Creating a plugin which adds new item into wxSmith

From Code::Blocks

Contents

What you should know before reading this tutorial

This tutorial shows you how to create a new plugin that extends the functionality of the wxSmith plugin. Before you start you should have wxWidgets and Code::Blocks compiled from source code. I also recommend that you read Creating a simple "Hello World" plugin, although it's not necessary, it will familiarize you with the Code::Blocks plugin architecture and the creation of new plugins. I also assume that you know at least basics of C++ ;).

In this tutorial we will add a wxChart item into wxSmith. We won't cover all aspects of the wxChart class here, but this tutorial may be a good starting point for creating more powerfull and sophisticated extensions :) It demonstrates only the basics of wxSmith, so I hope to write a few more tutorials covering other parts of this system in the future.

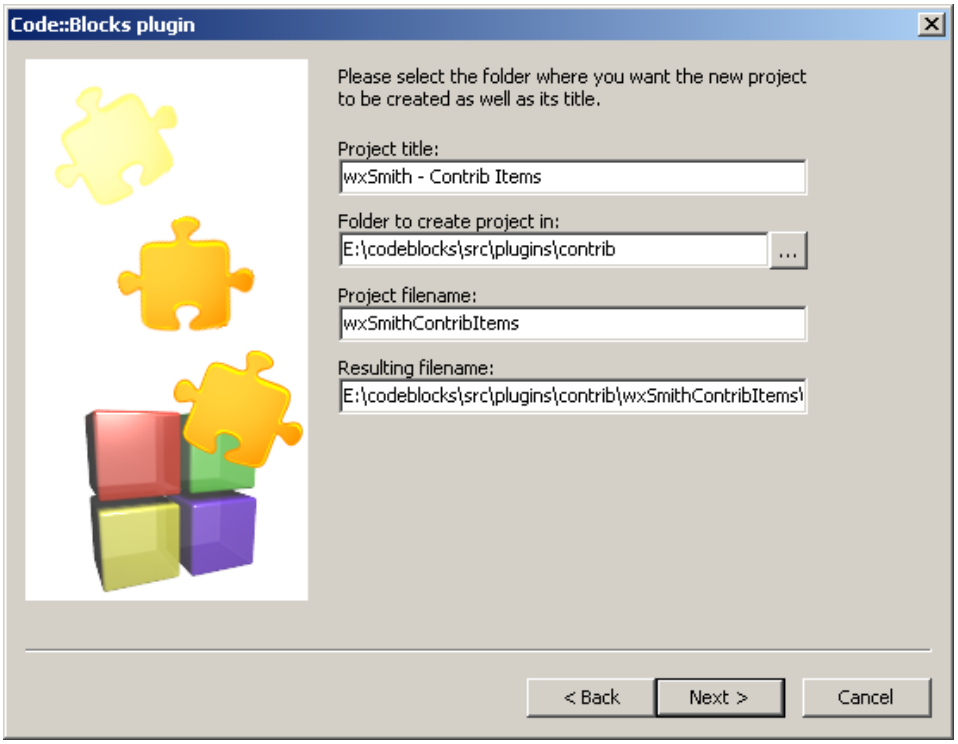
This tutorial was created in a Windows XP environment but it should be quite easy to compile and run it on Linux (at least by using Code::Blocks).

Before we begin I need to make one more disclaimer: the internal structure of wxSmith may change. I'll try to stay as close to current interface as possible and I will try to not write about aspects of wxSmith that I'm planning to change (or I'll at least notify you that it may be outdated soon), but I cannot guarantee that all your work will compile after a year (or maybe even few months). I'll try to keep these tutorials updated so you'll be able to get back here and find changes.

Creating new plugin

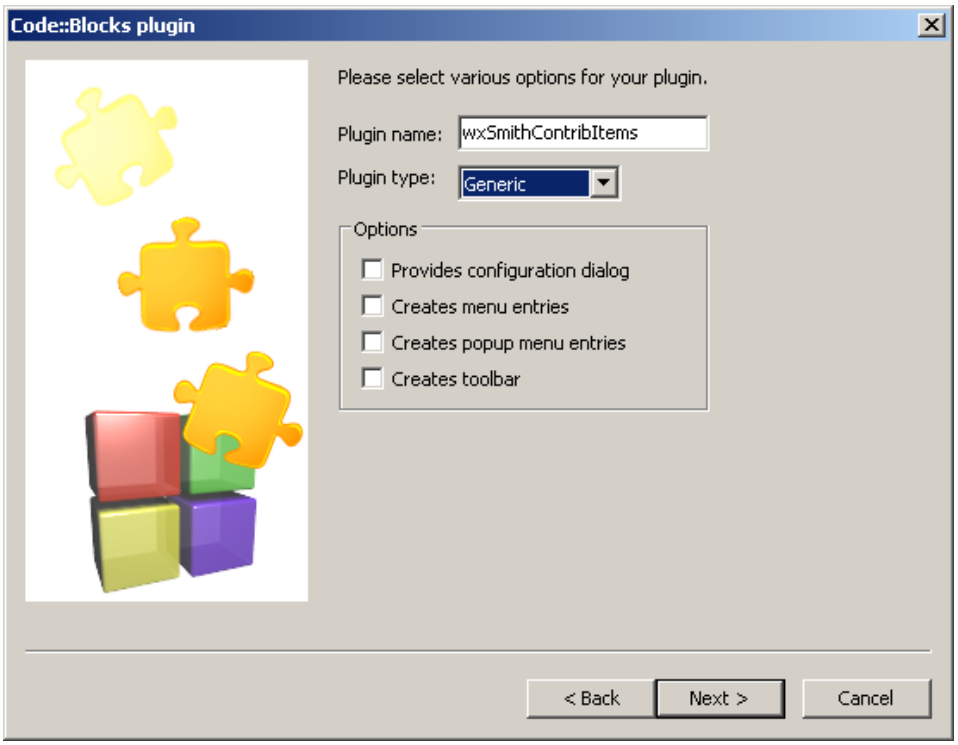
Let's start with new plugin. Most of informations here will be simillar to Creating a simple "Hello World" plugin, but there may be some small differences :)

We start with new Code::Blocks plugin wizard. It's available in File->New->Project menu. We select Code::Blocks plugin there and start with following window:



I've named my plugin *wxSmith - Contrib Items* and placed it into contrib plugins folder. Choosing this folder probably wasn't a good idea. Any other folder, not related to Code::Blocks will work the same, and that should be your choice ;)

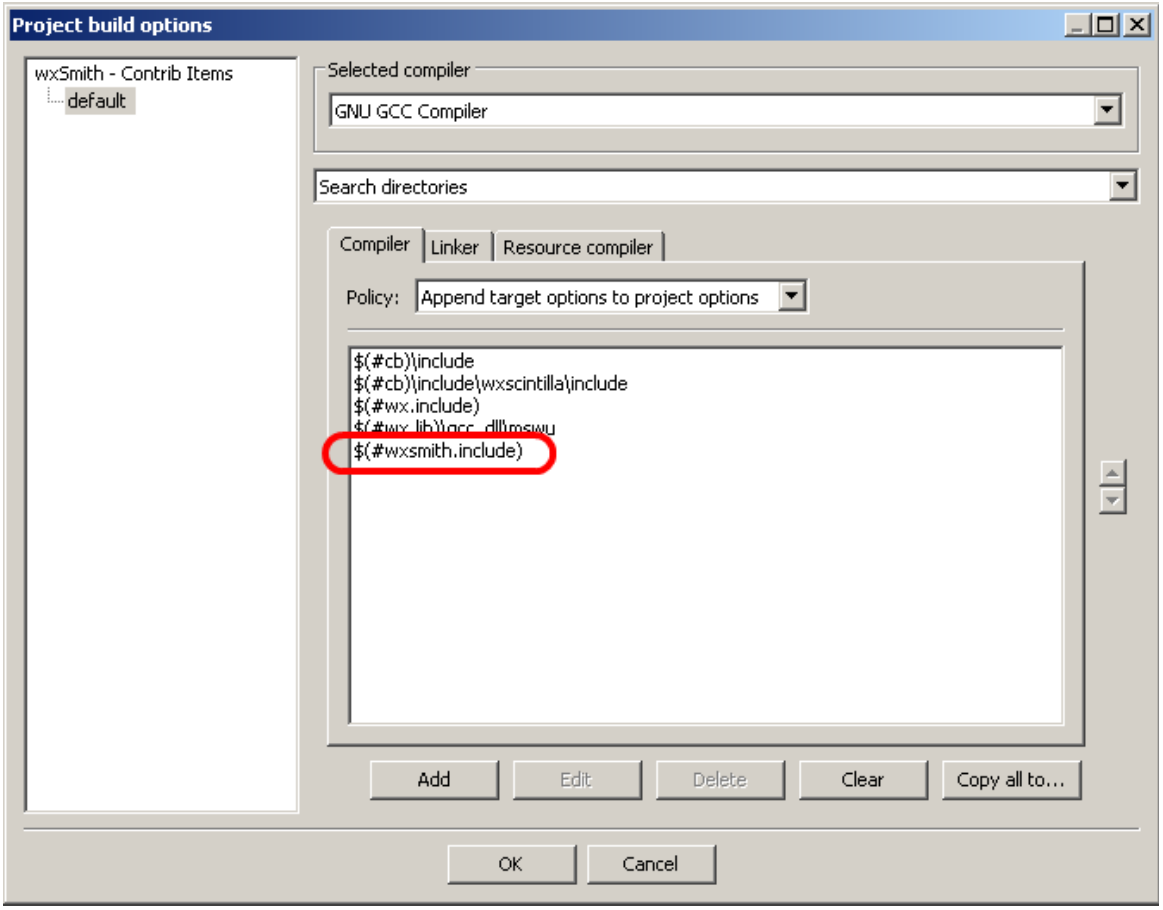
On the next wizard page we choose plugin type.



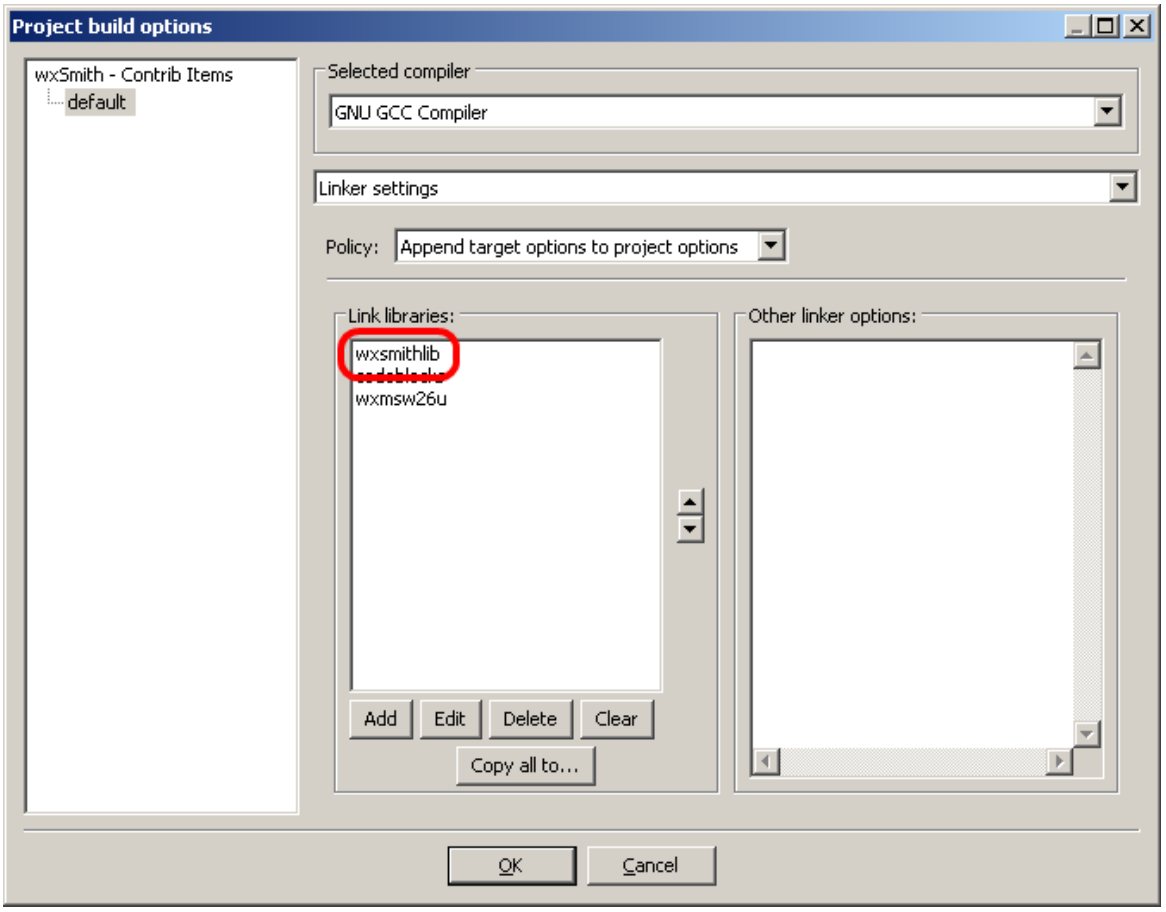
Because this plugin doesn't match any particular plugin type, we use Generic. In fact, our plugin won't behave as other plugins do. It will use totally different system for extending wxSmith's functionality. But even though, we have to create it using some scheme to let Code::Blocks recognize and load it.

We follow the instructions in wizard and end up with new plugin.

We should now be able to compile out project, but let's add binding to wxSmith first. We need to add wxSmith's directory into include locations list:

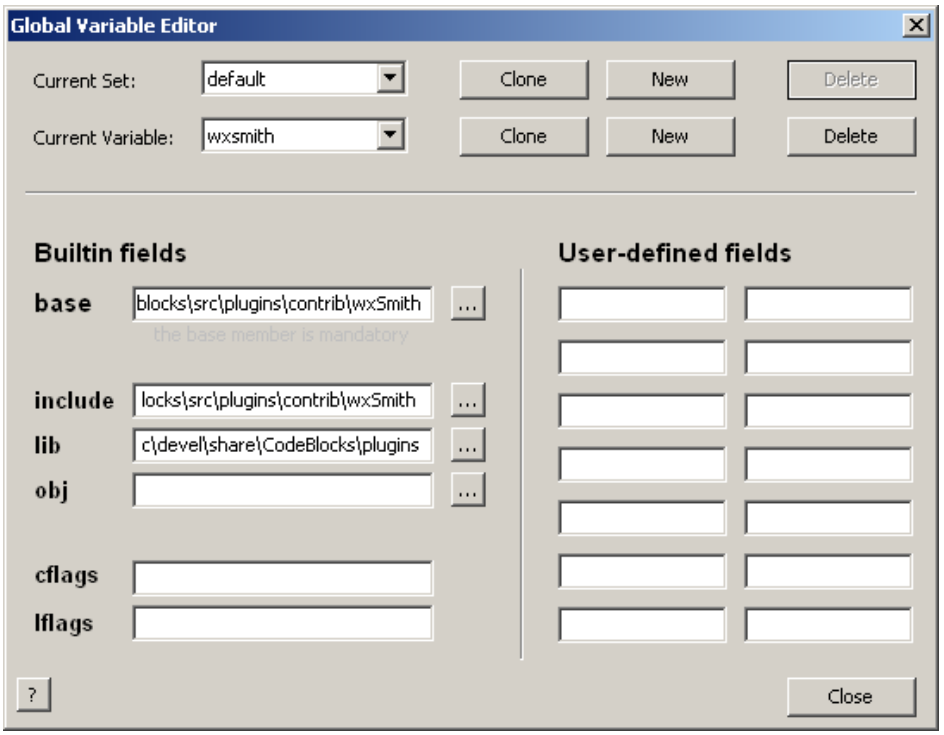


Last thing we need to add to bind our plugin into wxSmith is wxSmith's library:



Make sure it's on the top of list since it looks like MingW does care about the order in which libraries are linked.

Now we close project's options and try to compile. Code::Blocks will ask for settings for wxSmith. We have to fill base path which is required, but also include and lib directories since they are not usual:



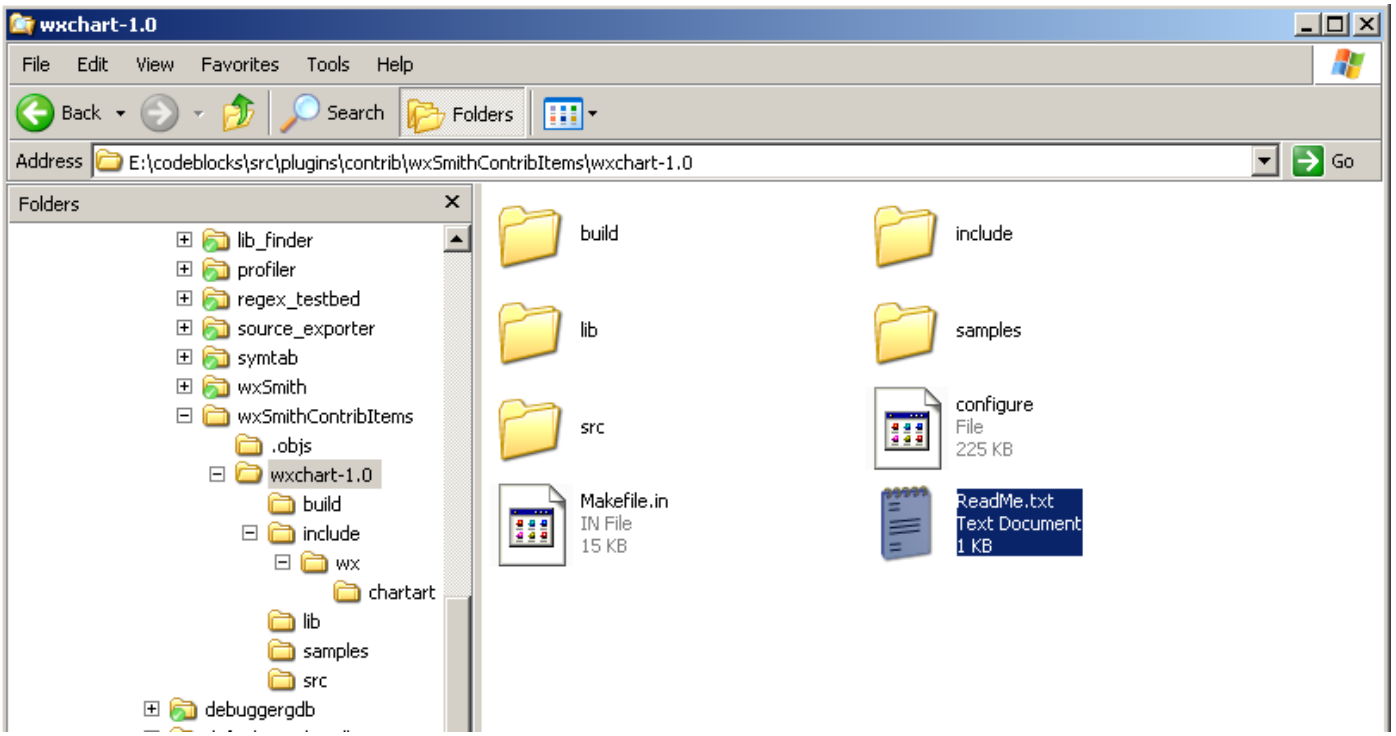
Note that include dir points to wxSmith's directory. There's no need to specify custom lib directory

because it's now located in code::blocks's root path.

If everything compiles fine, we can move to next step

Compiling-in contrib widget

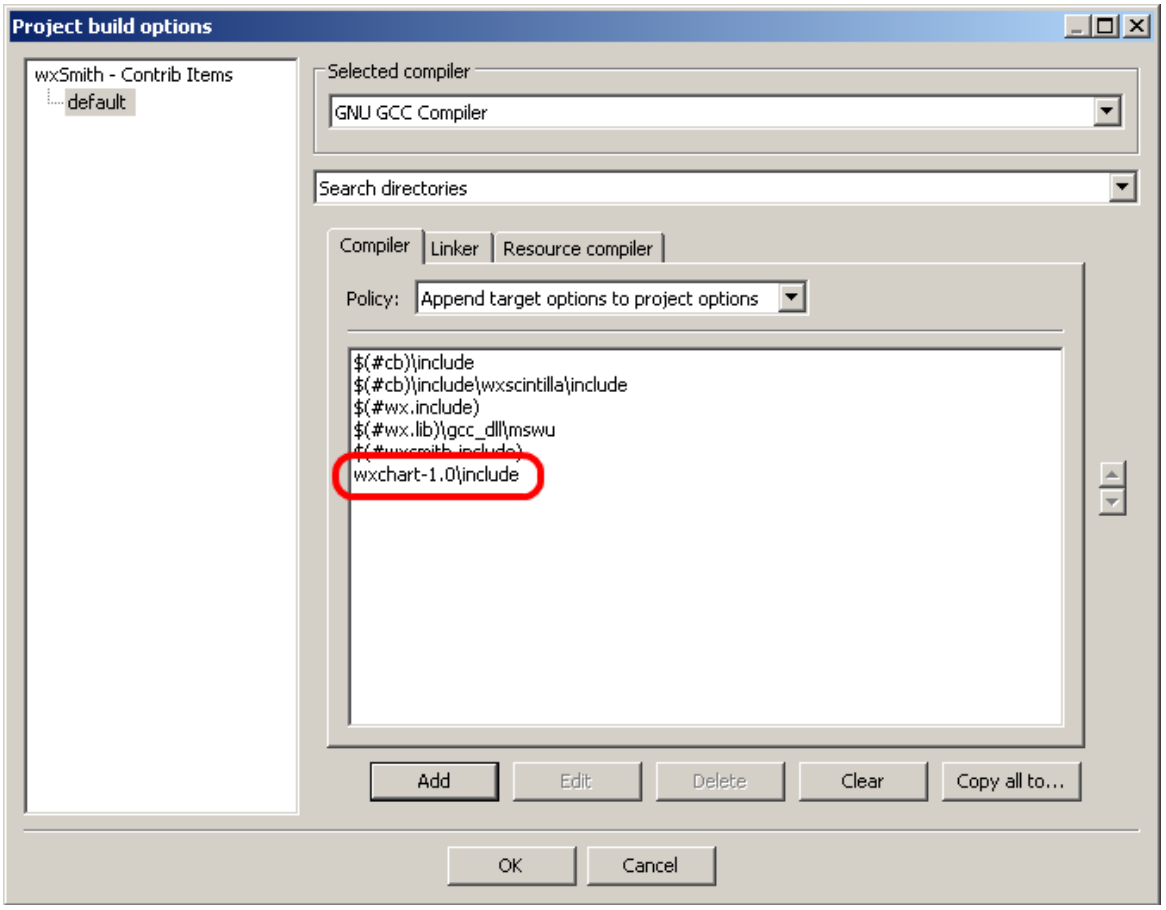
Now it's time to add widget we want to add into wxSmith right into our plugin. I have choosen wxChart hosted at wxCode (http://wxcode.sourceforge.net/components/wxchart/), mostly because it will be easy to add and it looks promising :) First thing we need is source code. It's freely available at sourceforge (the link above). After unpacking it into our plugin's folder the directory structure should look like this:



I'd like to compile this source just as a part of plugin. Compiling it as external dll may lead to some problems because plugin would consist of many files. The easiest way is just to add wxChart's files into plugin's project.

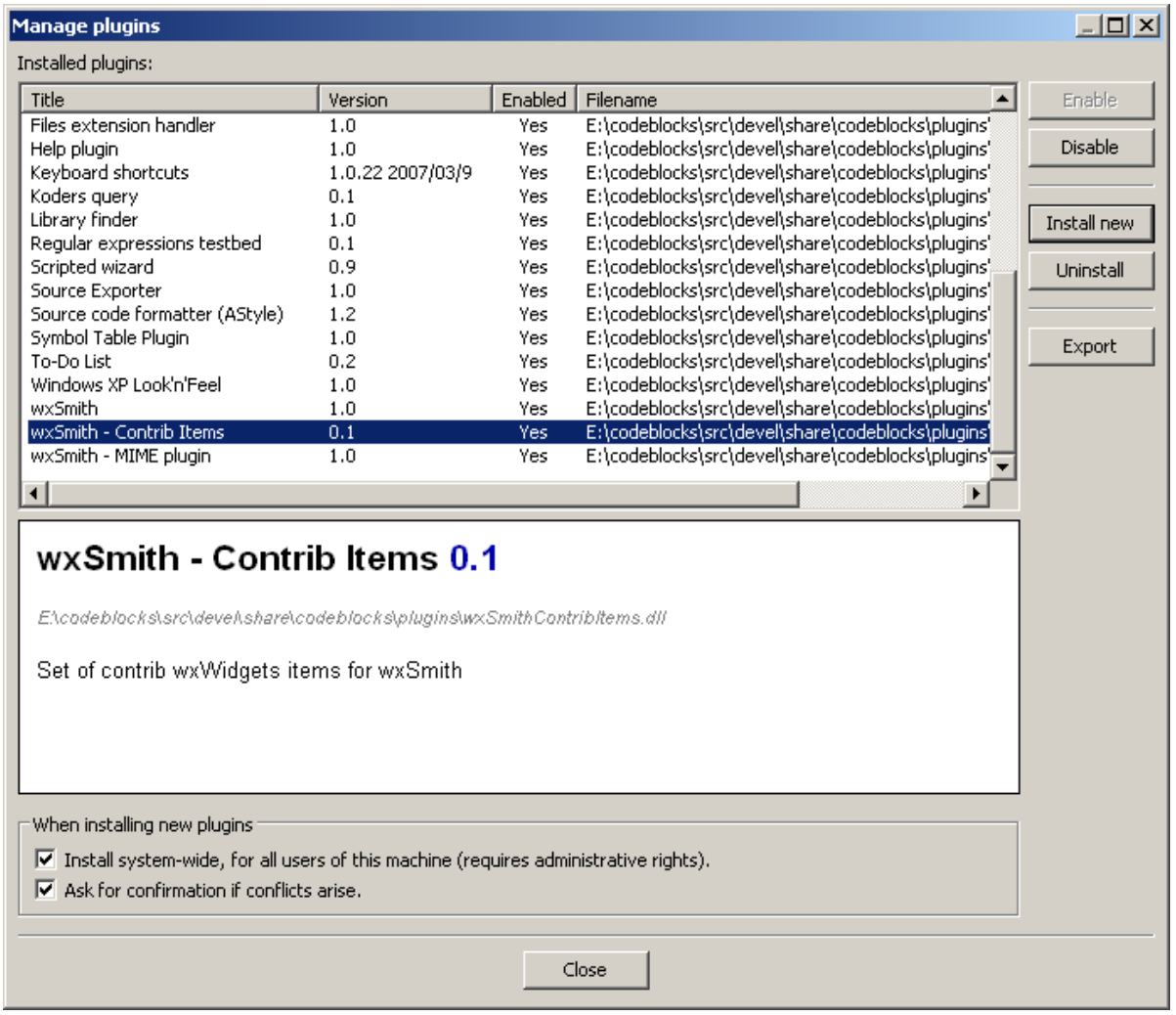
Files we need are placed in wxchart-1.0\include\wx and wxchart-1.0\src (watch out for file in samples folder since it will try to create whole application). We just add the content of these folders into project.

There's one more thing left to make wxChart compile with wxSmith. We have to add it's include dir into search paths:



After adding that, project should compile fine with wxChart with it.

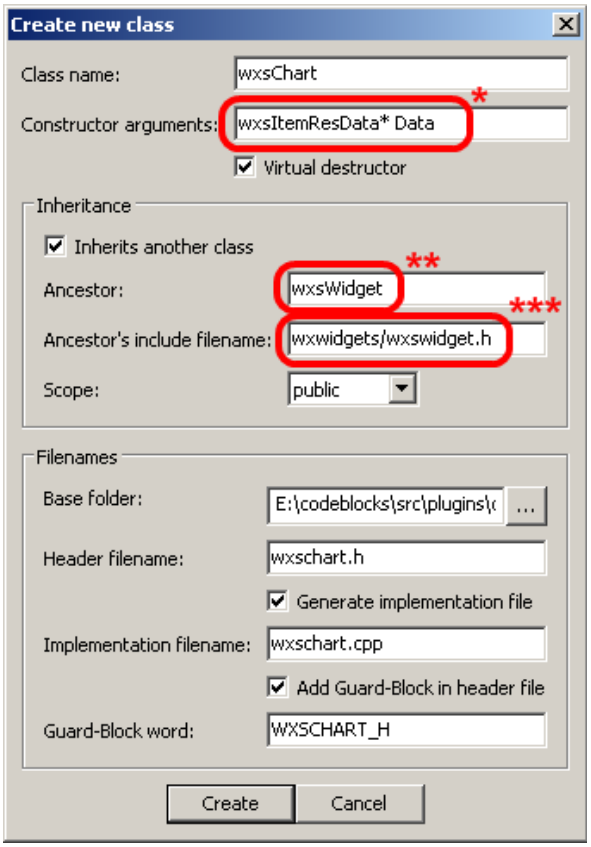
At this point I wanted to test whether our plugin still works. To do that I've installed it in plugin manager (remember that you must install it in Code::Block you have compiled yourself, otherwise there may be some version problems). I've got the proove that it all went OK:



Enabling new item inside wxSmith

Adding suporting class

Now it's time to enable our item inside wxSmith. To do this first thing we have to do is to create new class that will support our wxChart. The name of class should be unique to prevent conflicts with other classes. Convention used in wxSmith is to replace wx in class names with wxs. So we create wxsChart class:



Note three extra settings which have to be added: consructor arguments (*), Ancestor(**) and Ancestor's include filename (***). Filling up these three fields make things much easier.

Item's images

Another thing we need for out item are icons. wxSmith require two icons for each item. First one should be 32x32 pixels (it's used for bigger version of palette and inside tools pane), second one 16x16 (used in resource tree and default small palette). I've created them from wxChart's screenshots using Paint.NET and converted them to XPM using GIMP. You may wonder why I want XPM files. The reason I've choosen that is that XPM files can be linked directly into source code with one single #include and they're directly supported inside wxBitmap and wxImage constructors. When we link images into dll, we won't have to take care about some extra image files required for plugin to work.

Creating global information objects

The only thing needed to register item inside wxSmith is to create one variable from template called *wxsRegisterItem*. Created object will automatically register and unregister item from wxSmith and will provide basic informations about item before it's created. Because it does require bitmaps, we will load xpm data first. After instantiating wxsRegisterItem template we also define set of used styles, but that will be described later.

```
namespace
{
    // Loading images from xpm files
    #include "images/wxchart16.xpm"
    #include "images/wxchart32.xpm"

    // This code provides basic informations about item and register
    // it inside wxSmith
    wxsRegisterItem<wxsChart> Reg(
        _T("wxChartCtrl"),                  // Class name
        wxsTWidget,                         // Item type
        _T("wxWindows"),                    // License
```

```
        _T("Paolo Gava"),                       // Author
        _T("paolo_gava@NOSPAM!@hotmail.com"),  // Author's email (in real plugin there's no need to do anti-spam tricks ;)
)
        _T("http://wxcode.sourceforge.net/components/wxchart/"),    // Item's homepage
        _T("Contrib"),                          // Category in palette
        80,                                     // Priority in palette
        _T("Chart"),                            // Base part of names for new items
        wxsCPP,                                 // List of coding languages supported by this item
        1, 0,                                   // Version
        wxBitmap(wxchart32_xpm),                // 32x32 bitmap
        wxBitmap(wxchart16_xpm),                // 16x16 bitmap
        false);                                 // We do not allow this item inside XRC files

    // Defining styles
    WXS_ST_BEGIN(wxsChartStyles,_T("wxSIMPLE_BORDER"))
        WXS_ST_DEFAULTS()
    WXS_ST_END()
}
```

The important thing here is that we put all this information inside a nameless namespace. This will prevent redeclaration errors since usually all wxsRegisterItem instatiations produce a variabe called *Reg*.

The arguments passed to wxsRegisterItem are easy to understand. Not all of them are used now, many are given just as information which may be used in future (f.ex. license may be used to check whether we can use item freely or not). Essential data fields which are used by wxSmith now are:

- Class name – name of item's class, will be used while generating declaration of item's variable and for new operator, it's also identifier for item type, so there should not be two items with same name (only one of them will be available)
- Item type - type of item, for widgets like wxChart (they don't have children), it should be *wxsTWidget*. Type of item should match base class of class created to support this item. Other options are: wxsTContiner (f.ex. wxPanel), wxsTSizer, wxsTSpacer (only one item of this type is present) and wxsTool (f.ex. wxTimer). This tutorial concentrates on wxsTWidget classes so information presented here may be not enough to create other item types.
- Category in palette - giving empty category will prevent the item from being displayed, this name should not be translated in info (it must use _T() instead of _()), it will be translated later while generating palette
- Priority - Items with high values are placed on the left side of item palette and are easily accesible because of that
- Base part for variable names - it's used to generate variable name and identifier for items which don't have one or have invalid values
- Languages - set of languages supported by item, currently only wxsCPP is supported
- bitmaps - used in palette and resource browser
- XRC switch - if this value is true, this item is allowed in resources using XRC files. If it's false, item is not supported by XRC (which is usual case for contrib items). This value may be replaced by flags in future to allow supporting more characteristics of item. After switching to flags, code should still compile becuase XRC flag will have value 1 (value of true when it's converted to integer).

After registration object, we define item's styles. wxChart makes here two style sets: it's own made as enum STYLE (you should be carefull about that because it is really common name and it's defined globally in <chartctrl.h>) and flags (styles of window). When we talk about styles in wxSmith we always mean style argument as described in wxWidgets documentation (so here it will be flags). To help generating style set, wxSmith gives set of macros. Definition of set begins with WXS_ST_BEGIN. First argument is name of set, second is default style. WXS_ST_DEFAULTS() adds all default styles listed in wxWindow class help (not all may be used by particular item). Adding user-defined styles is done through WXS_ST(<style>) macro, but it wont be described with details in this tutorial. Definition of styles end with WXS_ST_END().

In this implementation we will discard internal wxChart's styles setting them always to DEFAULT_STYLE.

Constructor

Now we have all informations about item, so we can create our class. We had to declare these informations before supporting class' constructor because constructor require them. Here's some code from wxsChart:

```
wxsChart::wxsChart(wxsItemResData* Data):
    wxsWidget(
        Data,               // Data passed to constructor
        &Reg.Info,          // Info taken from Registering object previously created
        NULL,               // Structure describing events, we have no events for wxChart
        wxsChartStyles)     // Structure describing styles
{
}
```

Required functions

And that's all for constructor. When we try to compile it now, compiler complains about pure virtual functions. So we have to implement them. List of functions is as follows:

- Generating source code for this item

```
void OnBuildCreatingCode();
```

- Building preview of this item (either for editor or for full window preview)

```
wxObject* OnBuildPreview(wxWindow* Parent,long Flags);
```

- Enumerating extra properties of this widget

```
void OnEnumWidgetProperties(long Flags);
```

Let's take a look at implementations of these functions. First code function generating source code of item;

```
void wxsChart::OnBuildCreatingCode()
{
    switch ( GetLanguage() )
    {
        case wxsCPP:
            AddHeader(_T("<wx/chartctrl.h>"),GetInfo().ClassName);
            Codef(_T("%C(%W,%I,DEFAULT_STYLE,%P,%S,%T);\n"));
            break;

        default:
            wxsCodeMarks::Unknown(_T("wxsChart::OnBuildCreatingCode"),GetLanguage());
    }
}
```

Firs thing to mention is that this function may support multiple languages. Currently there's only CPP supported, but that will be expanded in future. The structure

```
   switch ( GetLanguage() )
   {
       case wxsCPP:
           ... Add code here ...
           break;

       default:
           wxsCodeMarks::Unknown(_T("<Function name>"),GetLanguage());
   }
```

is common for all functions generating source code. The *wxsCodeMarks::Unknown* call is not obligatory, but may ease finding unfinished parts of source code when new languages will be added.

Code generation uses Codef() function which is a helper when generating source code. It works simillar to Printf function but has different formating characters in most cases and support for standard format characters (like %s or %d) is very simplified. In the code above following formating characters were used:

- %C – creation prefix, it usually expands to <VARIABLE> = new <CLASS>, but may also be <VARIABLE>.Create (for instances instead of pointers, this must be used carefully since not

all classes provide Crate function simillar to constructor) or simply Create (when this is root item – f.ex. wxDialog and we're initializing the class itself)
- %W – pointer to parent **W**indow
- %I – current **I**dentifier
- %P – item's **P**osition
- %S – item's **S**ize
- %T – item's s**T**yle

Additionally in function generating source code we list headers required by this resource. It's made by call:

```
AddHeader(_T("<wx/chartctrl.h>"),GetInfo().ClassName);
```

where first argument is name of include file and the second one is name of class declared in this header. Adding both include name and class name may be used to generate forward declarations instead of including header which may speed-up compilation of generated files. This function can also be used to list headers (and classes) used only for resource generation purposes (like wxFont which is not required in class declaration). To add such header use following convention:

```
AddHeader(_T("<wx/barchartpoints.h>"),_T(""),hfLocal);
```

If you compare it with previous AddHeader call you may notice additional argument where on can post header flags. hfLocal means that this include (and class) is used only locally while generating resource.

Next comes function generating preview – it should be similar to function generating code but the output should be object.

```
wxObject* wxsChart::OnBuildPreview(wxWindow* Parent,long Flags)
{
    return new wxChartCtrl(Parent,GetId(),DEFAULT_STYLE,Pos(Parent),Size(Parent),Style());
}
```

The important thing is Flags argument, currently it contain one flag: pfExact which say whether this is preview generated for editor or real preview.

To check when preview is built to be used in preview, use:

```
if ( !(Flags&pfExact) ) { ... do some code ... }
```

And analogically to check when it will be used in preview window, use:

```
if ( Flags&pfExact ) { ... do some code ... }
```

This may be used to optimize the performance for some items which require time consuming initialization (f.ex. WxHtmlWindow which may request to load some web page).

Third function is coded as follows:

```
void wxsChart::OnEnumWidgetProperties(long Flags)
{
}
```

It's empty because we do not provide our custom properties now. Properties will be explained in other tutorial.

Now it's time to compile out plugin. We only need to add *#include <wx/chartctrl.h>* to include chart control and voila, compiles fine. So, let's test it now in real C::B environment.

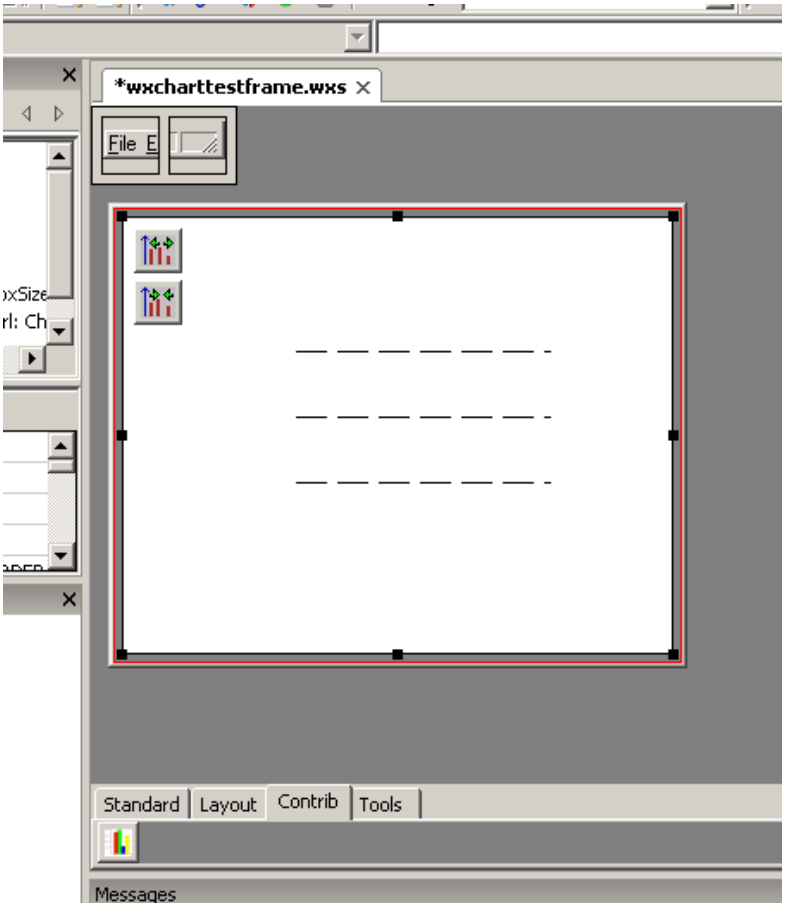
Testing of new plugin

New control is on the palette, it can be added but it shows some error messages about missing bitmaps in .rc file. First I tried to add them into some local rc file, but it didn't work (AFAIK Windows tries to look for resources inside exe file only). So I've done it in some rather dirty matter: I've forced to use XPM files instead of windows resources. It required following changes in chartctrl.cpp:

```
//#if !defined(__WXMSW__) && !defined(__WXPM__)
    #include "wx/chartart/chart_zin.xpm"
    #include "wx/chartart/chart_zot.xpm"
//#endif

...

//        #if defined(__WXMSW__) || defined(__WXPM__)
//            return wxBitmap(wxT("chart_zin_bmp"), wxBITMAP_TYPE_RESOURCE);
//        #else
            return wxBitmap( chart_zin_xpm );
//        #endif

...

//        #if defined(__WXMSW__) || defined(__WXPM__)
//            return wxBitmap(wxT("chart_zot_bmp"), wxBITMAP_TYPE_RESOURCE);
//        #else
            return wxBitmap( chart_zot_xpm );
//        #endif
```

(lines in bold were commented out).

After recompiling we got new item working without problems:



Some tips

In this example we have created nameless namespace to hide structures containing informatino about item. To fully avoid the risk of doubled symbols, whole class could be put inside such nested namespace. That can be done because this class is not accessed outside anywhere directly as wxsChart. Whole wxSmith see it as wxsWidget. Drawback of this solution is that debugger may not

see content of nameless namespaces.