can specify text rotation by a fixed degree value, however for *circular* charts (like Pie Charts or Ring Charts) you might consider using one of the special `KDChartParams` values `TANGENTIAL_ROTATION` or `SAGITTAL_ROTATION` to have the rotation angle of your texts adjusted to the rotation angle of the data point they are aligned to. The extra `specifyingPositiveValues` parameter is useful e.g. for Bar Charts where the data value texts are shown on different positions depending on the data value being negative or positive.
`setDataValuesColors()`	takes `color*` (its default being the special pointer value `KDCHART_DATA_VALUE_AUTO_COLOR`) pointing to the `QColor` to be used (if not zero) and `background` specifying the `QBrush` for the background of the texts if different from its default `Qt::NoBrush` setting.
`setAllowOverlappingDataValueTexts()`	can be used to deactivate KD Chart's layout behavior: data value texts that would overwrite existing texts are skipped and not printed.

In addition to the parameters mentioned above all of these function take a last `chart`

parameter which you would normally ignore since the default is set to KD-CHART_ALL_CHARTS resulting in the settings being applied to all charts that are controlled by this KDChartParams instance. In case you are displaying an Additional Chart you could set this parameter to either zero or one for specifying different data value text settings for the main chart and for the additional chart, e.g. you might want to display the additional chart's texts in another color or using a smaller font.

Note

All parameters specified for the data value texts can be reset to KD Chart's defaults by calling the following KDChartParams function:

```
setPrintDataValuesWithDefaultFontParams();
```

See the Reference Documentation for details on this method.

Cell specific Properties

While the KDChartParams class enables you to set a large number of chart specific or dataset specific parameters KD Chart offers an extra technique for design modifications on data cell level: with the most widely used chart types Bar Chart and Line Chart you can further enhance the layout by means of the KDChartPropertySet class providing a powerful set of options for either *changing* the look of one data bar (or line, resp.) or *adding* some cell specific layout elements: vertical and/or horizontal lines which optionally can show their own markers on one or both ends.

The Property Set Concept

KD Chart's concept of cell specific properties is both simple and flexible: in addition to containing the data value(s) each data cell stores an extra *property set ID*. When the chart is drawn these ID values are used to find the property sets assigned to the cells and the respective bar's or line's look is modified accordingly.

Each new KDChartData object is initialized automatically with the default value KD-CHART_PROPSET_NORMAL_DATA used for cells without any special properties. While (in theory) you *could* assign special property values to this build-in property set this normally might not a very good idea since these values would apply to every(!) data cell that has no other property set assigned. Such general changes would rather be done by calling chart specific KDChartParam functions, e.g. you would invoke KDChart-Params::setLineMarker(true) instead of changing the normal-data property set.

A KDChartPropertySet holds a set of property specifications each of which is stored in a "Property Set ID & Property Value" pair of parameters:

"Property Set ID" — the ID of the *property set* that is to be used for retrieving the value of this property, this can be:

- `KDChartPropertySet::OwnID` to be used if you want to specify the property value by *this* property set's respective "Property Value" parameter.
- An ID value specifying *another* property set would indicate that the respective property value shall be retrieved from that set. In this case the following "Property Value" parameter would be ignored.
- The default `KDChartPropertySet::UndefinedID` means that *neither* another property set's ID *nor* an own value is specified for this property.

"Property Value" — the value to be used for this property.

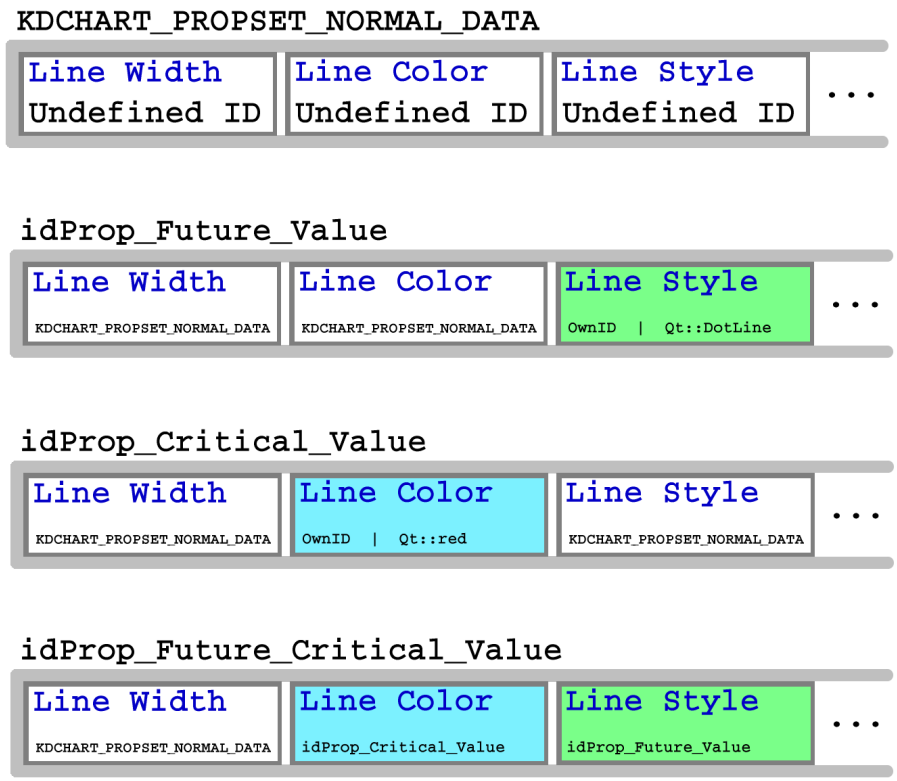
This is ignored if the respective "Property Set ID" is *not* `KDChartPropertySet::OwnID` as explained above.

See the next chapter for an sample program defining a simple hierarchical structure of property sets and assign them to the respective data cells.

Hierarchical Property Sets

Using the `KDChartPropertySet` class we can easily define our own property sets and even create hierarchical structures of such sets as illustrated in Figure 6.8.

Figure 6.8. A Simple Property Set Hierarchy



As shown in the illustration above (see Figure 6.8) we can inherit our own property sets from the default set `KDCHART_PROPSET_NORMAL_DATA`, but you are also free to do without and use the default constructor only.

Assuming that `p` is our `KDChartParams` instance such inheritance can be specified by using the third constructor of the `KDChartPropertySet` class. By passing to it both a descriptive name and a parent ID we would set all of the set's "Property Set ID" entries to that ID:

```
KDChartPropertySet propSetForFutureValue(
        "future value",
        KDCHART_PROPSET_NORMAL_DATA );
```

Now we can set the special line style that we want to use for cells having this property set assigned:

```
propSetForFutureValue.setLineStyle(
        KDChartPropertySet::OwnID, Qt::DotLine );
```

Our second property set (for critical values) is defined accordingly:

```
KDChartPropertySet propSetForCriticalValue(
        "critical value",
        KDCHART_PROPSET_NORMAL_DATA );
propSetForCriticalValue.setLineColor(
        KDChartPropertySet::OwnID, Qt::red );
```

OK, so we have specified these two property sets, but at the moment they are nothing more than two objects laying on our program's stack. To actually use them we need two IDs that we can store in our data cells: we obtain these IDs by *registering* our property sets with our `KDChartParams` object `p`:

```
const int idProp_Future_Value
          = p.registerProperties( propSetForFutureValue );
const int idProp_Critical_Value
           = p.registerProperties( propSetForCriticalValue );
```

Having registered these two property sets we now can use their IDs to specify our third set which we want to assign to cells containing data that are both future *and* critical, and we also register this property set after declaring it:

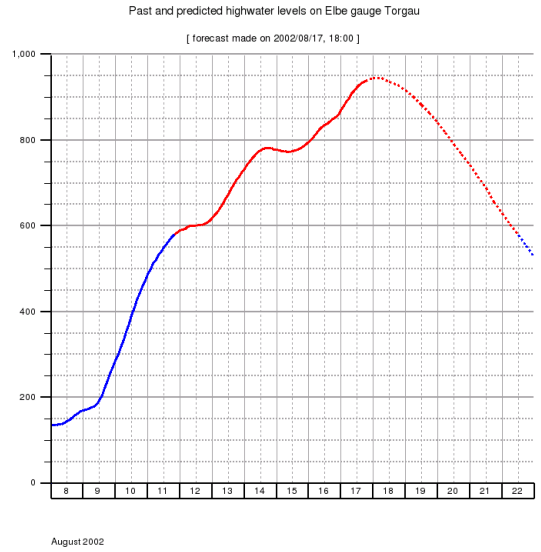
```
KDChartPropertySet propSetForFutureCriticalValues(
        "future + critical values",
        KDCHART_PROPSET_NORMAL_DATA );
propSetForFutureCriticalValues.setLineStyle( idProp_Future_Value,
        Qt::NoPen );  // this 2nd parameter will be ignored
propSetForFutureCriticalValues.setLineColor( idProp_Critical_Value,
        QQColor() );
const int idProp_FutureCritical_Value
        = p.registerProperties( propSetForFutureCriticalValues );
```

Now we have three new property sets which we can assign to our data cells by passing their ID numbers; e.g. if our data are stored in a `KDChartTableData` object called `d` and its cells hold both a double and a date value we could do something like this:

```
const double alarmLevel = 580.0;
const QDate today  = QDate::currentDate();
for( int iCell = 0;  iCell < numCells;  ++iCell ){
    KDChartData& cell = d.cell( 0, iCell );
    bool bIsFuture   = cell.dateTimeValue( 2 ).date() > today;
    bool bIsCritical = cell.doubleValue(   1 )        > alarmLevel;
    if( bIsFuture || bIsCritical )
        cell.setPropertySet(  bIsFuture
                            ? (   bIsCritical
                                ? idProp_FutureCritical_Value
                                : idProp_Future_Value )
                            : idProp_Critical_Value );
}
```

The following screen shot (see Figure 6.9) shows how a line chart using these little code pieces might look like:

Figure 6.9. Line Chart Using Hierarchical Property Sets



Compile and run the tutorial file `kdchart_step06.cpp` to see how easy it is to create such hierarchical property sets and use them to enhance your charts.

Drawing Extra Lines/Markers To A Value

In the previous chapter we used the `KDChartPropertySet` class to modify the look of our line chart's data lines. Now we will add two *extra* lines marking the last true (not predicted) data value on both of the axes.

Doing this is very easy since the `KDChartPropertySet` class provides a wide range of configuration options for adding extra horizontal and/or vertical lines which can optionally show their own markers on one or both ends.

To make our program act like a real world application we declare two different property sets: one to be used in case the last true data cell contains a *normal* value and a second one that will be taken if it has a *critical* value.

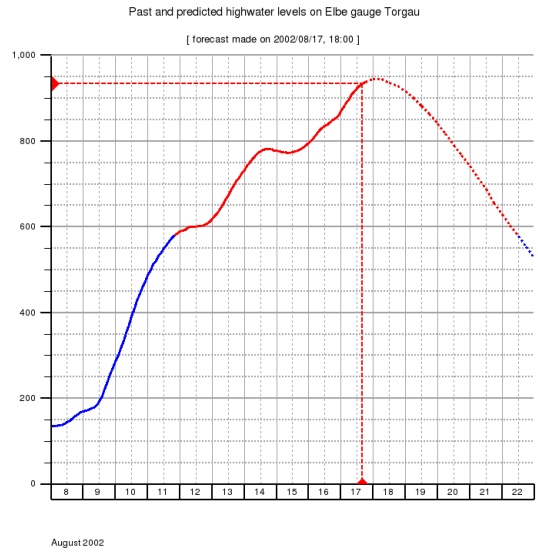
To avoid duplicate definitions we make the later set dependent both on the prior one *and* on the `idProp_Critical_Value` set which we declared in the previous chapter. So every changed settings on either the "normal last value" set or the `idProp_Critical_Value` set will result in the "critical last value" set being adjusted

automatically:

```
// The first set inherits KDCHART_PROPSET_NORMAL_DATA:
KDChartPropertySet lastTrueValuePropsNormal(
        "two lines marking the last true value (if normal)",
        KDCHART_PROPSET_NORMAL_DATA );
// specify the extra lines and markers:
lastTrueValuePropsNormal.setExtraLinesAlign(
        KDChartPropertySet::OwnID, Qt::AlignLeft | Qt::AlignBottom );
lastTrueValuePropsNormal.setExtraLinesWidth(
        KDChartPropertySet::OwnID, -4 );
// more parameter specifications.
// (see the Reference Documentation of the KDChartPropertySet class)
// ...
int idProp_LastTrueValueNormal
    = p.registerProperties( lastTrueValuePropsNormal );

// The second set inherits the first set
// plus the idProp_Critical_Value set (for the line/marker colors):
KDChartPropertySet lastTrueValuePropsCritical(
        "two lines marking the last true value (if critical)",
        idProp_LastTrueValueNormal );
// color settings are inherited from the idProp_Critical_Value set:
lastTrueValuePropsCritical.setLineColor(
        idProp_Critical_Value, QColor() );
lastTrueValuePropsCritical.setExtraLinesColor(
        idProp_Critical_Value, QColor() );
lastTrueValuePropsCritical.setExtraMarkersColor(
        idProp_Critical_Value, QColor() );
int idProp_LastTrueValueCritical
    = p.registerProperties( lastTrueValuePropsCritical );
```

The following screen shot (see Figure 6.10) shows our line chart having one of these two new property sets assigned to the last true data value:

Figure 6.10. Drawing Extra Lines To A Value



Compile and run the tutorial file `kdchart_step06a.cpp` to see how easy it how the chart shown in Figure 6.10 was created.

Adding Separate Lines/Markers

In the previous chapter we used the `KDChartPropertySet` class to draw some lines from both axes to the the last true data value. Now we will add some *separate* lines indicating the normal water level and the flood alarm level of the Torgau/Elbe gauge.

Displaying such separate lines requires special action: we *cannot* assign the respective property sets to our normal data cells, since it might be that none of the cells contain the values we want to indicate by our lines. Therefor our task must be done like this:

1. First add an extra dataset storing some anchor cells specifying the exact values for these two indicator lines.

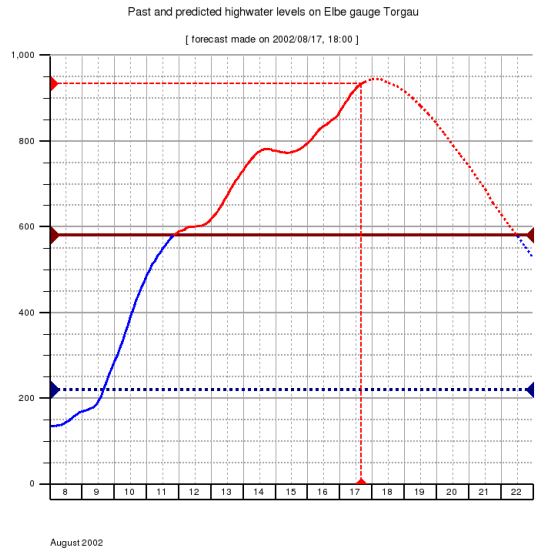
2. Now we can declare and register the new property sets for our lines and assign them to these anchor cells.

```
// use a two dataset wide table instead of one
KDChartTableData d( 2, maxValues );
// store the normal data in the dataset #0 cells as usual
// ...

// store the extra anchor cells in the dataset #1 cells
// using some date/time values that are in the range
// of the normal data cell's date/times
d.setCell( 1, 0,
        KDChartData(
            220.0,
            QDateTime(QDate(2002, 8, 17), QTime( 15, 0 )) ) );
d.setCell( 1, 1,
        KDChartData(
            580.0,
            QDateTime(QDate(2002, 8, 17), QTime( 16, 0 )) ) );

// declare and register the normal water level line
KDChartPropertySet horiLinePropsA(
        "horizontal line at 2.20m level" );
horiLinePropsA.setLineStyle(KDChartPropertySet::OwnID, Qt::NoPen);
horiLinePropsA.setShowMarker(
        KDChartPropertySet::OwnID,
        false );
horiLinePropsA.setExtraLinesAlign(
        KDChartPropertySet::OwnID,
        Qt::AlignLeft | Qt::AlignRight );
// more parameter specifications...
// (see the Reference Documentation of the KDChartPropertySet class)
// ...
int idHoriLinePropsA
    = p.registerProperties( horiLinePropsA );

// declare and register the alarm water level line
KDChartPropertySet horiLinePropsB(
        "horizontal line at 5.80m level",
        idHoriLinePropsA );
horiLinePropsB.setExtraLinesStyle(
        KDChartPropertySet::OwnID,
```

```
        Qt::SolidLine );
int idHoriLinePropsB
    = p.registerProperties( horiLinePropsB );

// assign these new property sets to our extra anchor cells
d.cell( 1, 0 ).setPropertySet( idHoriLinePropsA );
d.cell( 1, 1 ).setPropertySet( idHoriLinePropsB );
```

The following Figure 6.11 shows our line chart using these two new property sets to indicate the water levels by two horizontal lines:

Figure 6.11. Adding Separate Lines/Markers



Compile and run the tutorial file `kdchart_step06b.cpp` to see how the chart shown above (see Figure 6.11) was created.

Field of Application

KD Chart's property set concept offers more than we can show in this manual, just try yourself following the instructions given in the Reference Documentation, e.g. you might use the above shown `KDChartPropertySet::setExtraLinesAlign()` method to *center* your line(s) by passing `Qt::AlignVCenter` or `Qt::AlignHCenter` which would allow you to also specify a line *length* by calling `KDChartPropertySet::setExtraLinesLength()`, this special function sets the true or relative size of a centered line since such lines do not touch the axes.

Another idea might be to highlight one of your data points by adding a marker to it (in a

line chart not displaying markers normally) which could be easily achieved by using the function `KDChartPropertySet::setShowMarker()`. So you could enable or disable drawing of one point's marker ignoring the general marker flag's state.

For bar charts, the following method can be used to highlight one of the bars: `KDChart-PropertySet::setBarColor()`. If your bar chart is shown in three-dimensional look this function would set not only the base color of a bar but also calculate its side and top shade colors, see the chapter on Colors for details.

Furthers you might want to have these extra lines and/or extra markers drawn *in front of* the data bars (or lines, resp.) instead of behind them which can be specified for each individual property set by calling the `KDChartPropertySet` function `setExtraLines-InFront()`, otherwise they are drawn behind the data representations.

You could even add an extra line marker without showing its extra line by specifying the line but setting its style to `Qt::NoPen`. This technique might be used together with the special `setExtraLinesInFront()` function for highlighting one of the data points by drawing an extra ring marker around it, the following simple property set would do the job:

```
KDChartPropertySet extraMarkerWithoutExtraLine(
        "ring marker drawn in front of its data point",
        KDCHART_PROPSET_NORMAL_DATA );

// specify the extra line:
extraMarkerWithoutExtraLine.setExtraLinesAlign(
        KDChartPropertySet::OwnID, Qt::AlignLeft );
// draw the extra line and its markers in front of the data
extraMarkerWithoutExtraLine.setExtraLinesInFront(
        KDChartPropertySet::OwnID, true );
// make the extra line invisible
extraMarkerWithoutExtraLine.setExtraLinesStyle(
        KDChartPropertySet::OwnID, Qt::NoPen );

// show an extra marker at the right side of the extra line
// since this is the respective data point's position
extraMarkerWithoutExtraLine.setExtraMarkersAlign(
        KDChartPropertySet::OwnID, Qt::AlignRight );
extraMarkerWithoutExtraLine.setExtraMarkersSize(
        KDChartPropertySet::OwnID, QSize(-50, -50) );
extraMarkerWithoutExtraLine.setExtraMarkersColor(
        KDChartPropertySet::OwnID, Qt::blue );
extraMarkerWithoutExtraLine.setExtraMarkersStyle(
        KDChartPropertySet::OwnID, KDChartParams::LineMarkerRing );

int idProp_ExtraMarkerWithoutExtraLine
    = p.registerProperties( extraMarkerWithoutExtraLine );
```

To see a sample chart that was created by applying this technique please have a look at an even more advanced version of our chart (see Figure 6.14) featuring a little green ring marker to highlight the last maximal water level's data point. The other functions used in that chart are explained in detail in the following chapter on Custom Text Boxes.

Of course, such separate extra markers could also be drawn *behind* the normal markers, e.g. a yellow `LineMarkerCircle` marker might look nice in a Point Chart as the one shown in Figure 3.10.

Note

While the functions described above can be used to set a *single* one of the `KDChartPropertySet` parameters, you also could specify *all of them* in one go by using the forth constructor of this class:

```
KDChartPropertySet( const QString& name,

    // for LINE charts only:

    // parameters modifying normal line and marker properties
    int idLineWidth,  int                lineWidth,
    int idLineColor,  const QColor&      lineColor,
    int idLineStyle,  const Qt::PenStyle& lineStyle,
    int idShowMarker, bool               showMarker,

    // for LINE and BAR charts:

    // parameters adding extra line properties
    // (only drawn if their alignment is specified)
    int idExtraLinesAlign,   uint               extraLinesAlign,
    int idExtraLinesInFront, bool               extraLinesInFront,
    int idExtraLinesLength,  int                extraLinesLength,
    int idExtraLinesWidth,   int                extraLinesWidth,
    int idExtraLinesColor,   const QColor&      extraLinesColor,
    int idExtraLinesStyle,   const Qt::PenStyle& extraLinesStyle,

    // parameters adding extra marker properties
    // (only drawn if extra lines alignment is specified)
    int idExtraMarkersAlign,uint          extraMarkersAlign,
    int idExtraMarkersSize, const QSize&  extraMarkersSize,
    int idExtraMarkersColor,const QColor& extraMarkersColor,
    int idExtraMarkersStyle,int           extraMarkersStyle,

    // for BAR charts only:

    // parameters modifying normal bar properties
    int idShowBar,  bool          showBar,
    int idBarColor, const QColor& barColor );
```

The Reference Documentation explains each of these parameters in detail.

Adding Custom Text Boxes

KD Chart enables you to easily add custom text boxes to your chart. You can specify the Position of the boxes and adjust their Appearance in detail. This chapter will demonstrate the flexibility of the custom box concept, showing you what can be done and how to do it. Additional information as always is provided by the KD Chart Reference Documentation.

As indicated by their class name such `KDChartCustomBox` objects do not have to contain text but they can also be empty: you could use an empty box as a reference area to draw a Simple Frame at a special place of your chart, perhaps for showing a Picture or your company emblem.

Position of the Box

The `KDChartCustomBox` class enables you to position the box to any anchor point of any of the Areas of a Chart. This includes all anchor points of all custom boxes that were added before, so you can even attach a new custom box to an edge (or to a corner or to the center, resp.) of another custom box.

Reference Area

Any area of your chart can be used as reference area for positioning a custom box.

Note

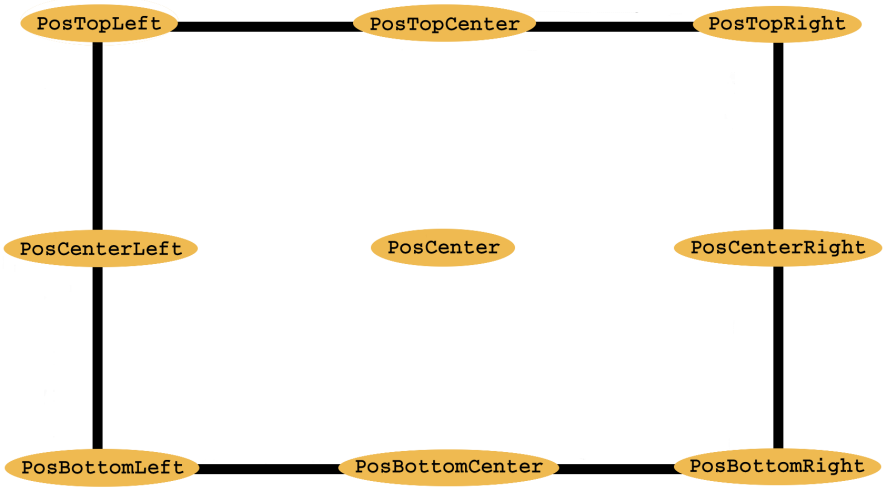
Chapter 5, *Areas of a Chart* explains the area concept by an illustration (see Figure 5.1) showing the positions and names of the most commonly used areas in a chart. This illustration was created by the tutorial file `kdchart_step03.cpp` which you will fully understand when you are through with our custom box chapter. For now you might want to look at this drawing and study the explanations given in the areas chapter since the rest of this chapter assumes that you know the area concept.

Let us now select a reference area for the custom box to enhance our water levels chart (see Figure 6.11) which we developed in the previous chapter on Cell specific Properties. We choose the *left axis* area since we will use this box to print "(cm)" on top of the ordinate axis, so the area ID is `KDChartEnums::AreaAxisBASE + KDChartAxisParams::AxisPosLeft`.

Anchor Point

Areas have nine anchor points as specified in the `KDChartEnums::PositionFlag` enum (see Figure 6.12).

Figure 6.12. The Anchor Points And Position Flags



For positioning our text box on top of the left axis let us use the upper right corner of that area as reference point: `KDChartEnums::PosTopRight`.

Anchor Alignment

Now we define how the box will be aligned to its anchor using a subset of the flags defined by the `Qt::AlignmentFlags` enum:

`Qt::AlignLeft`	box at the right side of the anchor point
`Qt::AlignRight`	box at the left side of the anchor point
`Qt::AlignHCenter`	box horizontally centered to the anchor point
`Qt::AlignBottom`	box above the anchor point

`Qt::AlignTop`	box below the anchor point
`Qt::AlignVCenter`	box vertically centered to the anchor point

We can use any reasonable combination of these flags or just take the `Qt::AlignCenter` value to have our box centered in both directions. However using the special values `Qt::AlignAuto` and `Qt::AlignJustify` is *not* possible for the anchor alignment.

Our sample box shall be positioned above and at the left of its anchor point, so we set the alignment flag to `Qt::AlignBottom | Qt::AlignRight`.

Anchor Delta

The anchor delta settings are useful in case the box shall not touch its anchor point but leave a gap between.

Delta Width

The width of the horizontal gap between the anchor and the box (`deltaX`) and the width of the vertical gap (`deltaX`) can be specified in three different ways:

- Positive values are taken as exact offsets.
- Negative values *by default* are interpreted as per mil of the width of the drawing area for the horizontal gap, or per mil of its height for the vertical gap, respectively.
- By setting the special flag `deltaScaleGlobal` to *false* we declare that negative values are to be interpreted as percent of the actual *font size* used for this box.

Since we do not want our text to touch the anchor point we leave a little gap between the box and the point by specifying a horizontal anchor delta of `-75` and a vertical delta of `-100` while setting the `deltaScaleGlobal` flag to `false`.

Delta Alignment

KD Chart normally uses the anchor delta values to make a *gap* between the box and its anchor point. So a box that is aligned by `Qt::AlignRight` would be moved to the left if the `deltaX` value is not zero.

This default behavior can be changed by setting the extra `deltaAlign` flag to a value different from its default `Qt::AlignAuto` setting.

Note

While the anchor alignment might be seen as the answer to the question: "On which side of the *box* is the anchor point?" you might see the delta alignment as the answer to the question "On which side of the *gap* is the anchor point?". Here is an example:

A *delta alignment* of `Qt::AlignRight` means the box will be moved to the left by the amount calculated using the `deltaX` value so there will be a gap between the right side of the box and its anchor point if the main *anchor alignment* is set to `Qt::AlignRight` too—the anchor point will to be outside of the box then. However if the main *anchor alignment* is set to `Qt::AlignLeft` the box will be moved to the right and the anchor point will be inside the box. We learn the following general rule:

Tip

Equal anchor and delta alignments produce a *gap*.

Different alignments make the box *cover* its anchor point.

Possible values for the `deltaAlign` flag are:

- `KDChartCustomBox::AlignAuto`
- `Qt::AlignLeft | Qt::AlignTop`
- `Qt::AlignLeft | Qt::AlignBottom`
- `Qt::AlignRight | Qt::AlignTop`
- `Qt::AlignRight | Qt::AlignBottom`

Using `AlignVCenter` or `AlignHCenter` or `AlignCenter` does *not* make sense here: center delta alignment will cause KD Chart to ignore the respective delta settings, so `deltaX` (or `deltaY`, resp.) would become ineffective.

For our sample chart we leave the delta alignment flag on its default status `AlignAuto` since we want the box to be moved to the left and to the top, which is done by default if the main alignment is set to `Qt::AlignBottom | Qt::AlignRight`.

Appearance of the Box

In this chapter on specifying the appearance of our `KDChartCustomBox` object we learn about its Size and Rotation, how to format the Text to be shown, hot to set its Base Font and its Base Color and how to display a colored Background behind the box. Extra information on background images is given in the separate chapter on Pictures.

Size

Calculating the size of a custom box can be done it three different ways:

- by specifying a fixed size
- by declaring a relative size to be adjusted to the actual size of the drawing area
- by setting both the width and the height of the box to *zero*: if `fontScaleGlobal` is set to `true` (see below at Base Font) and both the box width and height are set to `zero` the size of the box will be calculated automatically to match the `usedWidth()` and the `height()` of the `KDChartCustomBox` object displayed.

  The content is interpreted as *rich text* then, even if not framed by `"<qt>"` and `"</qt>"`.

In our example we will add some text about the left ordinate's unit of measurement and we want this text to grow and shrink according to changing widget size because the axis label texts will do that as well. Thus we decide to use a dynamic size for the box and have the text size adjusted to the box height (details on the Base Font will be given below).

To make the box size 20 percent of drawing area's width and 3.5 percent of its height we take a `-200` for `width` and a `-35` for the `height` parameter.

Rotation

Since our little text shall be printed horizontally above the left ordinate axis no rotation is needed:

Our box object will be instantiated by the normal `KDChartCustomBox` constructor (instead of using the special one taking an extra `int rotation` as first parameter, see the Reference Documentation for details.)

Text

The content of a `KDChartCustomBox` is controlled by KD Chart's little auxiliary class `KDChartTextPiece` holding both a `QString` specifying the text to be displayed and a `QFont` to be used as Base Font for this text.

The text can be a normal string but you might prefer using a `QSimpleRichText` string instead, starting with `<qt><p>` and ending with `</p></qt>`.

KD Chart uses Qt's default `QStyleSheet` object for the `KDChartTextPiece` class, so please have a look at the detailed information provided by the Qt Reference Documenta-

tion on the QStyleSheet class to learn all about the "Structuring tags" that can be used to format your texts.

For our chart we use this simple string producing a right aligned slanted text in normal brackets:

```
<qt><p align=right>(<i>cm</i>)</p></qt> .
```

Base Color

Let us now specify a QColor to be used for all parts of the text that are *not* framed by an extra ... pair specifying a font color. For this we just pass a Qt::darkBlue to our box constructor's color parameter.

Base Font

While the font used inside the box can be changed by means of the tag we still specify a *base font* to be used for all text parts *not* framed by an extra ... pair.

In our sample chart we did not alter the font to be used for the left ordinate's label texts so the default axis label font will be used, we want to print the unit of measurements in the same font, so we pass a simple QFont("helvetica", 12) to the KDChart-TextPiece specifying the content of the box (see above).

Having set the base font we now declare its size, so the size specified with the QFont shown above will be ignored. We achieve this by setting the following two parameters:

fontSize	the general size of the font, this might be overwritten inside the content by using some extra ... or some <big> ... </big> or ... tags as described by the Qt Reference Documentation on the QStyleSheet class. Positive values will be taken as absolute font size, negative ones are interpreted as per mil value of the size of the drawing area (or of the height of the box in case fontScaleGlobal is set to false, resp.). *However* if fontSize is zero no dynamic calculating will be done but the size of the content parameter's QFont value will be used (see above).
fontScaleGlobal	(by default true) specifies whether dynamic font size calculation (see the fontSize parameter) will be done based upon the size of the drawing area or (if it is set to false) based upon the size of the box.

To make the text use the full height of our little box we set the `fontSize` parameter to `-1000` and the `fontScaleGlobal` flag to `false`.

Background

The background of our box can be set by passing an appropriate `QBrush` to the constructor's `paper` parameter, but we do not want an extra background here, so we let its default value: `Qt::NoBrush`.

Making it all Work

In the previous chapters we learned how to determine the right values for the parameters specifying the different settings of our `KDChartCustomBox` object. Now we will add a little box to our sample chart to inform the viewer about the measurement unit of the left ordinate axis, so let us pass the values found in the previous chapters to the constructor:

```
KDChartCustomBox( KDChartTextPiece(
                      "<qt><p align=right>(<i>cm</i>)</p></qt>",
                      QFont( "helvetica" , 1 ) ),    // content
                  -1000,                             // fontSize
                  false,                             // fontScaleGlobal
                  -75,                               // deltaX
                  -100,                              // deltaY
                  -200,                              // width
                  -35,                               // height
                  Qt::darkBlue,                      // color
                  Qt::NoBrush,                       // paper
                  KDChartEnums::AreaAxisBASE
                   + KDChartAxisParams::AxisPosLeft, // area
                  KDChartEnums::PosTopRight,         // position
                  Qt::AlignBottom | Qt::AlignRight,  // align
                  0, 0,                              // dataRow, dataCol
                  0,                                 // (data3rd)
                  KDChartCustomBox::AlignAuto,       // deltaAlign
                  false );                           // deltaScaleGlobal
```

Note

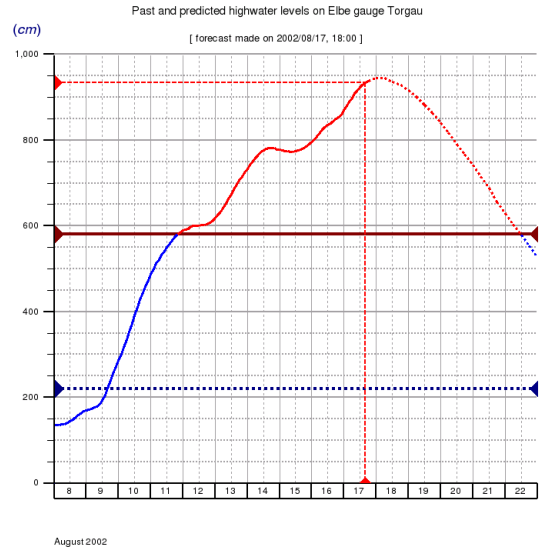
The `dataRow` and `dataCol` parameters are not used in this sample since our box is not aligned to a data point. If the `area` were set to `AreaChartDataRegion` these values would specify the *row* and the *col* of the cell in the `KDChartTableData` used.

To actually *insert* this box into our chart's layout we just pass this constructor call to the `KDChartParams::insertCustomBox()` function:

```
p.insertCustomBox( KDChartCustomBox( /*parameters as shown above*/ ) );
```

The following Figure 6.13 shows our line chart using this new custom box to indicate the unit of measurement:

Figure 6.13. Line chart showing a Custom Text Box



Compile and run the tutorial file `kdchart_step06c.cpp` to see how the chart shown above (see Figure 6.13) was created.

Tip

In addition to the rich set of options provided by the `KDChartCustomBox` class KD Chart offers some very simple convenience functions for adding *axis title texts* to your chart, please see the separate chapter on Axis Titles for details.

More Boxes

While in this manual we can *not* give a *complete* overview on all possible uses of the `KDChartCustomBox` class we still want to show two other sample boxes to get an impression of the wide range of things this concept enables you to do:

Let us start by adding a multi-colored box below the data area to inform about the meaning of the extra lines we added in the chapter on Cell specific Properties.

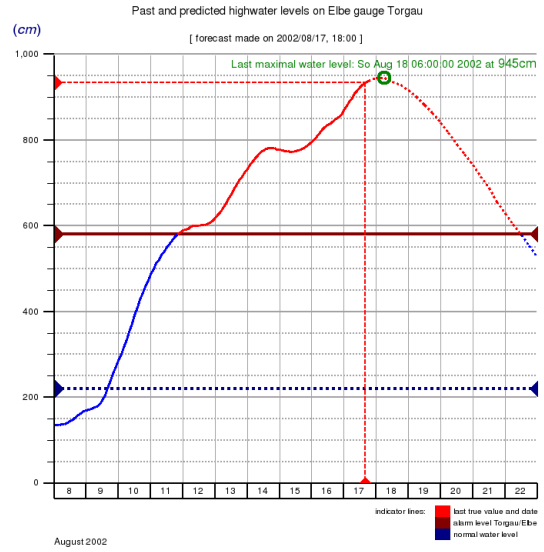
Using our knowledge gained from studying the previous chapters and the Qt Reference

Documentation on `QStyleSheet` we apply some table magic to create a box aligned to the bottom right corner of `AreaInnermost`. The technique used for this is explained in detail in the Detached Custom Legends sub chapter.

To conclude we add a little green ring marking the predicted apex of this flood wave using some Cell specific Properties and we write the a respective note above that point.

Our little chart now looks like Figure 6.14:

Figure 6.14. Line chart with several Custom Boxes



Compile and run the tutorial file `kdchart_step06d.cpp` to see how KD Chart's different techniques were used to create the chart shown above (see Figure 6.14). Note that things were looking a bit overcrowded so we removed the vertical grid sub lines as we learned in the chapter on Axis Grid Lines.

Tip

Some special effects could be created by chaining several boxes to each other: since *any* area of our chart can be used as reference area for a new box we can simply take the ID of another `KDChartCustomBox` object that we inserted before: this ID is returned by the `insertCustomBox()` method used above. So instead of ignoring the return value we would store it in an `int` variable (e.g. `idFirstBox`) and pass that to the constructor of the new box to be created composing the value of the `area` parameter like this: `KDChartEnums::AreaCustomBoxesBASE + idFirstBox`.

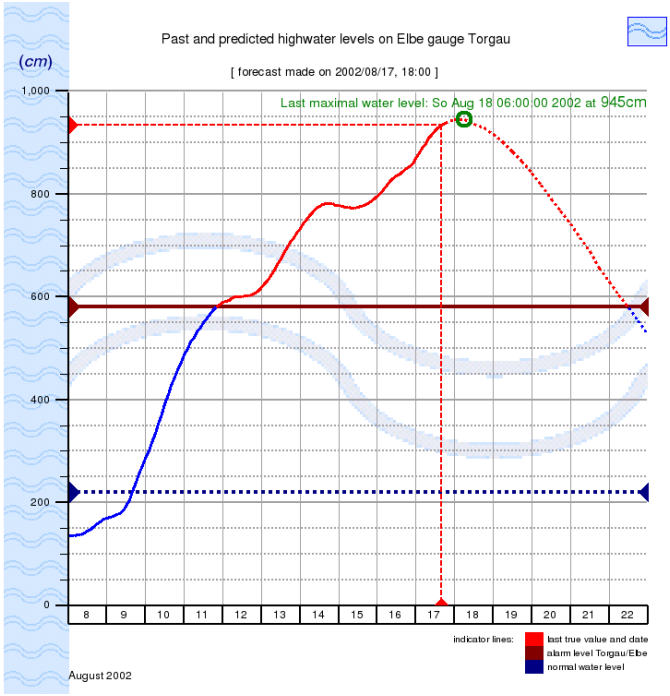
Pictures

KD Chart enables us to beautify our chart by adding pictures using the flexibility of its framing technique described in the chapter on Area Frames. Allowing specification of a `QBrush` and/or a `QPixmap` parameter the handy `KDChart-Params::setSimpleFrame()` method offers two ways of specifying a pixmap:

- For a *tiled* pixmap brush we pass it to the respective `QBrush` constructor `QBrush::QBrush(const QColor& color, const QPixmap& pixmap)` when specifying the `background` parameter of our method.
- To get the pixmap *centered*, *stretched* or *scaled* we use the extra `backPixmap` parameter together with the respective `KDFrame::BackPixmapMode` value for the `backPixmapMode` parameter.

The following version of the chart created in the previous chapters demonstrates some of the various ways how to add a pixmap: Figure 6.15.

Figure 6.15. Line chart with various Pictures



Compile and run the tutorial file `kdchart_step06e.cpp` to see how the framing and custom box techniques were used to create the chart shown above (see Figure 6.15).

Tip

The little picture in the chart's upper right corner shows a special technique useful for adding a company emblem or something similar to any x/y position of your chart:

To get a reference area of your image's size you first insert an auxiliary empty `KDChartCustomBox` object at the desired position and then assign a simple frame to the area of this box, see the chapter on Custom Text Boxes for details on this class. Note that both the pixmap and the border line are specified with the frame as usual, the auxiliary box is completely invisible.

Detached Custom Legends

If you have looked through the previous chapter you have probably wondered how the table like legend box in the lower right corner of the previous screenshot (see Figure 6.15) was defined.

The following describes how to replace the normal legend box by your own custom element.

The idea is simple: hide the built-in legend box and add a `KDChartCustomBox` instead in order to have your own, custom, content shown.

```
// hide the legend
p.setLegendPosition( KDChartParams::NoLegend );

// show a custom box
p.insertCustomBox(
  KDChartCustomBox(
    KDChartTextPiece(
      "<qt><table>"
        "<tr>"
          "<td align=right width=90%>"
             "indicator lines:"
          "</td>"
          "<td>"
            "<table cellspacing=1, cellpadding=0>"
              "<tr>"
                "<td bgcolor=#ff0000>"
                  " "
                "</td>"
                "<td>"
                  "<nobr>last true value and date</nobr>"
                "</td>"
              "</tr>"
              "<tr>"
                "<td bgcolor=#800000>"
                  " "
                "</td>"
                "<td>"
```

```
                "<nobr>alarm level Torgau/Elbe</nobr>"
              "</td>"
            "</tr>"
            "<tr>"
              "<td bgcolor=#000080>"
                " "
              "</td>"
              "<td>"
                "<nobr>normal water level</nobr>"
              "</td>"
            "</tr>"
          "</table>"
        "</td>"
      "</tr>"
    "</table></qt>",
    QFont( "helvetica", 12) ),
  -240,
  false,
  0,
  0,
  -667,
  -80,
  Qt::black,
  Qt::NoBrush,
  KDChartEnums::AreaInnermost,
  KDChartEnums::PosBottomRight,
  Qt::AlignBottom | Qt::AlignRight,
  0,0,0, // no cell number since not aligning to a data point
  KDCHART_AlignAuto,
  true
 )
);
```

The `KDChartCustomBox` is aligned to the bottom right corner of the AreaInnermost, with a relative (per mille of chart size) distance of (667, 80) and a relative font size of 240. It contains a `KDChartTextPiece` showing a bit of QML text, according to the documentation of the `QSimpleRichText` class, matching the default `QStyleSheet` class.

Grid

Note

Grid lines are controlled by their respective axis: horizontal lines by a vertical axis (by default the left ordinate) and vertical lines by a horizontal axis (by default the bottom abscissa), therefor the functions for manipulating them are part of the axis methods and they are explained in this context, so please have a look at the Grid Lines chapter for detailed information.

Embedding Your Chart

You can embed KD Chart's drawing code into the output of your applications in the following two ways:

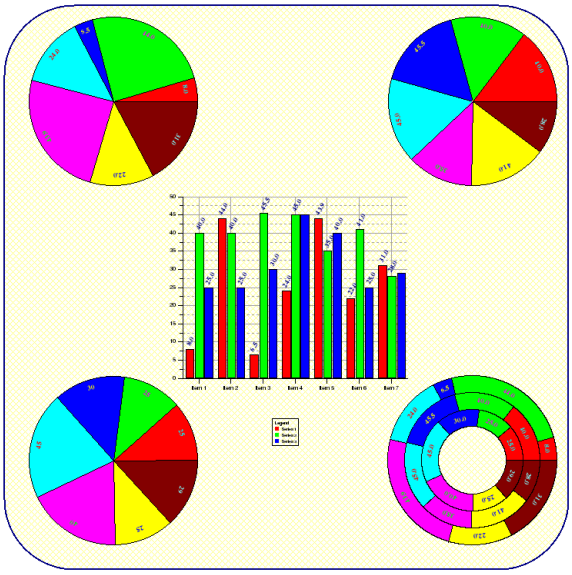
- Make KD Chart draw into your `QPainter` at a position and size specified by you.
- Add your drawings to the area of the chart before or after KD Chart draws itself.

Let us see how these two techniques can be used to fine-tune your charts and to add custom content to them.

Let KD Chart use your `QPainter`

KD Chart lets you for embed charts into the painting code of your program by using `KDChart::paint()` rather than adding a `KDChartWidget` to your layout. The following program shows how to do this: Figure 6.16.

Figure 6.16. Drawing charts into your QPainter



Compile and run the tutorial file `kdchart_step02b.cpp` to see how the charts are drawn in the method `TutorialWidget::paintEvent()` (see Figure 6.16).

Access KD Chart's Geometry Information

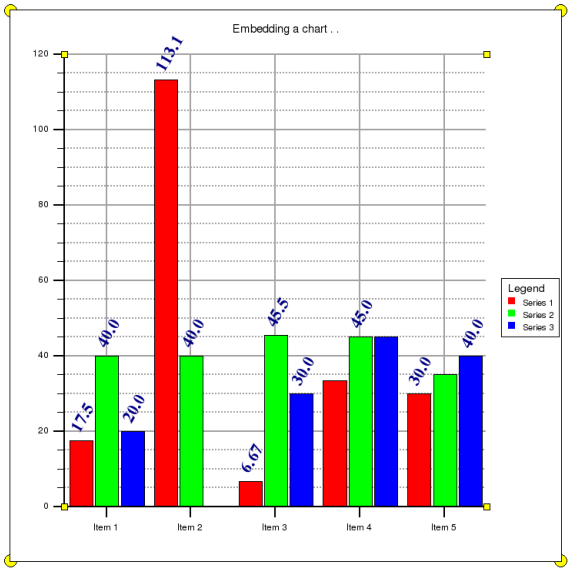
For adding custom content, KD Chart lets you access the drawing area and provides you with the geometry information of the chart by means of two methods in the KD Chart API:

`KDChart::setupGeometry()` and `KDChart::painterToDrawRect()` which you can use together with `KDChart::paint()` to access the respective parts of your `QPainter` before or after KD Chart draws the chart(s).

The following sample program demonstrates how to use your own drawing code before and after KD Chart paints the chart: Figure 6.17.

First we draw some circles on the corners of the `AreaOutermost` area, then we let KD Chart execute its own painting code, and finally we add some rectangles on the corners of the `AreaData` area:

Figure 6.17. Add Custom Drawing Code to a Chart



Compile and run the tutorial file `kdchart_step02c.cpp` to learn how to obtain geometry information from KD Chart, e.g. in order to add your own drawing code to the data area. (see Figure 6.17).

Using Your Own Data Structures

Starting with version 1.1 of KD Chart, you can directly use your own data structures without the need of passing data values into an instance of `KDChartTableData`. You can thus modify or add both data values and/or cell property IDs using your own algorithms without calling `KDChartTableData::setCell()`. This enables tight integration of KD Chart with your program and avoids expensive data duplication and copying operations at run-time.

To use this feature, just declare your own data handling class that inherits KD Chart's abstract base class `KDChartTableDataBase`. Make sure that your class implements the following pure virtual methods:

- `uint rows() const` *returns* the number of data rows currently stored in your data structure.
- `uint cols() const` *returns* the number of data columns currently stored in your data structure.
- `bool cellCoord( uint row, uint col, QVariant& _value, int coordinate=1 ) const` *retrieves* one of the values stored in the data cell adressed by (row,col).

  Note that coordinate 1 is addressing the Y value while coordinate 2 can be used to retrieve the X value, if any. The method returns true if row and col specify a valid cell location.
- `bool cellProp( uint row, uint col, int& _prop ) const` *retrieves* the ID of the PropertySet assigned to the data cell adressed by (row,col).

  In case you do not want to use cell-specific properties, make sure to return *Zero* for all cells, and KD Chart will use the built-in default property set. The method returns true if row and col specify a valid cell location.
- `void setCell( uint row, uint col, const QVariant& _value1, const QVariant& _value2=QVariant() )` *specifies* one or both of the values to be stored in the data cell addressed by (row,col).
- `setProp( uint row, uint col, int _propSet=0 )` *specifies* the ID of the PropertySet to be assigned to the data cell addressed by (row,col).
- `void expand( uint rows, uint cols )` *increases* the number of rows and/or columns stored in this table.

The sample program `kdchart_step07.cpp` demonstrates how to create your own data handling class by inheriting from `KDChartTableDataBase`.

Compile and run the tutorial file `kdchart_step07.cpp` to see how it works, and feel free to use this code as a starting point for your own implementation.

Tip

For further speeding up the geometry calculation and data visualization in your charts, you can use the `KDChartTableDataBase` methods `setUsedRows()` and `setUsedCols()` to inform KD Chart about the number of data cells (or rows) that are actually being used.

Chapter 7. Interactive Charts

Providing more than a static charting engine KD Chart offers not only a broad field of options to customize, enrich and fine tune your charts as shown in the previous chapters, but it also allows for different ways of interaction with your users:

While the optional Reporting Mouse Events feature enables your application to perform its own actions when the user clicks on a data bar (or a pie slice, a line marker, etc) it is also possible to use the options provided by the `QPainter` class for Scrolling And Zooming your chart.

Reporting Mouse Events

KD Chart allows you to let the user to some extent interact with the displayed charts. In order to make use of this, it is easiest to use `KDChartWidget` and connect to one or more of the nine signals it emits. Each signal represents one combination of a mouse action (press, release, click) and a mouse button (left, middle, right). The respective signal is emitted when the user clicks on one of the data regions in the chart.

Since the computation of the data regions is quite expensive at runtime, it is turned off by default. If you want to make use of the signals, make sure to first to set the `activeData` property to `true`. The easiest way to do this is by calling the `KDChartWidget` function `setActiveData( true )`.

If you want full flexibility, you can also access the internal data regions by calling `KDChartWidget::dataRegions()`. This method returns a pointer to a list of `KDChartDataRegion` structures, each of which contains a lot of information about the data region it represents, including the `QRegion` object representing the geometry of the region as well as a `row` and `col` value pair indicating the data cell which the respective region belongs to.

If you do not use `KDChartWidget` but rather call `KDChart::paint()` directly the data region list is returned in the fourth parameter.

Note

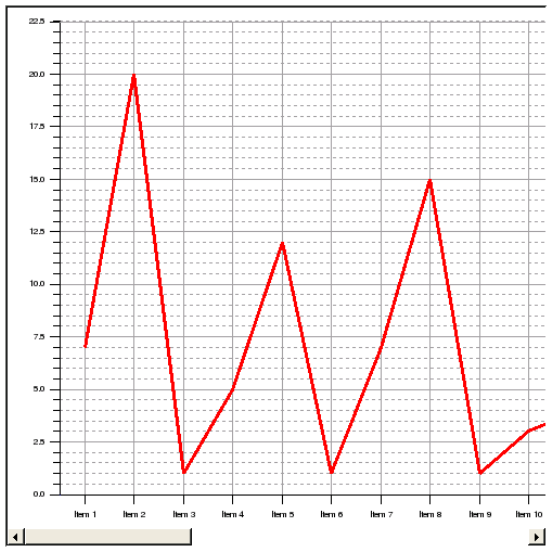
Please note that this list of regions is only valid until your chart is repainted the next time.

Scrolling And Zooming

KD Chart's consequent use of the `QPainter` class makes it easy to scroll and zoom the resulting graph: just embed your `KDChartWidget` into a `QScrollView`. The resulting chart could look like the one shown in the screenshot below (see Figure 7.1). This chart

is zoomed in horizontal direction by a factor of 3.0 and it is also embedded into a `QScrollView` scrolling in horizontal direction:

Figure 7.1. Scrolling And Zooming



Compile and run the tutorial file `kdchart_step09.cpp` to see how the scrolling widget shown in Figure 7.1 was set up: For your convenience, this tutorial file introduces a separate `KDChartScrollView` class taking care of both the scrolling area and the zooming of the `KDChartWidget` contained therein.

Feel free to reuse the `KDChartScrollView` class and integrate it into your own application in original or modified form, it is just a simple suggestion on how things could be done.

Future versions of KD Chart will allow further interaction features that will make it possible to drive your application from the chart.

Chapter 8. Multiple Charts

There are two ways to display multiple charts, i.e., several chart types in one display:

1. Combining two charts very tightly by displaying them together in one `KDChartWidget` sharing the same abscissa axis.

   Requiring some axis configuration this feature is described in the context of the instructions given on axes, please have a look at the chapter on Combining Two Charts for details.

2. Using separate `KDChartWidget` objects that are positioned next to each other: using a `QGridLayout` would arrange them automatically.

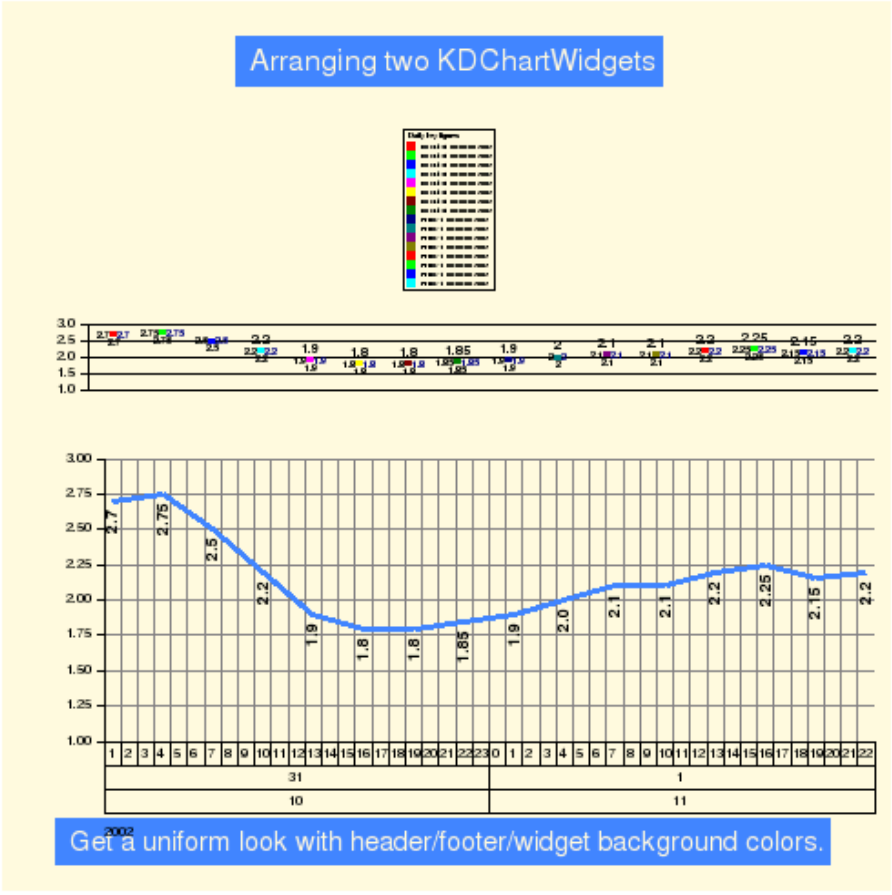
   You might position two charts above each other and make them look like a coherent unit by enabling one or several of the top chart's Header Areas and drawing a simple Area Frame around its `AreaHeaders` proceeding accordingly with the bottom chart's footer(s).

   Same colors for the top chart's headers area and the bottom chart's footers area would and a common background color for both of your `KDChartWidget` instances would make them look like they were one widget.

   Let us look at a little application based upon the program developed in Chapter 2, *Three Steps to Your Chart* at the beginning of this manual. Using some of KD Chart's layout options we can make two widgets look like they were just a single one.

Compile and run the tutorial file `kdchart_step08.cpp` to see how easy it is to obtain good looking results as shown in Figure 8.1. Note that made the top chart's abscissa axis labels invisible: this was done for design reasons solely and is not necessary.

Figure 8.1. Arranging two KDChartWidgets



The source code that was used to produce the screen shot shown above is listed here for your convenience:

```
 1 /* -*- Mode: C++ -*-
      KDChart - a multi-platform charting engine
   */

 5 /****************************************************************************
   ** Copyright (C) 2001-2003 Klar#lvdalens Datakonsult AB.
   ** All rights reserved.
   **
   ** This file is part of the KDChart library.
10 **
   ** This file may be distributed and/or modified under the terms of the
   ** GNU General Public License version 2 as published by the Free Software
   ** Foundation and appearing in the file LICENSE.GPL included in the
   ** packaging of this file.
```

```
** 
** Licensees holding valid commercial KDChart licenses may use this file
** in accordance with the KDChart Commercial License Agreement provided
** with the Software.
**
** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
**
** See http://www.klaralvdalens-datakonsult.se/?page=products for
**   information about KDChart Commercial License Agreements.
**
** Contact info@klaralvdalens-datakonsult.se if any conditions of this
** licensing are not clear to you.
**
**********************************************************************/
#include "kdchart_step08.h"

#include <iostream>
#include <qapplication.h>
#include <qtimer.h>
#include <qlayout.h>
#include <qlabel.h>
#include <qpainter.h>
#include <qprinter.h>
#include <qpaintdevicemetrics.h>
#include <qpicture.h>

#include <KDChartWidget.h>
#include <KDChartTable.h>
#include <KDChartParams.h>
#include <KDChartAxisParams.h>

TutorialWidget::~TutorialWidget()
{
}

void TutorialWidget::saveAndClose()
{
    QPixmap picture( QPixmap::grabWindow( winId() ) );
    picture.save("image.png", "PNG", 100);
    close();
}

int main ( int argc, char* argv[] )
{
    QApplication a ( argc, argv );

    // declare the main widget before processing the command line
    // parameters, so we can connect a timer shot to the widget...
    TutorialWidget* mainWidget = new TutorialWidget( QSize(520,520) );

    // process our parameters to enable screenshot mode possibly
    bool bScreenshotMode = false;
    for ( int i = 1; i < a.argc(); ++i ){
        QString s = QString(argv[i]).upper();
        if( s == "-S" ){
            if( !bScreenshotMode ){

                // we make a screenshot of the widget's content and quit
                QTimer::singleShot(2000,mainWidget,SLOT(saveAndClose()));
                bScreenshotMode = true;
            }
        }else if( s == "-?" || s == "/?" || s == "-H" || s == "/H" ||
                  s == "-HELP" || s == "/HELP" || s == "--HELP" ){

            // we show the help text and end the program
            std::cout << "\n\n" << argv[0]
            << " -s      saves the widget as file image.png and quits\n";
            return 0;
        }
```

```
    }

    // ********************************************************************
    // Main Widget Layout
    // ********************************************************************
        // make it compile on windows using qt232
        #if QT_VERSION >= 200 && QT_VERSION < 300
        mainWidget->setPalette( bScreenshotMode
                                                    ? Qt::white
                                                    : Qt::lightGray );
        #else
    mainWidget->setPaletteBackgroundColor( bScreenshotMode
                                       ? Qt::white
                                       : Qt::lightGray );
        #endif

    mainWidget->setCaption( "KDChart Tutorial - Step 8" );
    QVBoxLayout theLayout( mainWidget );
    theLayout.setMargin( 16);
    theLayout.setSpacing(16);
    if( !bScreenshotMode ){
        QLabel* label1 =
            new QLabel("arranging two KDChartWidget objects:", mainWidget);
                    // make it compile on windows using qt232
                #if QT_VERSION >= 200 && QT_VERSION < 300
                label1->setPalette( Qt::white );
                #else
                label1->setPaletteBackgroundColor( Qt::white );
                #endif

        theLayout.addWidget( label1, 0 );
    }


    // common values for both widgets
    QColor chamois(   255,248,222 );
    QColor aquamarin( 66,135,255 );


    // ------------------------------------------------------------
    // Declaring chart #1 which will be positioned at the bottom
    // ------------------------------------------------------------


    // ********************************************************************
    // Chart #1: Params
    // ********************************************************************

    KDChartParams p1;
    p1.setGlobalLeading(  7,7, 7,0 );
    p1.setSimpleFrame( KDChartEnums::AreaFooters,
                       0,0,  0,0,
                       true,
                       true,
                       KDFrame::FrameFlat,
                       1,
                       0,
                       Qt::NoPen,
                       QBrush( aquamarin ) );
    p1.setHeaderFooterText( KDChartParams::HdFtPosFooter,
            " Get a uniform look with header/footer/widget background colors. " );
//          " Use header/footer and widget colors for an intergrated look. " );
    p1.setHeaderFooterFont( KDChartParams::HdFtPosFooter,
                            QFont( "helvetica",1 ),
                            true,
                            68 );
    p1.setHeaderFooterColor( KDChartParams::HdFtPosFooter,
                             chamois );

    p1.setChartType( KDChartParams::Line );
```

```
    p1.setLineWidth( 3 );

    QFont dataValuesFont1( "helvetica", 10, QFont::Bold );
    QColor textColor1( Qt::black );
    p1.setPrintDataValues( true,
            KDCHART_ALL_CHARTS,
            0,
            KDCHART_DATA_VALUE_AUTO_DIGITS,
            &dataValuesFont1,
            35,
            KDCHART_DATA_VALUE_AUTO_COLOR,
            KDChartEnums::PosTopCenter,
            Qt::AlignLeft | Qt::AlignVCenter,
            0,
            33,
            270,
            KDChartEnums::PosBottomCenter,
            Qt::AlignRight | Qt::AlignVCenter,
            0,  // X gap is zero
            33, // Y gap is 100/33 of the font height
            270 );

    p1.setPrintDataValuesColor(
            KDCHART_ALL_CHARTS,
            &textColor1 );

    p1.setDataColor( 0, aquamarin );

    p1.setLegendPosition( KDChartParams::NoLegend );

    KDChartAxisParams pa;
    pa = p1.axisParams( KDChartAxisParams::AxisPosBottom);
    pa.setAxisGridStyle( Qt::SolidLine );
    pa.setAxisLineColor( Qt::black );
    pa.setAxisGridColor( Qt::darkGray );
    pa.setAxisShowGrid( true );
    pa.setAxisShowSubDelimiters( false );
    p1.setAxisParams( KDChartAxisParams::AxisPosBottom, pa );

    pa = p1.axisParams( KDChartAxisParams::AxisPosLeft );
    pa.setAxisGridStyle( Qt::SolidLine );
    pa.setAxisLineColor( Qt::black );
    pa.setAxisGridColor( Qt::darkGray );
    pa.setAxisShowGrid( true );
    pa.setAxisShowSubDelimiters( false );
    pa.setAxisValueStart( 1.0 );
    pa.setAxisValueEnd(   3.0 );
    p1.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );


    // ********************************************************************
    // Chart #1: Table Data
    // ********************************************************************

    // A real world application would retrieve its the data from some
    // external source calculating their number dynamically,
    // here we just use 16 random values.
    // Note that the values are ordered by date/time  -  otherwise
    // the line would run backwards at places where a value has an
    // earlier time than its preceeding value.
    KDChartTableData d1( 1, 16 );
    d1.setCell( 0, 0,  2.7,
                       QDateTime(QDate(2002,10,31),QTime( 1,30,0)) );
    d1.setCell( 0, 1,  2.75,
                       QDateTime(QDate(2002,10,31),QTime( 4,30,0)) );
    d1.setCell( 0, 2,  2.5,
                       QDateTime(QDate(2002,10,31),QTime( 7,30,0)) );
    d1.setCell( 0, 3,  2.2,
                       QDateTime(QDate(2002,10,31),QTime(10,30,0)) );
    d1.setCell( 0, 4,  1.9,
```

```
                          QDateTime(QDate(2002,10,31),QTime(13,30,0)) );
    d1.setCell( 0, 5,  1.8,
                          QDateTime(QDate(2002,10,31),QTime(16,30,0)) );
    d1.setCell( 0, 6,  1.8,
                          QDateTime(QDate(2002,10,31),QTime(19,30,0)) );
    d1.setCell( 0, 7,  1.85,
                          QDateTime(QDate(2002,10,31),QTime(22,30,0)) );
    d1.setCell( 0, 8,  1.9,
                          QDateTime(QDate(2002,11, 1),QTime( 1,30,0)) );
    d1.setCell( 0, 9,  2.0,
                          QDateTime(QDate(2002,11, 1),QTime( 4,30,0)) );
    d1.setCell( 0,10,  2.1,
                          QDateTime(QDate(2002,11, 1),QTime( 7,30,0)) );
    d1.setCell( 0,11,  2.1,
                          QDateTime(QDate(2002,11, 1),QTime(10,30,0)) );
    d1.setCell( 0,12,  2.2,
                          QDateTime(QDate(2002,11, 1),QTime(13,30,0)) );
    d1.setCell( 0,13,  2.25,
                          QDateTime(QDate(2002,11, 1),QTime(16,30,0)) );
    d1.setCell( 0,14,  2.15,
                          QDateTime(QDate(2002,11, 1),QTime(19,30,0)) );
    d1.setCell( 0,15,  2.2,
                          QDateTime(QDate(2002,11, 1),QTime(22,30,0)) );


    // ------------------------------------------------------
    // Declaring chart #2 which will be positioned at the top
    // ------------------------------------------------------


    // ********************************************************************
    // Chart #2: Params
    // ********************************************************************

    KDChartParams p2;
    p2.setGlobalLeading(  7,7, 0,7 );
    p2.setSimpleFrame( KDChartEnums::AreaHeaders,
                       0,0,  0,0,
                       true,
                       true,
                       KDFrame::FrameFlat,
                       1,
                       0,
                       Qt::NoPen,
                       QBrush( aquamarin ) );
    p2.setHeaderFooterText( KDChartParams::HdFtPosHeader,
                            " Arranging two KDChartWidgets " );
    p2.setHeaderFooterFont( KDChartParams::HdFtPosHeader,
                            QFont( "helvetica",1 ),
                            true,
                            72 );
    p2.setHeaderFooterColor( KDChartParams::HdFtPosHeader,
                             chamois );

    p2.setChartType(        KDChartParams::HiLo );
    p2.setHiLoChartSubType( KDChartParams::HiLoOpenClose );
    p2.setLineWidth( 3 );

    p2.setLegendPosition( KDChartParams::LegendTop );
    p2.setLegendTitleText( "Daily key figures" );

    QFont dataValuesFont2( "helvetica", 10, QFont::Bold );
    QColor textColor2( Qt::black );
    QColor textColor2close( Qt::darkBlue );
    p2.setHiLoChartPrintOpenValues( true,
            &dataValuesFont2,
            20,
            &textColor2 );
    p2.setHiLoChartPrintCloseValues( true,
            &dataValuesFont2,
```

```
            20,
            &textColor2close );
    p2.setHiLoChartPrintHighValues( true,
            &dataValuesFont2,
            25,
            &textColor2 );
    p2.setHiLoChartPrintLowValues( true,
            &dataValuesFont2,
            20,
            &textColor2 );

    pa = p2.axisParams( KDChartAxisParams::AxisPosBottom );
    pa.setAxisLineColor( Qt::black );
    pa.setAxisShowGrid( false );
    pa.setAxisLabelsVisible( false );
    p2.setAxisParams( KDChartAxisParams::AxisPosBottom, pa );

    pa = p2.axisParams( KDChartAxisParams::AxisPosLeft );
    pa.setAxisGridStyle( Qt::SolidLine );
    pa.setAxisGridLineWidth(KDCHART_AXIS_GRID_AUTO_LINEWIDTH );
    pa.setAxisLineColor( Qt::black );
    pa.setAxisGridColor( pa.axisLineColor() );
    pa.setAxisShowGrid( true );
    pa.setAxisShowSubDelimiters( true );
    pa.setAxisValueStart( 1.0 );
    pa.setAxisValueEnd(   3.0 );
    pa.setAxisValueDelta( 0.5 );
    p2.setAxisParams( KDChartAxisParams::AxisPosLeft, pa );

    p2.setLegendSource( KDChartParams::LegendManual );

    // ********************************************************************
    // Chart #2: Table Data
    // ********************************************************************

    // we start with a one-day table, it will be expanded when necessary
    KDChartTableData d2( 1, 4 );  //  low/high/open/close need 4 columns

    QDate date;
    double lowValue  =0.0;// initializing avoids ugly compile-time warnings
    double highValue =0.0;
    double closeValue=0.0;
    int iDay = -1;
    int numVals = d1.cols();
    for( int iVal = 0; iVal < numVals; ++iVal ){

        QVariant valueY, valueX;
        if( d1.cellCoords( 0, iVal, valueY, valueX ) &&
            QVariant::Double   == valueY.type() &&
            QVariant::DateTime == valueX.type() ){

            const double value(valueY.toDouble()   );
            QDateTime    date( valueX.toDateTime() );
            if( date.isNull() ||
                date != valueX.toDateTime().date() ){
                date  = valueX.toDateTime().date();

                // set the day to its current number
                ++iDay;

                // set the legend text for this new day
                p2.setLegendText( iDay, date.toString() );

                // increase the table if this is not the very first day
                if( iDay )
                    d2.expand( d2.rows()+1, 4 );

                // store this day's open value
                d2.setCell( iDay, 2, value );
```

```
                // store the previous day's low/high/close values if any
                if( iDay ){
                    d2.setCell( iDay-1, 0, lowValue );
                    d2.setCell( iDay-1, 1, highValue );
                    d2.setCell( iDay-1, 3, closeValue );
                }
                lowValue  = value;
                highValue = value;
            }else{
                lowValue  = QMIN( lowValue,  value );
                highValue = QMAX( highValue, value );
            }
            closeValue = value;
        }
    }

    // store the last day's low/high/close values
    if( iDay ){
        d2.setCell( iDay, 0, lowValue );
        d2.setCell( iDay, 1, highValue );
        d2.setCell( iDay, 3, closeValue );
    }


    // ********************************************************************
    // Create the Layout, insert the Chart Widgets and run the program
    // ********************************************************************
    QGridLayout chartGridLayout( &theLayout, 2,1 );
    chartGridLayout.setMargin( 0);
    chartGridLayout.setSpacing(0);

    theLayout.setStretchFactor(&chartGridLayout, 1);

    KDChartWidget *chartWidget_1 = new KDChartWidget(&p1,&d1, mainWidget);
    KDChartWidget *chartWidget_2 = new KDChartWidget(&p2,&d2, mainWidget);
        // make it compiles on windows using qt232
        #if QT_VERSION >= 200 && QT_VERSION < 300
        chartWidget_1->setPalette( chamois );
        chartWidget_2->setPalette( chamois );
        #else
    chartWidget_1->setPaletteBackgroundColor( chamois );
    chartWidget_2->setPaletteBackgroundColor( chamois );
        #endif
    chartGridLayout.addWidget( chartWidget_1, 1,0 );
    chartGridLayout.addWidget( chartWidget_2, 0,0 );

    a.setMainWidget( mainWidget );

    mainWidget->show();

    return a.exec();
}
```