WxSmith tutorial: Accessing items in resource

From Code::Blocks

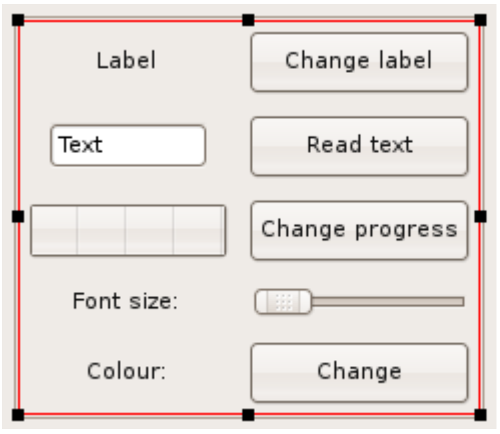
Contents

Accessing Components in a Form

Welcome to the next tutorial. This time I'll show you how to access items in a resource. I'll show you some basics - for example how to read data from text boxes, change labels, and some more advanced things like changing colours and fonts while the application is running. As usual we will start with an empty application. You shouldn't have any problems with it - all instructions are in the first tutorial.

Setting Up the Form

In this tutorial we will work on items so let's add some:



In this form, there is a wxBoxSizer, and in it a wxPanel, and on the panel, a wxFlexGridSizer with 2 columns, both growable (to achieve this enter “0,1” (without the quotes) in the Growable Cols property of the sizer, indicating columns 0 and 1). The following array shows which widgets are where in the window.

```
wxStaticText           wxButton
wxTextCtrl             wxButton
wxGauge                wxButton
wxStaticText           wxSlider
wxStaticText           wxButton
```

All widgets and the wxPanel behind them have their Expand property checked.

Here is what we want to make the buttons do:

- The button in the first row will change the text in the static text left of it.
- The button in the second row will read whatever text the user has written into the TextCtrl left of it and display that text in a message box.
- The button in the third row will advance the gauge to the left of it by one tenth of the distance across the gauge, thus simulating a typical progress bar.
- Moving the slider in the fourth row will change the size of the font in the static text just left of it.
- A click on the button in the bottom row will allow the user to change the color of the text in the StaticText control just left of it.

Accessing Components through their Member Functions

If you add an item to the form in the editor, in most cases wxSmith will add a new member variable into the form's C++ class. All those members are listed in the header file, in our present case, the file *Tutorial_6Main.h*. They are between wxSmith's special comments: *//(*Declarations...* and *//*)*. The list should look like this:

```
...
class Tutorial_6Frame: public wxFrame
{
    ...

        //(*Declarations(Tutorial_6Frame)
        wxSlider* Slider1;
        wxButton* Button4;
        wxStaticText* StaticText2;
        wxButton* Button1;
        wxGauge* Gauge1;
        wxStaticText* StaticText1;
        wxStaticText* StaticText3;
        wxButton* Button2;
        wxButton* Button3;
        wxStatusBar* StatusBar1;
        wxTextCtrl* TextCtrl1;
        //*)
    ...
};
...
```

Each component which has such a member variable will also have the following properties:

- **Var name** - name of the used variable
- **Is member** - switch whether this item should be accessible through a class member variable

So if you want to change the name used to access the item, **Var name** is the right property to change.

wxSmith forces a few restrictions on variable names. The most obvious is that each variable name must be a valid C++ identifier, so you cannot use special characters or even spaces. Another limitation is that variable names must be unique.

If the name is invalid, wxSmith will automatically replace it with a name that matches the criteria.

Changing the Label in wxStaticText

Let's do a basic exercise. When we click the “Change label” button, we want the program to change the label of the first wxStaticText control to “Label changed”. To make it do so, double click on the button to generate an event handler and change the code to the following:

```
void Tutorial_6Frame::OnButton1Click(wxCommandEvent& event)
{
    StaticText1->SetLabel(_("Label changed"));
    Layout();
}
```

In the first line of the function body, we used the **SetLabel()** function to change the text of the label. Because the length of the text changes, we must recalculate positions, which is done by calling the **Layout()** function.

Static items (those which can't be changed by the user) usually have two functions: GetLabel and SetLabel which read and write content presented by this item.

You may wonder why we used some weird notation for the string:

```
_("Label changed")
```

instead of a simple

```
"Label changed"
```

There are two advantages of this device:

- By adding _(...) around our string we prepare our application for the translation process. wxWidgets can help in developing multi-language applications. When a translation is being made, wxWidgets will automatically search for the translation of the phrase "Label changed" in a database of equivalent strings in the source and target languages.

- wxWidgets may be provided in two versions: with Unicode support and without it (ANSI build). In the case of a Unicode version, we would have to use Unicode strings: **L"Label changed"** and in case of the ANSI version we would have to use standard string notation: **"Label changed"**. Using the **_(...)** macro insures that no matter what wxWidgets version is used, it will always produce the proper code.

There's an alternative version of the string macro which works similarly to _(...) which is written in the form _T(...) or wxT(...). It works like the _() one but the string is never translated.

Reading the Value from wxTextCtrl

Now let's read something that's written by the user. We will use the wxTextCtrl (with variable TextCtrl1) for this and show the text using a standard message box. Let's double click the **Read text** button and change the code to the following:

```
void Tutorial_6Frame::OnButton2Click(wxCommandEvent& event)
{
    wxString Text = TextCtrl1->GetValue();
    wxMessageBox(_("User entered text:\n") + Text);
}
```

In the first line, we read the value from the TextCtrl and store it in a variable called Text and of the type wxString. wxString is the wxWidgets implementation of a string class and this library uses it as a base for string representation.

In the second line, we call the **wxMessageBox** function which shows a standard message box just as if it were a modal dialog.

Usually items which provide content entered by the user have two member functions: GetValue and SetValue. You can use them to read and write the content of such items.

Changing the value of wxGauge

Now let's combine reading and writing of a component's value in one function. We will use wxGauge for this. We will use it's two member functions: GetValue and SetValue. I've written earlier that those functions usually exist in items where the user can enter some value. Well, that rule is not without exceptions as we will now see as we write the handler for the third button. Double click the button and fill in the handler as follows:

```
void Tutorial_6Frame::OnButton3Click(wxCommandEvent& event)
{
    int NewValue = Gauge1->GetValue()+10;
    if ( NewValue > 100 ) NewValue = 0;
    Gauge1->SetValue(NewValue);
}
```

In the first line, we generate a new progress value by reading the current one and adding 10 to it. In the second line, we prevent the value from getting out of range. (By default, wxGauge has a range set to 0...100. These values can be changed in the Properties browser.) In the third line, we write the new value into the gauge.

Using the Value from wxSlider to Change Font Size

Now let's do something more advanced. In our resource there is a wxSlider and text saying that it changes font size. We will change the font of this label. But we will have to update the font size in two steps:

- As long as the user drags the slider we should change the font size only
- When the user finishes dragging we should layout the window because the size

of the text changes

wxSlider provides two events that can be used precisely for this purpose. The first is **EVT_COMMAND_SCROLL_THUMB_TRACK** which is fired while dragging the slider. The second is **EVT_COMMAND_SCROLL_THUMB_RELEASE** which is fired when the user finishes dragging.

Because we are using events which are not default for the slider, double clicking on the slider will not work -- as it did with buttons -- to create a frame for the handler. We have to add the two handlers through the Properties browser. You can switch between editing standard properties and editing events by clicking on buttons at the top of browser:

.

When you have switched to events, search for the required events and choose **Add new handler** from the drop-down list.

Here's the code for the **EVT_COMMAND_SCROLL_THUMB_TRACK** handler:

```
void Tutorial_6Frame::OnSlider1CmdScrollThumbTrack(wxScrollEvent& event)
{
    wxFont Font = StaticText2->GetFont();
    Font.SetPointSize( Slider1->GetValue() );
    StaticText2->SetFont(Font);
}
```

Here we get the font used by the StaticText control by using the **GetFont()** function, change the font's size by using **SetPointSize()** and write the font back by using the **SetFont()** function. **GetFont()** and **SetFont()** are available for most components.

Now let's code **EVT_COMMAND_SCROLL_THUMB_RELEASE**:

```
void Tutorial_6Frame::OnSlider1CmdScrollThumbRelease(wxScrollEvent& event)
{
    Layout();
    GetSizer()->SetSizeHints(this);
}
```

In the first line, we adapt the positions of items to the new font size just as in the case of changing the label in wxStaticText. In the second line, we recalculate the minimal size of the window making sure that all items will have enough space.

(*Use Panel1->GetSizer()->SetSizeHints(this);for the second line in the above statement, where Panel1 is the name of the base panel in your frame.The original code, rather than resize to the new required size, just resizes to the original, default size.Ensnarer 03.08.2015*)

Changing a Component's Color

The last thing we will do in this tutorial is to change the color of some item. As in the case of fonts we will change the label on the left side of the **Change** button. Since we will use the standard event of this button, we can add an event handler by double-clicking on it. And here's the code:

```
void Tutorial_6Frame::OnButton4Click(wxCommandEvent& event)
{
    wxColour OldColour = StaticText3->GetForegroundColour();
    wxColour NewColour = wxGetColourFromUser(this,OldColour);

    if ( NewColour.IsOk() )
    {
        StaticText3->SetForegroundColour(NewColour);
        Refresh();
    }
}
```

(wxWidets originated in Edinburgh and therefore uses some quaint spellings going back to Anglo-Norman times.) At the beginning we read the current color into the variable **OldColour**. In the next line, we call the **wxGetColourFromUser** function which opens a dialog where the user can choose a colour.

At the end, we set the new colour by using the **SetForegroundColour** function. Refresh is needed to change colour after setting it.

If, however, you try to compile this code, you will get an error message telling you that wxGetColourFromUser() is an unknown function. Since we put it in manually, we must also put in manually its header. So open the Tutorial_6Main.h file and add #include <wx/colordlg.h>.

The NewColour.IsOk() call is used because if the user cancels the colour selection it will return false. But on the Ubuntu Linux 11.10 installation of wxWidgets, the wxColour class did not have an IsOk() funtion, so in fact you must remove this call. According to wxWidgets documentation, the wxColour class **does** indeed have the IsOk() member function, but it was not found.

(FIXING: For get the code in the previous handler worked, just replace that on to:

```
wxColour OldColour = StaticText3->GetForegroundColour();

wxColourData dc;

dc.SetColour(OldColour);

wxColourDialog clrdlg(this,&dc);

if ( clrdlg.ShowModal() == wxID_OK )

{

   StaticText3->SetForegroundColour(clrdlg.GetColourData().GetColour());

   Refresh();

}
```

)

More information

We've reached the end of this tutorial. I showed only a few operations on typical components; there is much more you can do. For more details you can check wxWidgets' documentation available here (http://www.wxwidgets.org/manuals/stable/wx_contents.html).

Previous | Index | Next