WxSmith tutorial: Hello world

From Code::Blocks

Contents

Tutorial 1. "Hello, World!"

In this first tutorial, we will use wxSmith to put the words "Hello, World!" on the screen. Ever since the appearance of *The C Programming Language* in 1978, writing such a program has been the traditional first step in learning nearly every computer language.

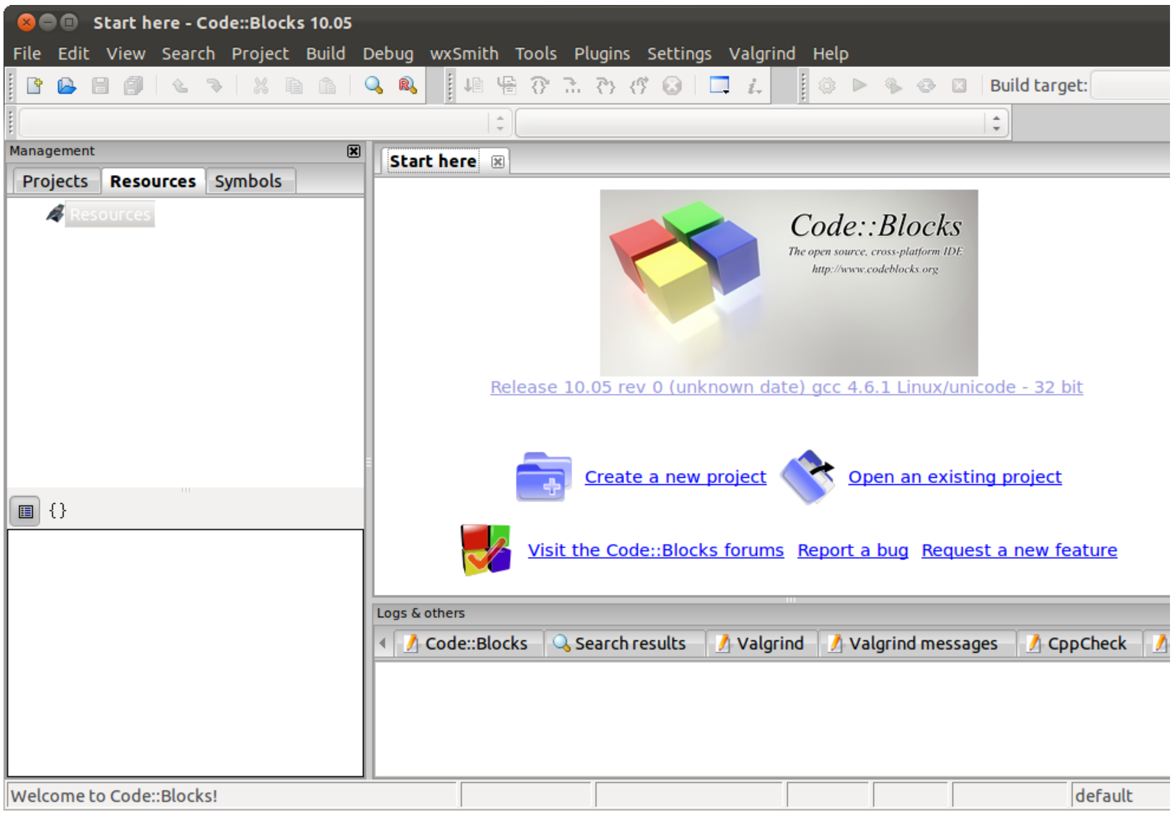
Before we start, you must either compile wxWidgets or download precompiled binaries and header files. These steps are described on the wiki pages mentioned here:

- Windows users should see the wiki page:Compiling wxWidgets 2.8.6 to develop Code::Blocks (MSW) or Installing Code::Blocks from source on Windows or WxWindowsQuickRef. There are two ways to install the wxWidgets libraries, You can download the wxWidgets source and compile it yourself, or you can download the wxPack from [1] (http://wxpack.sourceforge.net/) and install it. The wxPack contains a pre-compiled wxWidgets library, so you can save a lot of time by that route.

- Ubuntu Linux users can install wxWidgets from the Ubuntu repositories. Be sure to install also **build-essential** to get the C++ compiler.

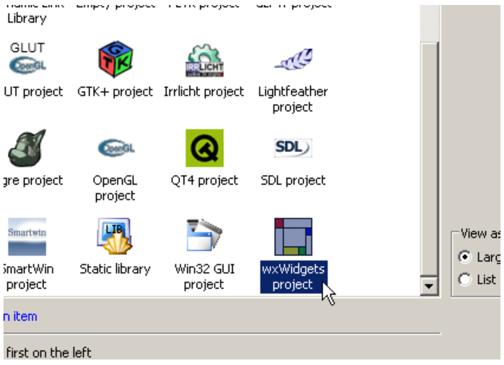
We will assume that you have wxWidgets ready and working with Code::Blocks. You should probably create a directory with a name something like CBProjects for saving your work on these tutorials. Each tutorial will be a separate "project" and will occupy a file in this directory.

After downloading and installing Code::Blocks, double-click its icon on the desktop to start it. Here is the Code::Blocks opening window:



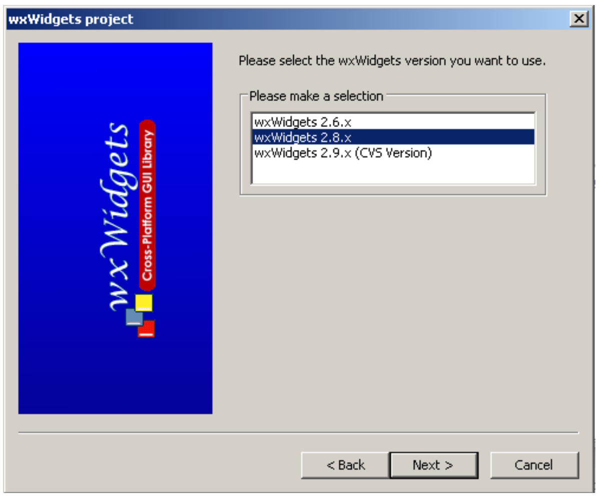
Start here - Code::Blocks 10.05
File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Valgrind Help
Build target:
Management
Projects Resources Symbols
Resources
Start here
Code::Blocks
The open source, cross-platform IDE
http://www.codeblocks.org
Release 10.05 rev 0 (unknown date) gcc 4.6.1 Linux/unicode - 32 bit
Create a new project
Open an existing project
Visit the Code::Blocks forums
Report a bug
Request a new feature
Logs & others
Code::Blocks
Search results
Valgrind
Valgrind messages
CppCheck
Welcome to Code::Blocks!
default

Click "Create a New Project" and then find and double-click the wxWidgets Project icon, pointed to by the cursor in the screenshot below.



The next screen is a "Welcome" screen. Unless you just enjoy being welcomed by the program, you will check the box which eliminates it on future startups.

The next screen looks like this:


wxWidgets project

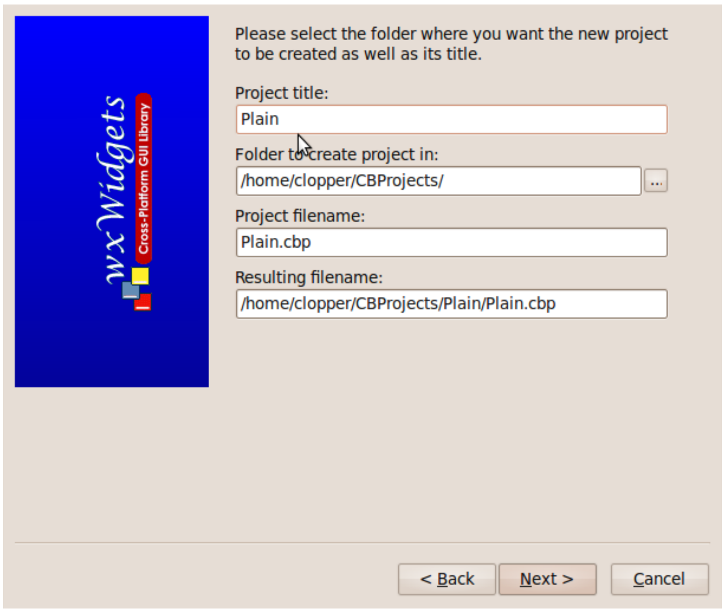
Please select the wxWidgets version you want to use.

Please make a selection

wxWidgets 2.6.x
wxWidgets 2.8.x
wxWidgets 2.9.x (CVS Version)

Select wxWidgets 2.8x and click the "Next >" button at the bottom of the screen.

That brings you to the window where you specify the Project title and folder where you want to save the project. Other fields are then filled automatically. I chose “Plain” as the Project title and the other fields then adjusted to look like this:


Please select the folder where you want the new project to be created as well as its title.

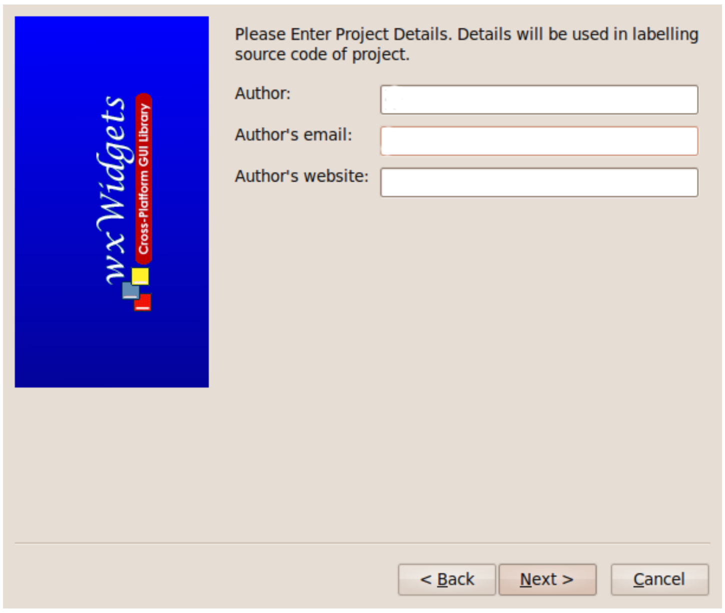
Project title: Plain

Folder to create project in: /home/clopper/CBProjects/

Project filename: Plain.cbp

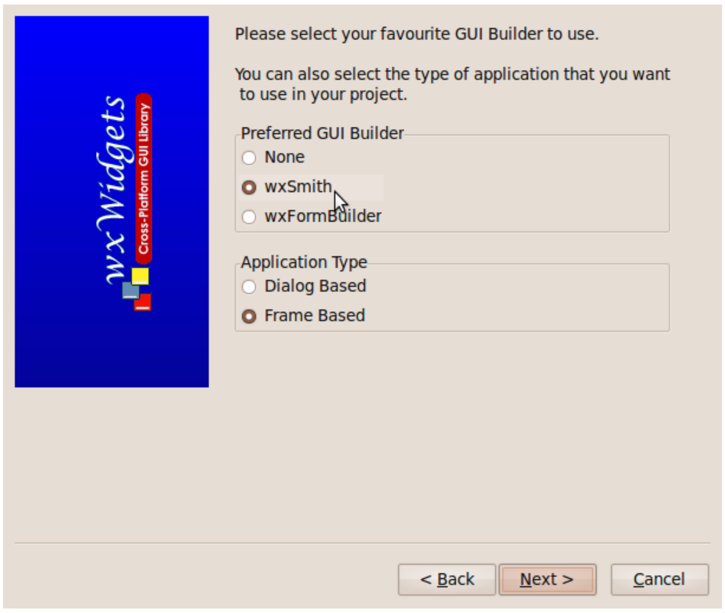
Resulting filename: /home/clopper/CBProjects/Plain/Plain.cbp

Then click the “Next >” button at the bottom. The next screen, shown below, allows you to mark your work with your identity. Fill in what you wish.


Please Enter Project Details. Details will be used in labelling source code of project.

Author:	
Author's email:	
Author's website:	

That brings up the window for choosing the GUI builder you want to use, wxSmith or FormBuilder. We choose, of course, wxSmith.


Please select your favourite GUI Builder to use.

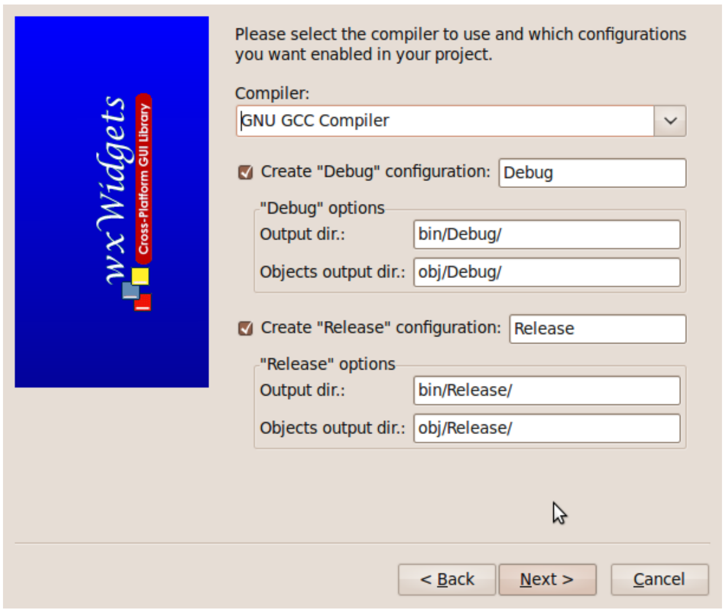
You can also select the type of application that you want to use in your project.

Preferred GUI Builder: None [✓ wxSmith] wxFormBuilder

Application Type: Dialog Based [✓ Frame Based]

Click the "Next" button at the bottom.

The next screen specifies what C++ compiler we want to use and whether we want a debug build (with extra code for debugging) or a smaller release build for use by others. We will go for the GNU CPP compiler and both build options, to be put in separate subfolders.



Please select the compiler to use and which configurations you want enabled in your project.

Compiler: GNU GCC Compiler

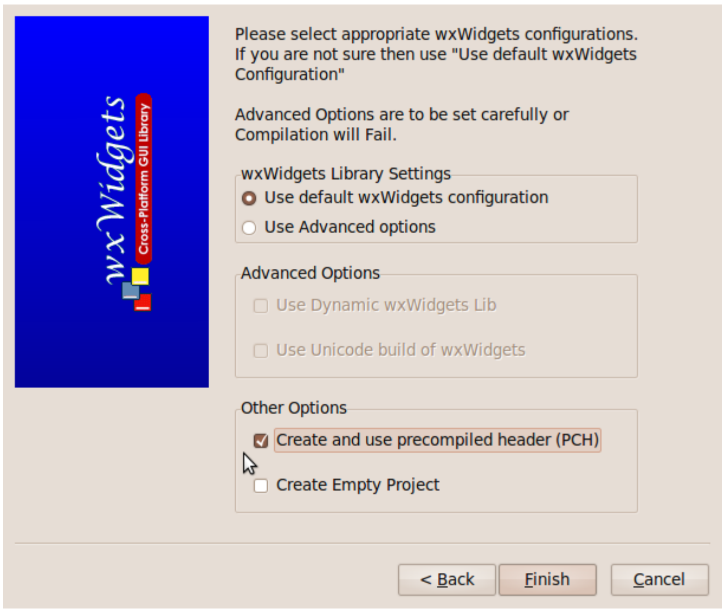
[✓] Create "Debug" configuration: Debug

"Debug" options
Output dir.: bin/Debug/
Objects output dir.: obj/Debug/

[✓] Create "Release" configuration: Release

"Release" options
Output dir.: bin/Release/
Objects output dir.: obj/Release/

The final screen of this sort asks whether we want to use the standard, default, configuration or to use Advanced options. We will take the default. But, at the bottom, we choose to “Create and use precompiled headers" which speeds up compilations after the first.



Please select appropriate wxWidgets configurations. If you are not sure then use "Use default wxWidgets Configuration"

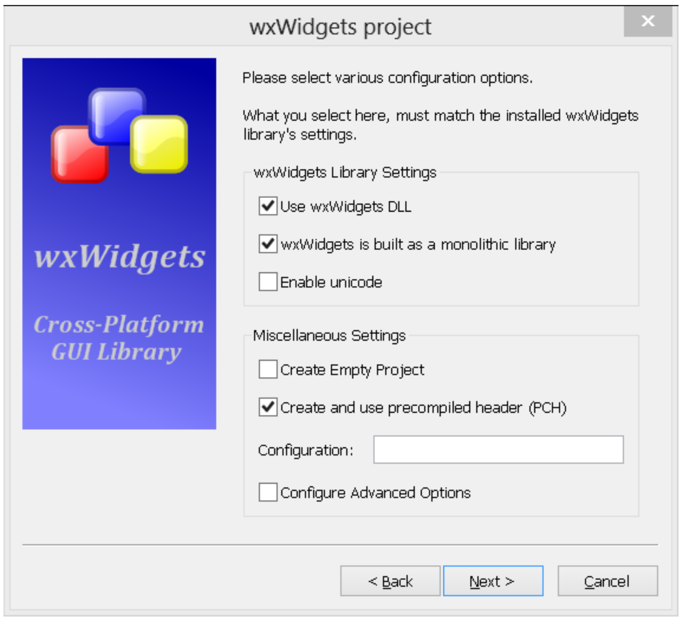
Advanced Options are to be set carefully or Compilation will Fail.

wxWidgets Library Settings
● Use default wxWidgets configuration
○ Use Advanced options

Advanced Options
[ ] Use Dynamic wxWidgets Lib
[ ] Use Unicode build of wxWidgets

Other Options
[✓] Create and use precompiled header (PCH)
[ ] Create Empty Project

In wxWidgets 3.0.2 the screen looks like this. WardXmodem 21:38, 20 May 2015 (CEST)



wxWidgets project

Please select various configuration options.

What you select here, must match the installed wxWidgets library's settings.

wxWidgets Library Settings
[✓] Use wxWidgets DLL
[✓] wxWidgets is built as a monolithic library
[ ] Enable unicode

Miscellaneous Settings
[ ] Create Empty Project
[✓] Create and use precompiled header (PCH)
Configuration:
[ ] Configure Advanced Options

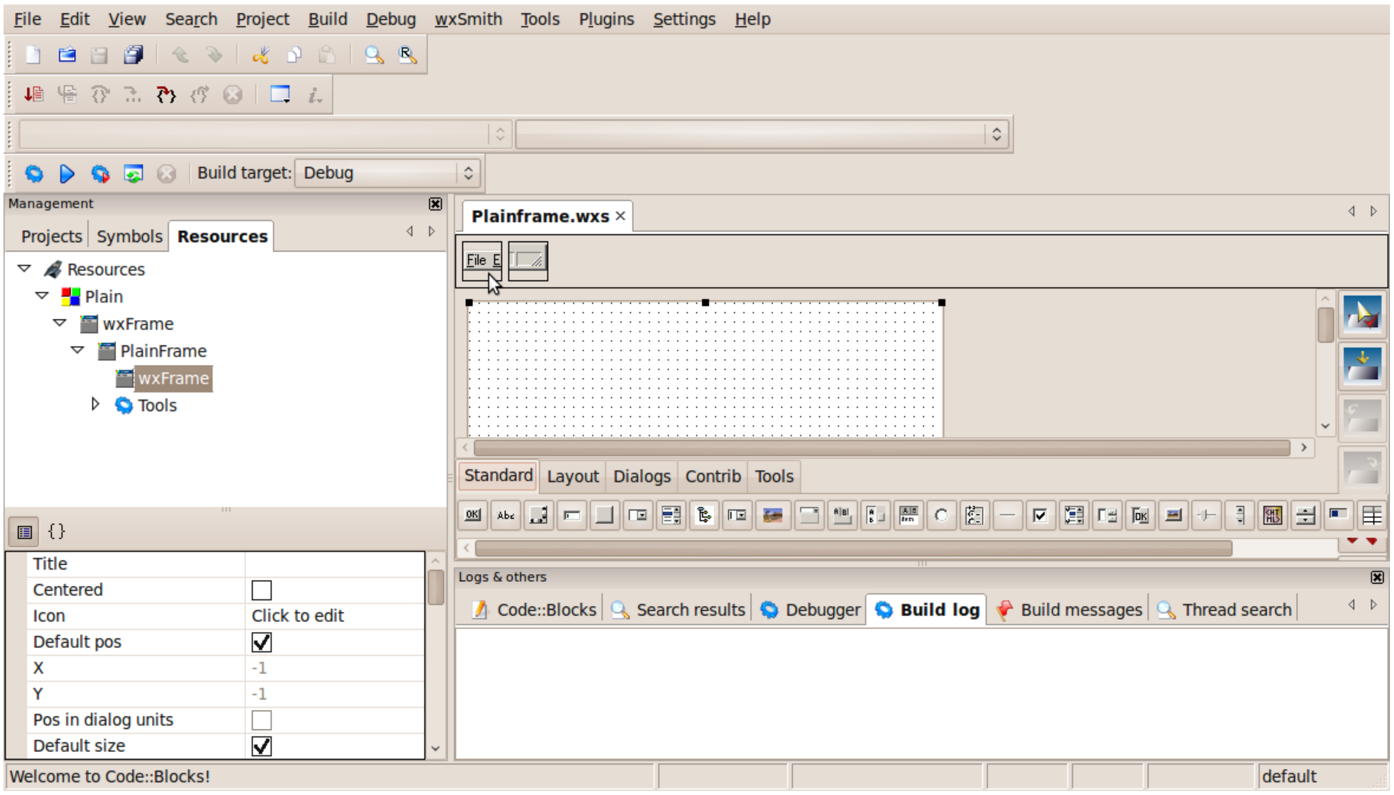
On the form where we chose the directory for saving the project, we chose a real directory, but that choice may lead to problems when you want to compile your project on other computers. Another choice is to use the **Global Variable system**. To open the dialog below or modify or change this global variable, you should click on the Code::Blocks main menu item **Settings | Global variables**. That brings up the Global Variables Editor.

When you use it, the location of the wxWidgets library is read from the variable you supply. This means that the location of the wxWidgets directory is not hard-coded inside your project, but is stored inside the global configuration of Code::Blocks. When your project is opened on a different computer, it will use that comoputer's configuration. The easiest option to use this global variable is to leave *$(#wx)* which means to use the global variable named *wx*. This name is reserved for wxWidgets and should be chosen in most cases.

When you use the global variable for the first time, it will ask for the wxWidgets directory. The only thing you need to do here is to enter a valid location in *base* field. This is required only for the first time since Code::Blocks will automatically use this setting for other projects you create.

Overview of the Interactive Development Environment (IDE)

At the end of this preliminary sequence of forms we come to the Code::Blocks main IDE window. It should look something like this:


File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help
Build target: Debug
Management
Projects Symbols Resources
Resources
Plain
wxFrame
PlainFrame
wxFrame
Tools
Plainframe.wxs
Standard Layout Dialogs Contrib Tools
Logs & others
Code::Blocks Search results Debugger Build log Build messages Thread search
Welcome to Code::Blocks!
default

Title	
Centered	☐
Icon	Click to edit
Default pos	✓
X	-1
Y	-1
Pos in dialog units	☐
Default size	✓

In the middle of the right side of the window, under the word “Plainframe.wxs”, is a panel with two square icons on the left; the one on the left brings up the menu bar editor while the one on the right brings up the status bar editor. We will return to the menu bar editor in the next tutorial. Below this panel and looking like a field of dots is the Editor window, where we will see both the GUI forms which we build and the C++ code we write. In the lower left corner of the whole window is the Properties/Events window, which will allow us to set properties of the component we are working on or to pick events, such as “OnClick” , to which we need to respond. To show properties, click on the left icon; to show events, click on the right icon marked {} to suggest C++ code.

Above the Properties/Events window is the Management window with three tabs, Projects, Symbols, and Resources. If all three tabs are not visible, drag the right border of the window to the right to reveal all three. (A “resource” in wxSmith parlance usually means a “form” or "window".) Above the Management window is a panel with several small but useful icons ( ) to

- Build the program, or
- Run it, or
- Build and run it, or
- Rebuild it, or
- Abort it.

When you run the cursor over them, the tool tips will tell you which is which.

Now look back to the Editor window. First of all, note that the window can be configured to use different colors,fonts and font sizes. Click "Settings ..." on the Code::Blocks main menu, and then "Editor". The screens are more or less self-explanatory.

Now back to the Editor window. Across the top you see three black square called drag handles. They have been set to cover a space larger than can be seen, but scrolling down reveals the whole field.

As an experienced programmer, you know there has got to be some code somewhere but you don't see it. To reveal it, double left click in the field of dots. Presto, there is the C++ code, and the *PlainMain.cpp tab has appeared above it. Clicking back on the Plainframe.wxs should return you to the view of the form, still the field of dots. (Again, it is important to have CB 10.04 or later.) In fact, we still have not seen all of the code. In particular, you will search in vain for the all-important OnInit() routine that starts the program. You can find it, however, by clicking Search | Find in Files and searching for OnInit with the scope of the search set to be “Project files”. The search finds it, sure enough, in PlainApp.cpp and opens this module in the Editor window.

The following discussion on having to add close() is no longer true, at least at Code::Blocks 13.12, wxWidgets 3.0.2.

We don't need OnInit right now, but we do need to make a change in PlainMain.cpp. Click on it in the bar above the editor, and its code appears in the editor.

Sometimes a program has a lot to do when the user clicks the close icon in the program's title bar, so wxSmith generates a frame for writing the code for those actions but it provides no code. Consequently, if we ran our program just as initially written by wxSmith, we won't be able to close it by just clicking that icon. To fix that inconvenient problem, find the spot in the code with these lines:

```
void PlainFrame::OnClose(wxCloseEvent& event)
{
}
```

Add the call to the Close() function so that the spot looks like this:

```
void PlainFrame::OnClose(wxCloseEvent& event)
{
    Close();
}
```

(*I found it had to be Destroy() not Close() WardXmodem 04:43, 24 September 2013 (CEST)*)

(*use event.Skip(TRUE); instead of Close(); or Destroy(). (Some) later tutorials correctly reference this. Ensnarer 03.08.2015*)

Remember to make this change immediately whenever you start a new project.

End of old "have to add close()" section. With newer releases the following is done for you:

```
void PlainFrame::OnQuit(wxCommandEvent& event)
{
    Close();
}
```

WardXmodem 22:43, 20 May 2015 (CEST)

End of updated close() information.

You can now build and run this super-simple program. To do so, you can do any one of three equivalent things:

- in the top menu bar, click Build | Build and run.
- find and click the icon that looks like a right-pointing triangle in front of a cogwheel. Hovering the mouse pointer over it will show the tool tip “Build and run”.
- tap the F9 key.

If all goes well, you will see a window come up with a menu of two items: “File” and “Help”. Try the “Help” menu item first. Then you can use “File | Quit” or the exit icon on the title bar to exit the program.

If you are using Ubuntu and get no main menu bar when you run this program, you have fallen victim to Ubuntu bug 662077 (https://bugs.launchpad.net/ubuntu/+source/wxwidgets2.8/+bug/662077). Beginning with release 10.10, Ubuntu releases have had a bug causing programs built with wxSmith to fail to show their main menu bar. If this bug affects you, you may want to uninstall appmenu-gtk. You do that by clicking Applications | Software Sources, then using the search window to find appmenu-gtk. Clicking on it will allow you to choose to remove it.

Well, so far we have a program that runs and quits, but it does not yet display "Hello, World!". That's next.

Hello, World!

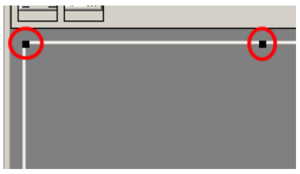
We now turn to the classic first program, namely getting the words “Hello, World!” onto the screen. The text could be added directly into the frame, as in GUI designers for a single operating sytem; but we need to learn to use sizers for cross-platform programming. What are sizers?

If you have been working with Java or Qt, you may remember something called Layout managers. The implementation in wxWidgets differs a little but does almost the same thing.

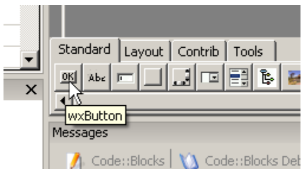
When adding components into windows you must somehow specify their position and size. wxWidgets tries to automate this process, and it uses sizers to do so. They automatically position and size window components, and they can also set the size of the window so that everything fits inside.

Sizers have one big advantage over direct placement of components. When you write cross-platform applications, you cannot assume that fonts, buttons, and so on have the same size on different operating systems. This problem can even occur on the same platform when using different window themes. When you use sizers, you don't have to worry about those problems. All sizing and placement is done automatically. Moreover, sizers can even reposition and resize window components when the main window changes size.

With that background, let's get started. When you look at the wxSmith editor, you'll notice that the area of the window is surrounded by eight black boxes or drag handles.



These drag handles surround the currently selected item. In our case it's a whole window. Adding new components to the window is simply done by clicking on one of the buttons in the palette at the bottom of the editor and then clicking somewhere on the resource. As you can see here,

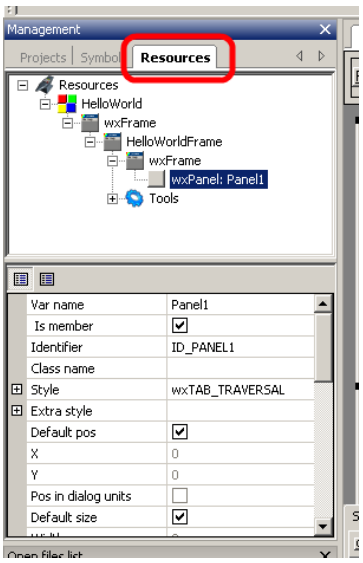


when you keep your mouse cursor over a component in the palette, it will show its name. That feature may help you find components until you become familiar with their icons.

Let's add a new wxPanel component to our window; it is the fourth icon from the left on the standard tab. You will see that when you click the wxPanel item and then move the cursor over the editor, some part of the editor will change color. That's intentional and is used to show where on the window the new item will be added.

After the wxPanel is in place, you will notice that the background color of the window has changed to light gray. Now our window will look much more like other windows. Note that currently the selection changed to the added panel but since its size is equal to the frame's, it's hard to find out what's really selected. To make sure that wxPanel is selected, click somewhere on the light gray area of our window (Note to Linux users: wxFrame and wxPanel usually have the same background so it may look like nothing has changed; you may ensure that there's a wxPanel by looking into the resource browser).

If you want to see the selection exactly or select items that overlap, you can use the resource browser. To show it, click on the resources tab located in the same place where the project browser is:



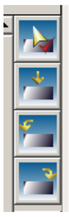
Var name	Panel1
Is member	✓
Identifier	ID_PANEL1
Class name	
Style	wxTAB_TRAVERSAL
Extra style	
Default pos	✓
X	0
Y	0
Pos in dialog units	
Default size	✓

The resource tab consists of two parts. The first is the resource tree which shows the structure of your resources and the second one is the property browser which is common in GUI designers. Inside the resource tree you can find the currently selected item. In our case it is the wxPanel class we added before. The selected item also shows the name of this item: Panel1. This name is basically the variable which can be used to access this component in your code.

We added a new component by selecting the place for it. This is wxSmith's default behavior. However, there is another mode of adding items. In this mode, the new item is added right after you click on the palette's button. The position of the new item is relative to the current selection - just like a cursor in text editors. Basically we can add new items into three different positions relative to the selection:

- Before the selected widget
- After the selected widget
- Into the selected widget.

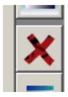
This insertion mode can be changed by clicking one of the four "placement" buttons on the right side of the editor:



The uppermost is the mode we used - "point by mouse". The next one means "add into", followed by "add before", and the last one means "add after". The current mode is shown by a red check mark on the button.

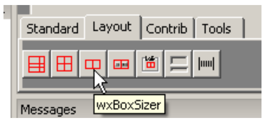
Sometimes a few selection modes are disabled because they are not valid for the currently selected item. For example, you cannot add any component into the wxButton component since wxWidgets declares it as one that cannot have child components. Similarly, when the main component (the window) is selected, you won't be able to add anything after it or before it.

Ooops! I said we would use sizers and then forgot all about them! We have to get rid of that wxPanel. Select the panel and then click on the red X button just below the four buttons we have been describing. It looks like this:



When you click this button, it deletes the current selection, so if the panel didn't disappear, make sure it's selected and click the button again. Now we should have returned to the state at the beginning.

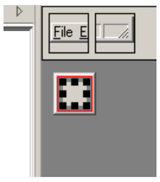
To use sizers, first we have to add a sizer into the main frame and then add the wxPanel into it. Sizers are available in the Layout tab on the palette. Switch to it and select wxBoxSizer:



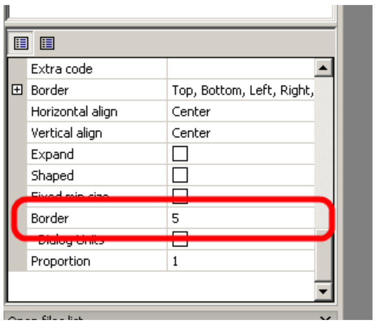
This sizer tries to position items in one line, either horizontal or vertical. Horizontal is the default, and we will use it.

After adding the sizer, two things have changed. The first is that our window now has a red border. This means that there is a sizer attached to it. When you look into the resource browser, you will also see that the sizer has been added. (You may need to click on the editor area to refresh the selection on the resource browser.) The second change is that the size of the window has shrunk to a small box. That is as it should be because the sizer is responsible for resizing the window to be no bigger than necessary to be visible and accommodate its components, and so far there are no components.

Now let's add our wxPanel. Make sure that you add it into the sizer, not into the main frame. After you do so, you will see something like this:

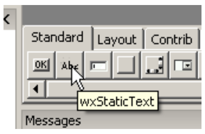


Here we can see our panel surrounded by drag boxes. There's also some dark gray border around it (it won't be visible on Linux). Each component added into the sizer can be surrounded by such a border. It's useful when you want to have spacing between components. But in our case we should get rid of it to get a better looking window. To do so, we will have to change a property inside the property browser. Search for border size and change it from 5 to 0:

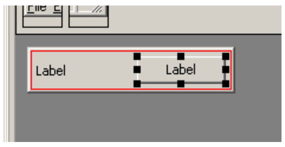


Extra code	
Border	Top, Bottom, Left, Right,
Horizontal align	Center
Vertical align	Center
Expand	☐
Shaped	☐
Fixed min size	☐
Border	5
Dialog Units	☐
Proportion	1

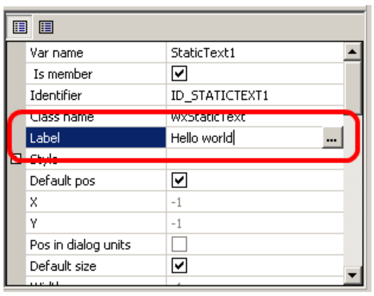
Now when we have our panel adjusted, we can finally add the "Hello world" text. Since we will also use sizers to manage items added into wxPanel, we have to repeat the addition of wxBoxSizer into wxPanel. After the sizer is in its place, switch back into the **Standard** tab on the palette and add a wxStaticText control:



With an eye to the future and for practice with adding components, let's add also a button next to the wxStaticText. First, click on wxButton. If you still have the "point by mouse" insertion mode turned on, then when you put the mouse over the wxStaticText component half of that component will be colored in light blue. When the left half of the text is highlighted, the new item will be added before the existing component; and when the right half is highlighted, the new component will be added after the existing one. Let's add the button after the text. Now we should have this content:



Now let's finally set our "Hello, World!" text. The wxStaticText control which we added before has the text "Label" on it. It is the default text set to all newly created wxStaticText classes. We will change it using the property browser similarly to changing the border size in the wxPanel class. To do so, select the wxStaticText component in our window, then find the Label property in the property browser and change it to "Hello, World!":



Var name	StaticText1
Is member	☑
Identifier	ID_STATICTEXT1
Class name	wxStaticText
Label	Hello world
Style	
Default pos	☑
X	-1
Y	-1
Pos in dialog units	☐
Default size	☑

If you want two or more lines of text on the wxStaticText control. click on the "..." button at the right end of the “Label” line or put "\n" inside the text, as in C formats.

At this point, you can click the "Build and run" and the program should run and show the magic words in the static text control. We could quit in triumph at this point, but the button we have put on the form is still a dud; clicking on it will do nothing. Let's give it a function.

Before going further, we need to deal with a question you may be asking. Why did we, after putting a box sizer onto our main window, proceed to put a wxPanel into it and then put another box sizer onto the panel? What is the point of the panel? Why not just put our two buttons into the first box sizer and eliminate the panel and second box sizer? The answer is that the panel gives us control of the background. We didn't use it, but it is often useful to be able to control it. In fact, these first three steps will be used in structuring every window used in these tutorials. To review, these three steps are:

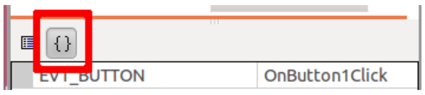
- Put a box sizer on the window
- Put a panel into the first box of the sizer

- Put a sizer of some sort onto the panel.

Responding to the Click of a Button

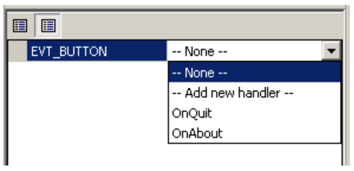
Since buttons usually cause something to happen when clicked, let's make our button close the window. The first thing we should do is to change the text shown on the button to tell the user what the button does. It is done just as in the case of the wxStaticText widget. Select the button, find "Label" in the property browser and set it to "Close" (or anything you like).

The second thing is to assign an event handler to this button. Basically, event handlers are functions which are called when some "event" occurs. In our case instead of "event" we could say more descriptively " a click on the button" because that is what we want to process. Like other GUI designers, wxSmith can automatically generate the framework code for such handlers. To bring up that ability, switch the property browser to the event browser by clicking on the second icon at the top of the property browser. The icon looks like {} to suggest C++ code.



EVT_BUTTON	OnButton1Click

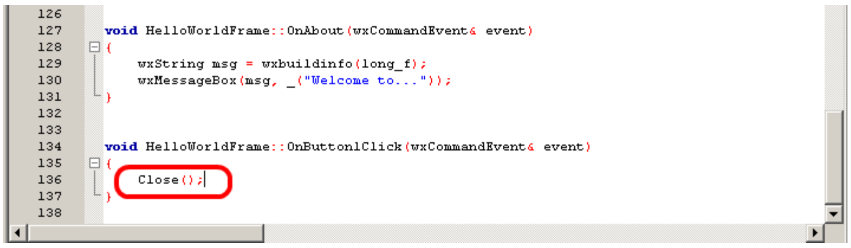
After you click on this icon, you will see all events supported by the currently selected component. The wxButton class supports only one event, the EVT_BUTTON event, so only this one is shown. When you click on this event and drop down the combo box, you will see the available options and handlers for this event:



EVT_BUTTON	-- None --

If you select **-- None --** no handler function will be automatically assigned to this event. (But it may always be set manually.) Selecting **-- Add new handler --** will generate a frame for the code of the new handler function. Below are shown other event handlers known to wxSmith which can be assigned to this event. We can see here OnQuit and OnAbout. Those handlers were created by wizard. We could use the OnQuit handler, but to see what happens, let's write our own handler. Select the **Add new handler** option. You will be asked for a name for the handler. This name is simply the name of the function which will be called as a reaction to this event. wxSmith will suggest the name OnButton1Click, and we may as well use it.

When you enter a new handler name, wxSmith will generate the framing code of a new handler function by that name, open the program file with the proper source code and put the cursor inside the handler function. (You may need to scroll up or down to show the line with cursor.) This function is a member of the class we have been building, so it can access all of functions in that class, including the one called **Close()**, which was created by the wizard. So when we want to close our window we just call the **Close()** function. The result looks like this:



```
void HelloWorldFrame::OnAbout(wxCommandEvent& event)
{
    wxString msg = wxbuildinfo(long_f);
    wxMessageBox(msg, _("Welcome to..."));
}


void HelloWorldFrame::OnButton1Click(wxCommandEvent& event)
{
    Close();
}
```

Build and run our project. It should show our "Hello, World!" text and our "Close" button. Click the Close button, and it should close.

This is end of this first tutorial, I hope that you followed it easily and are beginning to feel comfortable with Code::Blocks and wxSmith.

The tutorial was originally written by BYO, the creator of wxSmith. It was revised in March 2012 by Grouch to take account of changes in Code::Blocks, add a few clarifications, and change the accent in the English from youthful Polish to elderly American.

Index | Next

Retrieved from "https://wiki.codeblocks.org/index.php?title=WxSmith_tutorial:_Hello_world&oldid=9201"