WxSmith tutorial: Adding advanced properties into items

From Code::Blocks

Contents

Preface

In previous tutorial we learned how to add some basic properties into our custom item. In this tutorial we will learn how to create fully customizable properties by operating directly on wxProperytGrid and xml structures. Note that we will use wxPropertyGrid (http://wxpropgrid.sourceforge.net/) and TinyXml (http://www.grinninglizard.com/tinyxml/) libraries here so I recommend you get familliar with them first.

Let's take a look how wxSmith deals with fully customizable properties:

Fully customizable properties

To create fully costomizable property we have to handle few operations done on it:

- Adding property to property grid
- Reacting to change of this property
- Reading data from XML (XRC) structures
- Writing data to XML (XRC) structures

Each of these operations is done inside one virtual function of class which adds item into wxSmith (like wxsChart in our chart example). Overriding it and giving custom implementation allows us to freely operate on properties.

Adding custom property to property grid

To include our own properties in list of properties, we have to add following function into our class:

```
void OnAddExtraProperties(wxsPropertyGridManager* Grid);
```

Inside this class we can freely operate on given `wxsPropertyGridManager` (see wxPropertyGrid site (http://wxpropgrid.sourceforge.net/) to learn more on how to operate on this class), but the operations should be limited to adding properties and responding for property change (it may be risky to delete or change existing ones).

At the end of custom implementation of this function, we should also call the original function declared inside wxsWidget class:

```
void CustomClass::OnAddExtraProperties(wxsPropertyGridManager* Grid)
{
    // Add custom properties here

    wxsWidget::OnAddExtraProperties(Grid);
}
```

If it won't be called, item wont allow to edit any events because they are added inside `wxsWidget::OnAddExtraProperties(Grid);` call.

Reacting to change of custom properties

Processing changes of custom properties is simillar to creating them. We just have to override following function:

```
void OnExtraPropertyChanged(wxsPropertyGridManager* Grid,wxPGId Id);
```

It has extra parameter comparing to previous function - Id. It's id of property which was changed. It's advised that you compare value of Id with identifiers generated inside `OnAddExtraProperties` to avoid unnecessary reads from property grid. At the end of this function you should also call original one but only if you're sure that your properties have not changed:

```
void CustomClass::OnExtraPropertyChanged(wxsPropertyGridManager* Grid,wxPGId Id)
{
    if ( Id == Property_Id )
    {
        // Read value of property here

        NotifyPropertyChange(true);
        return;
    }

    wxsWidget::OnExtraPropertyChanged(Grid,Id);
}
```

Note that there's also call to `NorifyPropertyChange(true);` at the end of value reading. This call updates all things related to current resource: regenerates source code and XRC files if necessary, updates the content of editor and does few more things to

keep wxSmith's state up to date.

Reading data from XML (XRC) structures

Reading of our custom properties is done inside following function:

```
bool OnXmlRead(TiXmlElement* Element,bool IsXRC,bool IsExtra);
```

The Element argument is object represending xml node of current widget. To load custom properties we must locate child nodes and read data from them. Code::Blocks and wxSmith are using TinyXml (http://www.grinninglizard.com/tinyxml/) to operate on xml structures so to get more informations on how to read and manipulate xml data, read TinyXml's documentation (http://www.grinninglizard.com/tinyxmldocs/index.html).

The `IsXRC` and `IsExtra` arguments require some better description. From the very beginning wxSmith tends to be compatible with XRC files which allow storing structure of window inside xml file. Usually XRC files are limited to widgets which are provided with wxWidgets library. And they don't provide some extra data used by wxSmith like enteries for event handlers and variable name. wxSmith does split data of widgets into two parts - the firs one is strict XRC data and the second one is extra data provided by wxSmith. If `IsXRC` parameter is true, this mean that function should read XRC data part, if `IsExtra` is true, this mean that it should read extra data (both arguments may be true in one call). Most of contrib items would have XRC support disabled so we should read data only when IsExtra is true:

```
bool CustomClass::OnXmlRead(TiXmlElement* Element,bool IsXRC,bool IsExtra)
{
    if ( IsExtra )
    {
        // Process xml structures here
    }

    return wxsTool::OnXmlRead(Element,IsXRC,IsExtra);
}
```

At the end of this implementation we must call original function as usual ;)

Writing data to XML (XRC) structures

Writing data into xml structures is simillar to reading them. We have to implement following function:

```
bool CustomClass::OnXmlWrite(TiXmlElement* Element,bool IsXRC,bool IsExtra)
{
    if ( IsExtra )
    {
        // Store data into xml structure
    }

    return wxsTool::OnXmlRead(Element,IsXRC,IsExtra);
}
```

The meaning of arguments is same as in case of reading data. The only difference is that we write data here instead of reading it.

Adding custom property into wxChart

Now it's time to do some really complex task. We will add property which adds some data into chart widget so it would be possible to show some content in editor, preview and of course in target application. But before we do this, we need some knowledge about wxChart's data managment.

How wxChartCtrl handle data for properties

Each wxChartCtrl widget can show few sets of data which create one chart shape (one pie, one line etc). Each of these sets is stored inside class derived from wxChartPoints and the class type depends on type of chart we want to show. This structure keep data of points, percentages or any other data required to build chart. Following chart types are available in wxChart:

- Bar
- Bar3D
- Pie
- Pie3D
- Points
- Points3D
- Line
- Line3D
- Area
- Area3D

Different data types may be stored inside one chart which will result in many charts shown inside one control.

Now let's add some property.

Adding dynamically changing properties set

First thing we need here is to provide dynamically changing list of data sets (wxChartPoints structures). Because it's not as easy as in case of base properties, we have to handle our properties manually. So first we add new functions which will manage our custom properties.

We add this code into wxsChart's declaration:

```
        void OnAddExtraProperties(wxsPropertyGridManager* Grid);
        void OnExtraPropertyChanged(wxsPropertyGridManager* Grid,wxPGId Id);
        bool OnXmlRead(TiXmlElement* Element,bool IsXRC,bool IsExtra);
        bool OnXmlWrite(TiXmlElement* Element,bool IsXRC,bool IsExtra);
```

and the following initial implementations:

```
void wxsChart::OnAddExtraProperties(wxsPropertyGridManager* Grid)
{
    wxsWidget::OnAddExtraProperties(Grid);
```

```
}

void wxsChart::OnExtraPropertyChanged(wxsPropertyGridManager* Grid,wxPGId Id)
{
    wxsWidget::OnExtraPropertyChanged(Grid,Id);
}

bool wxsChart::OnXmlRead(TiXmlElement* Element,bool IsXRC,bool IsExtra)
{
    return wxsWidget::OnXmlRead(Element,IsXRC,IsExtra);
}

bool wxsChart::OnXmlWrite(TiXmlElement* Element,bool IsXRC,bool IsExtra)
{
    return wxsWidget::OnXmlWrite(Element,IsXRC,IsExtra);
}
```

These implementations are just calling original functions to let wxSmith do it's work inside of them.

Another thing we need before we start implementing new property is some place where we will store chart's data. It must of course be put into wxsChart class. We need some array describing wxChartPoints classes and probaly some structures describing points. But now let's put wxPGId value for each wxChartPoints class. This id will point to parent property which will handle other wxChartPoints properties inside:

```
        struct ChartPointsDesc
        {
            wxPGId Id;
        };

        WX_DEFINE_ARRAY(ChartPointDesc*,List);

        long m_Flags;
        List m_ChartPointsDesc;
        wxPGId m_ChartPointsCountId;
};
```

Currently each set of chart points is described by property id, but it will be extended later. Another property id which was added here (m_ChartPointsCountId) is used to handle count of wxChartPoints classes. Now we will add this property into property browser and will implement all the dynamics it does (adding / removing other properties).

First we have to update code generating custom properties, we do this by adding one property handling number of data sets and some properties per each set:

```
void wxsChart::OnAddExtraProperties(wxsPropertyGridManager* Grid)
{
    //   (1)
    Grid->SetTargetPage(0);

    //   (2)
    m_ChartPointsCountId = Grid->Append(wxIntProperty(_("Number of data sets"),wxPG_LABEL,
                                                      (int)m_ChartPointsDesc.Count()));

    //   (3)
    for ( int i=0; i<(int)m_ChartPointsDesc.Count(); i++ )
    {
        AppendPropertyForSet(Grid,i);
    }

    wxsWidget::OnAddExtraProperties(Grid);
}
```

In this code we switch to first page in property grid first *(1)* (this prevents adding properties to other pages because we can not be sure about current page), next we add number of sets *(2)* and then add properties for each set *(3)*, wchich will be done in AppendPropertyForSet function.

Now let's handle changes done in current properties:

```
void wxsChart::OnExtraPropertyChanged(wxsPropertyGridManager* Grid,wxPGId Id)
{
    //   (1)
    Grid->SetTargetPage(0);

    //   (2)
    if ( Id == m_ChartPointsCountId )
    {
        int OldValue = (int)m_ChartPointsDesc.Count();
        int NewValue = Grid->GetPropertyValueAsInt(Id);

        //   (3)
        if ( NewValue<0 )
        {
            NewValue = 0;
            Grid->SetPropertyValue(Id,NewValue);
        }

        if ( NewValue > OldValue )
        {
            //   (4)

            // We have to generate new entries
            for ( int i=OldValue; i<NewValue; i++ )
            {
                m_ChartPointsDesc.Add(new ChartPointsDesc());
                AppendPropertyForSet(Grid,i);
            }
        }
        else if ( NewValue < OldValue )
        {
            //   (5)

            // We have to remove some entries
            for ( int i=NewValue; i<OldValue; i++ )
            {
                Grid->Delete(m_ChartPointsDesc[i]->Id);
                delete m_ChartPointsDesc[i];
            }

            m_ChartPointsDesc.RemoveAt(NewValue,OldValue-NewValue);
        }

        //   (6)
        NotifyPropertyChange(true);
        return;
    }

    //   (7)
    for ( int i=0; i<(int)m_ChartPointsDesc.Count(); i++ )
    {
        if ( HandleChangeInSet(Grid,Id,i) ) return;
    }

    wxsWidget::OnExtraPropertyChanged(Grid,Id);
}
```

This function is much more complicated since it must dynamically add and remove properties from property grid. In this code we first switch to base page as in previous function *(1)*, then check if number of sets was changed *(2)*. If yes, we read new number of sets and make sure it's not negative *(3)* because user may give any value including negative ones which could cause some crash. Then we check if new number of sets is greater than current one and add new enteries if necessary *(4)* and also check if we have to remove some sets when new number of sets is smaller than

current one *(5)*. If we have updated proserties sets, we call `NotifyPropertyChange(true);` which will update editor's area, regenerate source code and update all stuff in wxSmith.

If it was not number of sets that changed, we try to update individual property sets by calling `HandleChangeInSet` function.

Note that when we remove sets in *(5)* we have to call `delete m_ChartPointsDesc[i];` to avoid memory leaks because it won't be deleted automatically. Because of same reason, we also have to alter destructor which will free memory used by wxsChart class:

```
wxsChart::~wxsChart()
{
    for ( size_t i=0; i<m_ChartPointsDesc.Count(); i++ )
    {
        delete m_ChartPointsDesc[i];
    }
    m_ChartPointsDesc.Clear();
}
```

Now let's implement functions functions opreating on individual property sets: `AppendPropertyForSet` and `HandleChangeInSet`. But before we do that, we need to update ChartPointsDesc structure a little bit.

Each chart points set has few global properties: name, type, colour and switch for showing label. Right now let's just add name and type since they are required to build wxChartPoints class. Also let's build structure for data of one point:

```
        struct PointDesc
        {
            wxString Name;
            double X;
            double Y;

            wxPGId Id;
            wxPGId NameId;
            wxPGId XId;
            wxPGId YId;
        };

        WX_DEFINE_ARRAY(PointDesc*,PointList);

        enum PointsType
        {
            Bar,
            Bar3D,
            Pie,
            Pie3D,
            Points,
            Points3D,
            Line,
            Line3D,
            Area,
            Area3D
        };

        struct ChartPointsDesc
        {
            wxPGId Id;
            wxPGId TypeId;
            wxPGId NameId;
            wxPGId PointsCountId;

            PointsType Type;
            wxString   Name;
            PointList  Points;
```

```
            ChartPointsDesc(): Type(Bar)
            {}

            ~ChartPointsDesc()
            {
                for ( size_t i=0; i<Points.Count(); i++ )
                {
                    delete Points[i];
                }
                Points.Clear();
            }
        };
```

Ok, now let's create code which will generate properties for set:

```
void wxsChart::AppendPropertyForSet(wxsPropertyGridManager* Grid,int Position)
{
    ChartPointsDesc* Desc = m_ChartPointsDesc[Position];
    wxString SetName = wxString::Format(_("Set %d"),Position+1);

    //   (1)
    Desc->Id = Grid->Append(wxParentProperty(SetName,wxPG_LABEL));

    //   (2)
    static const wxChar* Types[] =
    {
        _T("Bar"),    _T("Bar3D"),    _T("Pie"),  _T("Pie3D"),
        _T("Points"), _T("Points3D"), _T("Line"), _T("Line3D"),
        _T("Area"),   _T("Area3D"),   NULL
    };

    static const long Values[] =
    {
        Bar, Bar3D, Pie, Pie3D, Points, Points3D, Line, Line3D, Area, Area3D
    };

    //   (3)
    Desc->TypeId = Grid->AppendIn(Desc->Id,wxEnumProperty(_("Type"),wxPG_LABEL,Types,Values,Desc->Type));
    Desc->NameId = Grid->AppendIn(Desc->Id,wxStringProperty(_("Name"),wxPG_LABEL,Desc->Name));
    Desc->PointsCountId = Grid->AppendIn(Desc->Id,wxIntProperty(_("Number of points"),wxPG_LABEL,
(int)Desc->Points.Count()));

    //   (4)
    for ( int i=0; i<(int)Desc->Points.Count(); i++ )
    {
        AppendPropertyForPoint(Grid,Desc,i);
    }
}
```

In this function we first create parent property which will group properties of this set *(1),* generate arrays which will be used for chart type proeprty *(2),* generate properties for set *(3)* and add properties for points which are part of this set *(4)*. It may look little bit complicated now, but it is really easy and mostly we operate on wxPropertyGridManager class.

Now let's create function which will react on data updates to properties inside set:

```
bool wxsChart::HandleChangeInSet(wxsPropertyGridManager* Grid,wxPGId Id,int Position)
{
    //   (1)
    ChartPointsDesc* Desc = m_ChartPointsDesc[Position];
    bool Changed = false;
    bool Global = Id==Desc->Id;

    //   (2)
    if ( Global || Id == Desc->TypeId )
    {
        Desc->Type = (PointsType)Grid->GetPropertyValueAsInt(Desc->TypeId);
        Changed = true;
    }

    //   (3)
```

```
    if ( Global || Id == Desc->NameId )
    {
        Desc->Name = Grid->GetPropertyValueAsString(Desc->NameId);
        Changed = true;
    }

    //   (4)
    if ( Global || Id == Desc->PointsCountId )
    {
        int OldValue = (int)Desc->Points.Count();
        int NewValue = Grid->GetPropertyValueAsInt(Desc->PointsCountId);

        if ( NewValue<0 )
        {
            NewValue = 0;
            Grid->SetPropertyValue(Desc->PointsCountId,NewValue);
        }

        if ( NewValue > OldValue )
        {
            for ( int i=OldValue; i<NewValue; i++ )
            {
                PointDesc* NewPoint = new PointDesc;
                NewPoint->X = 0.0;
                NewPoint->Y = 0.0;
                NewPoint->Name = wxString::Format(_("Point %d"),i+1);
                Desc->Points.Add(NewPoint);
                AppendPropertyForPoint(Grid,Desc,i);
            }
        }
        else if ( NewValue < OldValue )
        {
            for ( int i=NewValue; i<OldValue; i++ )
            {
                Grid->Delete((Desc->Points[i])->Id);
                delete Desc->Points[i];
            }

            Desc->Points.RemoveAt(NewValue,OldValue-NewValue);
        }

        Changed = true;
    }

    //   (5)
    if ( !Changed || Global )
    {
        for ( int i=0; i<(int)Desc->Points.Count(); i++ )
        {
            if ( HandleChangeInPoint(Grid,Id,Desc,i,Global) )
            {
                Changed = true;
                //   (6)
                if ( !Global ) break;
            }
        }
    }

    if ( Changed )
    {
        //   (7)
        NotifyPropertyChange(true);
        return true;
    }

    return false;
}
```

That function may require some explanation. It returns true when any property in this chart points set has changed and false if not. At the beginning we extract right description *(1)* and check if id of changed property matches id of global parent property for whole set. If yes, we set Global flag to true which says that we will update all properties in set. This may be not necessary in new property grid versions but it's better to handle it this way to prevent any possible bugs and data looses. Next we test individual properties for change *(2), (3),* and monitor number of points

property in same way as we did react to changes of number of point sets *(4)*. Last thing is to forward information about property change into point data *(5)* (we do this only when there was no change so far or if we update whole set). Note here that we don't stop processing points in loop only when there's no global flag set and some point reported that it's content changed *(6)*. This have to be done this way because we have to be sure that all properties including points are updated when Global flag is set.

Those two functions introduced two more functions which operate at point level. First one is AppendProperyForPoint and the second one is HandleChangeInPoint. Implementation of these functions will be quite simple:

```
void wxsChart::AppendPropertyForPoint(wxsPropertyGridManager* Grid,ChartPointsDesc* SetDesc,int Position)
{
    PointDesc* Desc = SetDesc->Points[Position];
    wxString Name = wxString::Format(_("Point %d"),Position+1);

    Desc->Id = Grid->AppendIn(SetDesc->Id,wxParentProperty(Name,wxPG_LABEL));
    Desc->NameId = Grid->AppendIn(Desc->Id,wxStringProperty(_("Name"),wxPG_LABEL,Desc->Name));
    Desc->XId = Grid->AppendIn(Desc-
>Id,wxStringProperty(_("X"),wxPG_LABEL,wxString::Format(_T("%lf"),Desc->X)));
    Desc->YId = Grid->AppendIn(Desc-
>Id,wxStringProperty(_("Y"),wxPG_LABEL,wxString::Format(_T("%lf"),Desc->Y)));
}
```

We simply add one parent property for point and add one child property for each point's member. Second function has following implementation:

```
bool wxsChart::HandleChangeInPoint(wxsPropertyGridManager* Grid,wxPGId Id,ChartPointsDesc* SetDesc,int
Position,bool Global)
{
    PointDesc* Desc = SetDesc->Points[Position];

    bool Changed = false;
    if ( Id == Desc->Id ) Global = true;

    if ( Global || Id == Desc->NameId )
    {
        Desc->Name = Grid->GetPropertyValueAsString(Desc->NameId);
        Changed = true;
    }

    if ( Global || Id == Desc->XId )
    {
        Grid->GetPropertyValueAsString(Desc->XId).ToDouble(&Desc->X);
        Changed = true;
    }

    if ( Global || Id == Desc->YId )
    {
        Grid->GetPropertyValueAsString(Desc->YId).ToDouble(&Desc->Y);
        Changed = true;
    }

    return Changed;
}
```

We just read properties if they were changed of when global flag is set. Worth mentioning here is that the Global flag propagates from previous function which called it (`HandleChangeInSet`) and may be updated to true if parentp property of this point was changed.

Now we have full control over charts and their points. But it's not the end now. There are more four things to do: affect preview, affect source code, load from xml and

store into xml.

In this tutorial we will cover only xml operations. Affecting preview and source code will be described in next tutorial.

Updating XML structures

wxSmith uses XML strucures to store data of resource in many places. Most obvious one is XRC file, another example is WXS file and the third, hidden one is that wxSmith uses XML format to store entries in undo/redo buffers. So without routines responsible for storing and reading our custom property into xml, we don't have any of those features.

First problem we will solve is to store data into xml. Each chart point set will be stored inside <chartpointset> node and each point of this set will be stored inside <point> node which will be added as child of <chartpointset>. Some example structure may look like this:

```
    <object class="wxChartCtrl" name="ID_CHART1" variable="Chart1" member="no">
        <chartpointset name="" type="pie">
            <point name="Point 1" x="5.000000" y="6.000000" />
            <point name="Point 2" x="7.000000" y="8.000000" />
            <point name="Point 3c" x="9.000000" y="10.000000" />
        </chartpointset>
        <chartpointset name="" type="line3d">
            <point name="Point 1" x="0.000000" y="0.000000" />
            <point name="Point 2" x="10.000000" y="10.000000" />
        </chartpointset>
    </object>
```

So let's write function which will store data into xml:

```
bool wxsChart::OnXmlWrite(TiXmlElement* Element,bool IsXRC,bool IsExtra)
{
    for ( size_t i=0; i<m_ChartPointsDesc.Count(); i++ )
    {
        ChartPointsDesc* Desc = m_ChartPointsDesc[i];
        TiXmlElement* DescElem = Element->InsertEndChild(TiXmlElement("chartpointset"))
                                        ->ToElement();

        DescElem->SetAttribute("name",cbU2C(Desc->Name));
        switch ( Desc->Type )
        {
            case Bar:      DescElem->SetAttribute("type","bar");      break;
            case Bar3D:    DescElem->SetAttribute("type","bar3d");    break;
            case Pie:      DescElem->SetAttribute("type","pie");      break;
            case Pie3D:    DescElem->SetAttribute("type","pie3d");    break;
            case Points:   DescElem->SetAttribute("type","points");   break;
            case Points3D: DescElem->SetAttribute("type","points3d"); break;
            case Line:     DescElem->SetAttribute("type","line");     break;
            case Line3D:   DescElem->SetAttribute("type","line3d");   break;
            case Area:     DescElem->SetAttribute("type","area");     break;
            case Area3D:   DescElem->SetAttribute("type","area3d");   break;
        }

        for ( size_t j=0; j<Desc->Points.Count(); j++ )
        {
            PointDesc* PDesc = Desc->Points[j];
            TiXmlElement* PointElem = DescElem->InsertEndChild(TiXmlElement("point"))
                                              ->ToElement();
            PointElem->SetAttribute("name",cbU2C(PDesc->Name));
            PointElem->SetDoubleAttribute("x",PDesc->X);
            PointElem->SetDoubleAttribute("y",PDesc->Y);
        }
    }

    return wxsWidget::OnXmlWrite(Element,IsXRC,IsExtra);
}
```

In this function we iterate through all sets and store xml data for it. Note that TinyXml can not operate on wxStirng nor unicode characters directly. So we have to use `cbU2C` macro provided by Code::Blocks which will convert unicode string into standard char* (maybe it's not best solution since it should always use wxConvUTF8 because that's default encoding in xml files used by wxSmith).

Now let's write reading function. It's very simillar to writing function:

```
bool wxsChart::OnXmlRead(TiXmlElement* Element,bool IsXRC,bool IsExtra)
{
    for ( size_t i=0; i<m_ChartPointsDesc.Count(); i++ )
    {
        delete m_ChartPointsDesc[i];
    }
    m_ChartPointsDesc.Clear();

    for ( TiXmlElement* DescElem = Element->FirstChildElement("chartpointset");
          DescElem;
          DescElem = DescElem->NextSiblingElement("chartpointset") )
    {
        ChartPointsDesc* Desc = new ChartPointsDesc;
        Desc->Name    = cbC2U(DescElem->Attribute("name"));
        wxString Type = cbC2U(DescElem->Attribute("type"));

        if ( Type == _T("bar") )      Desc->Type = Bar;      else
        if ( Type == _T("bar3d") )    Desc->Type = Bar3D;    else
        if ( Type == _T("pie") )      Desc->Type = Pie;      else
        if ( Type == _T("pie3d") )    Desc->Type = Pie3D;    else
        if ( Type == _T("points") )   Desc->Type = Points;   else
        if ( Type == _T("points3d") ) Desc->Type = Points3D; else
        if ( Type == _T("line") )     Desc->Type = Line;     else
        if ( Type == _T("line3d") )   Desc->Type = Line3D;   else
        if ( Type == _T("area") )     Desc->Type = Area;     else
        if ( Type == _T("area3d") )   Desc->Type = Area3D;   else
                                      Desc->Type = Bar;

        for ( TiXmlElement* PointElem = DescElem->FirstChildElement("point");
```

```
                PointElem;
                PointElem = PointElem->NextSiblingElement("point") )
        {
            PointDesc* Point = new PointDesc;
            Point->Name = cbC2U(PointElem->Attribute("name"));
            (PointElem->QueryDoubleAttribute("x",&Point->X) == TIXML_SUCCESS) || (Point->X = 0.0);
            (PointElem->QueryDoubleAttribute("y",&Point->Y) == TIXML_SUCCESS) || (Point->Y = 0.0);

            Desc->Points.Add(Point);
        }

        m_ChartPointsDesc.Add(Desc);
    }

    return wxsWidget::OnXmlRead(Element,IsXRC,IsExtra);
}
```

In few places we used cbC2U which converts char* string into wxString.

Now if we test new chart, it stores and restores all data from xml structures (the easiest way to test it is to use undo/redo operations since they internally use xml structures). Now we can continue updating of wxChart item by generating better source code and preview. This will be covered in next tutorial.