Guide to porting applications from wxWindows 1.xx to 2.0

Julian Smart

March 1999

Contents

1 About this document

This document gives guidelines and tips for porting applications from version 1.xx of wxWindows to version 2.0.

The first section offers tips for writing 1.xx applications in a way to minimize porting time. The following sections detail the changes and how you can modify your application to be 2.0-compliant.

You may be worrying that porting to 2.0 will be a lot of work, particularly if you have only recently started using 1.xx. In fact, the wxWindows 2.0 API has far more in common with 1.xx than it has differences. The main challenges are using the new event system, doing without the default panel item layout, and the lack of automatic labels in some controls.

Please don't be freaked out by the jump to 2.0! For one thing, 1.xx is still available and will be supported by the user community for some time. And when you have changed to 2.0, we hope that you will appreciate the benefits in terms of greater flexibility, better user interface aesthetics, improved C++ conformance, improved compilation speed, and many other enhancements. The revised architecture of 2.0 will ensure that wxWindows can continue to evolve for the forseeable future.

Please note that this document is a work in progress.

2 Preparing for version 2.0

Even before compiling with version 2.0, there's also a lot you can do right now to make porting relatively simple. Here are a few tips.

- **Use constraints or .wxr resources** for layout, rather than the default layout scheme. Constraints should be the same in 2.0, and resources will be translated.

- **Use separate wxMessage items** instead of labels for wxText, wxMultiText, wxChoice, wxComboBox. These labels will disappear in 2.0. Use separate wxMessages whether you're creating controls programmatically or using the dialog editor. The future dialog editor will be able to translate from old to new more accurately if labels are separated out.

- **Parameterise functions that use wxDC** or derivatives, i.e. make the wxDC an argument to all functions that do drawing. Minimise the use of wxWindow::GetDC and definitely don't store wxDCs long-term because in 2.0, you can't use GetDC() and wxDCs are not persistent. You will use wxClientDC, wxPaintDC stack objects instead. Minimising the use of GetDC() will ensure that there are very few places you have to change drawing code for 2.0.

- **Don't set GDI objects** (wxPen, wxBrush etc.) in windows or wxCanvasDCs before they're needed (e.g. in constructors) - do so within your drawing routine instead. In 2.0, these settings will only take effect between the construction and destruction of temporary wxClient/PaintDC objects.

- **Don't rely** on arguments to wxDC functions being floating point - they will be 32-bit integers in 2.0.

- **Don't use the wxCanvas member functions** that duplicate wxDC functions, such as SetPen and DrawLine, since they are going.

- **Using member callbacks** called from global callback functions will make the transition easier - see the FAQ for some notes on using member functions for callbacks. wxWindows 2.0 will banish global callback functions (and OnMenuCommand), and nearly all event handling will be done by functions taking a single event argument. So in future you will have code like:

```
void MyFrame::OnOK(wxCommandEvent&event)

        ...
```

  You may find that writing the extra code to call a member function isn't worth it at this stage, but the option is there.

- **Use wxString wherever possible.** 2.0 replaces char * with wxString in most cases, and if you use wxString to receive strings returned from wxWindows functions (except when you need to save the pointer if deallocation is required), there should be no conversion problems later on.

- Be aware that under Windows, **font sizes will change** to match standard Windows font sizes (for example, a 12-point font will appear bigger than before). Write your application to be flexible where fonts are concerned. Don't rely on fonts being similarly-sized across platforms, as they were (by chance) between Windows and X under wxWindows 1.66.

Yes, this is not easy... but I think it's better to conform to the standards of each platform, and currently the size difference makes it difficult to conform to Windows UI standards. You may eventually wish to build in a global 'fudge-factor' to compensate for size differences. The old font sizing will still be available via wx_setup.h, so do not panic...

- **Consider dropping wxForm usage**: wxPropertyFormView can be used in a wxForm-like way, except that you specify a pre-constructed panel or dialog; or you can use a wxPropertyListView to show attributes in a scrolling list - you don't even need to lay panel items out.

  Because wxForm uses a number of features to be dropped in wxWindows 2.0, it cannot be supported in the future, at least in its present state.

- **When creating a wxListBox**, put the wxLB_SINGLE, wxLB_MULTIPLE, wxLB_EXTENDED styles in the window style parameter, and put zero in the *multiple* parameter. The *multiple* parameter will be removed in 2.0.

- **For MDI applications**, don't reply on MDI being run-time-switchable in the way that the MDI sample is. In wxWindows 2.0, MDI functionality is separated into distinct classes.

3 The new event system

The way that events are handled has been radically changed in wxWindows 2.0. Please read the topic 'Event handling overview' in the wxWindows 2.0 manual for background on this.

3.1 Callbacks

Instead of callbacks for panel items, menu command events, control commands and other events are directed to the originating window, or an ancestor, or an event handler that has been plugged into the window or its ancestor. Event handlers always have one argument, a derivative of wxEvent.

For menubar commands, the **OnMenuCommand** member function will be replaced by a series of separate member functions, each of which responds to a particular command. You need to add these (non-virtual) functions to your frame class, add a DECLARE_EVENT_TABLE entry to the class, and then add an event table to your implementation file, as a BEGIN_EVENT_TABLE and END_EVENT_TABLE block. The individual event mapping macros will be of the form:

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(MYAPP_NEW, MyFrame::OnNew)
    EVT_MENU(wxID_EXIT, MyFrame::OnExit)
END_EVENT_TABLE()
```

Control commands, such as button commands, can be routed to a derived button class, the parent window, or even the frame. Here, you use a function of the form EVT_BUTTON(id, func). Similar macros exist for other control commands.

3.2 Other events

To intercept other events, you used to override virtual functions, such as OnSize. Now, while you can use the OnSize name for such event handlers (or any other name of your choice), it has only a single argument (wxSizeEvent) and must again be 'mapped' using the EVT_SIZE macro. The same goes for all other events, including OnClose (although in fact you can still use the old, virtual form of OnClose for the time being).

4 Class hierarchy

The class hierarchy has changed somewhat. wxToolBar and wxButtonBar classes have been split into several classes, and are derived from wxControl (which was called wxItem). wxPanel derives from wxWindow instead of from wxCanvas, which has disappeared in favour of wxScrolledWindow (since all windows are now effectively canvases which can be drawn into). The status bar has become a class in its own right, wxStatusBar.

There are new MDI classes so that wxFrame does not have to be overloaded with this functionality.

There are new device context classes, with wxPanelDC and wxCanvasDC disappearing. See *Device contexts and painting* (p. 9).

5 GDI objects

These objects - instances of classes such as wxPen, wxBrush, wxBitmap (but not wxColour) - are now implemented with reference-counting. This makes assignment a very cheap operation, and also means that management of the resource is largely automatic. You now pass *references* to objects to functions such as wxDC::SetPen, not pointers, so you will need to derefence your pointers. The device context does not store a copy of the pen itself, but takes a copy of it (via reference counting), and the object's data gets freed up when the reference count goes to zero. The application does not have to worry so much about who the object belongs to: it can pass the reference, then destroy the object without leaving a dangling pointer inside the device context.

For the purposes of code migration, you can use the old style of object management - maintaining pointers to GDI objects, and using the FindOrCreate... functions. However, it is preferable to keep this explicit management to a minimum, instead creating objects on the fly as needed, on the stack, unless this causes too much of an overhead in your application.

At a minimum, you will have to make sure that calls to SetPen, SetBrush etc. work. Also, where you pass NULL to these functions, you will need to use an identifier such as wxNullPen or wxNullBrush.

6 Dialogs and controls

Labels

Most controls no longer have labels and values as they used to in 1.xx. Instead, labels should be created separately using wxStaticText (the new name for wxMessage). This will need some reworking of dialogs, unfortunately; programmatic dialog creation that doesn't use constraints will be especially hard-hit. Perhaps take this opportunity to make more use of dialog resources or constraints. Or consider using the wxPropertyListView class which can do away with dialog layout issues altogether by presenting a list of editable properties.

Constructors

All window constructors have two main changes, apart from the label issue mentioned above. Windows now have integer identifiers; and position and size are now passed as wxPoint and wxSize objects. In addition, some windows have a wxValidator argument.

Show versus ShowModal

If you have used or overridden the **wxDialog::Show** function in the past, you may find that modal dialogs no longer work as expected. This is because the function for modal showing is now **wxDialog:ShowModal**. This is part of a more fundamental change in which a control may tell the dialog that it caused the dismissal of a dialog, by calling **wxDialog::EndModal** or **wxWindow::SetReturnCode**. Using this information, **ShowModal** now returns the id of the control that caused dismissal, giving greater feedback to the application than just TRUE or FALSE.

If you overrode or called **wxDialog::Show**, use **ShowModal** and test for a returned identifier, commonly wxID_OK or wxID_CANCEL.

wxItem

This is renamed wxControl.

wxText, wxMultiText and wxTextWindow

These classes no longer exist and are replaced by the single class wxTextCtrl. Multi-line text items are created using the wxTE_MULTILINE style.

wxButton

Bitmap buttons are now a separate class, instead of being part of wxBitmap.

wxMessage

Bitmap messages are now a separate class, wxStaticBitmap, and wxMessage is renamed wxStaticText.

wxGroupBox

wxGroupBox is renamed wxStaticBox.

wxForm

Note that wxForm is no longer supported in wxWindows 2.0. Consider using the wxPropertyFormView class instead, which takes standard dialogs and panels and associates

controls with property objects. You may also find that the new validation method, combined with dialog resources, is easier and more flexible than using wxForm.

7 Device contexts and painting

In wxWindows 2.0, device contexts are used for drawing into, as per 1.xx, but the way they are accessed and constructed is a bit different.

You no longer use **GetDC** to access device contexts for panels, dialogs and canvases. Instead, you create a temporary device context, which means that any window or control can be drawn into. The sort of device context you create depends on where your code is called from. If painting within an **OnPaint** handler, you create a wxPaintDC. If not within an **OnPaint** handler, you use a wxClientDC or wxWindowDC. You can still parameterise your drawing code so that it doesn't have to worry about what sort of device context to create - it uses the DC it is passed from other parts of the program.

You **must** create a wxPaintDC if you define an OnPaint handler, even if you do not actually use this device context, or painting will not work correctly under Windows.

If you used device context functions with wxPoint or wxIntPoint before, please note that wxPoint now contains integer members, and there is a new class wxRealPoint. wxIntPoint no longer exists.

wxMetaFile and wxMetaFileDC have been renamed to wxMetafile and wxMetafileDC.

8 Miscellaneous

8.1 Strings

wxString has replaced char* in the majority of cases. For passing strings into functions, this should not normally require you to change your code if the syntax is otherwise the same. This is because C++ will automatically convert a char* or const char* to a wxString by virtue of appropriate wxString constructors.

However, when a wxString is returned from a function in wxWindows 2.0 where a char* was returned in wxWindows 1.xx, your application will need to be changed. Usually you can simplify your application's allocation and deallocation of memory for the returned string, and simply assign the result to a wxString object. For example, replace this:

```
char* s = wxFunctionThatReturnsString();
s = copystring(s); // Take a copy in case it's temporary
.... // Do something with it
delete[] s;
```

with this:

```
wxString s = wxFunctionThatReturnsString();
.... // Do something with it
```

To indicate an empty return value or a problem, a function may return either the empty string ("") or a null string. You can check for a null string with wxString::IsNull().

8.2 Use of const

The **const** keyword is now used to denote constant functions that do not affect the object, and for function arguments to denote that the object passed cannot be changed.

This should not affect your application except for where you are overriding virtual functions which now have a different signature. If functions are not being called which were previously, check whether there is a parameter mismatch (or function type mismatch) involving consts.

Try to use the **const** keyword in your own code where possible.

9 Backward compatibility

Some wxWindows 1.xx functionality has been left to ease the transition to 2.0. This functionality (usually) only works if you compile with WXWIN_COMPATIBILITY set to 1 in setup.h.

Mostly this defines old names to be the new names (e.g. wxRectangle is defined to be wxRect).

10 Quick reference

This section allows you to quickly find features that need to be converted.

10.1 Include files

Use the form:

```
#include <wx/wx.h>
#include <wx/button.h>
```

For precompiled header support, use this form:

```
// For compilers that support precompilation, includes "wx.h".
#include <wx/wxprec.h>

#ifdef __BORLANDC__
    #pragma hdrstop
#endif

// Any files you want to include if not precompiling by including
// the whole of <wx/wx.h>
#ifndef WX_PRECOMP
    #include <stdio.h>
    #include <wx/setup.h>
    #include <wx/bitmap.h>
    #include <wx/brush.h>
#endif

// Any files you want to include regardless of precompiled headers
#include <wx/toolbar.h>
```

10.2 IPC classes

These are now separated out into wxDDEServer/Client/Connection (Windows only) and wxTCPServer/Client/Connection (Windows and Unix). Take care to use wxString for your overridden function arguments, instead of char*, as per the documentation.

10.3 MDI style frames

MDI is now implemented as a family of separate classes, so you can't switch to MDI just by using a different frame style. Please see the documentation for the MDI frame classes, and the MDI sample may be helpful too.

10.4 OnActivate

Replace the arguments with one wxActivateEvent& argument, make sure the function isn't virtual, and add an EVT_ACTIVATE event table entry.

10.5 OnChar

This is now a non-virtual function, with the same wxKeyEvent& argument as before. Add an EVT_CHAR macro to the event table for your window, and the implementation of your function will need very few changes.

10.6 OnClose

The old virtual function OnClose is now obsolete. Add an OnCloseWindow event handler using an EVT_CLOSE event table entry. For details about window destruction, see the Windows Deletion Overview in the manual. This is a subtle topic so please read it very carefully. Basically, OnCloseWindow is now responsible for destroying a window with Destroy(), but the default implementation (for example for wxDialog) may not destroy the window, so to be sure, always provide this event handler so it's obvious what's going on.

10.7 OnEvent

This is now a non-virtual function, with the same wxMouseEvent& argument as before. However you may wish to rename it OnMouseEvent. Add an EVT_MOUSE_EVENTS macro to the event table for your window, and the implementation of your function will need very few changes. However, if you wish to intercept different events using different functions, you can specify specific events in your event table, such as EVT_LEFT_DOWN.

Your OnEvent function is likely to have references to GetDC(), so make sure you create a wxClientDC instead. See *Device contexts* (p. 9).

If you are using a wxScrolledWindow (formerly wxCanvas), you should call PrepareDC(dc) to set the correct translation for the current scroll position.

10.8 OnMenuCommand

You need to replace this virtual function with a series of non-virtual functions, one for each case of your old switch statement. Each function takes a wxCommandEvent& argument. Create an event table for your frame containing EVT_MENU macros, and insert DECLARE_EVENT_TABLE() in your frame class, as per the samples.

10.9 OnPaint

This is now a non-virtual function, with a wxPaintEvent& argument. Add an EVT_PAINT macro to the event table for your window.

Your function *must* create a wxPaintDC object, instead of using GetDC to obtain the device context.

If you are using a wxScrolledWindow (formerly wxCanvas), you should call PrepareDC(dc) to set the correct translation for the current scroll position.

10.10 OnSize

Replace the arguments with one wxSizeEvent& argument, make it non-virtual, and add to your event table using EVT_SIZE.

10.11 wxApp definition

The definition of OnInit has changed. Return a bool value, not a wxFrame.

Also, do *not* declare a global application object. Instead, use the macros DECLARE_APP and IMPLEMENT_APP as per the samples. Remove any occurrences of IMPLEMENT_WXWIN_MAIN: this is subsumed in IMPLEMENT_APP.

10.12 wxButton

For bitmap buttons, use wxBitmapButton.

10.13 wxCanvas

Change the name to wxScrolledWindow.

10.14 wxDialogBox

Change the name to wxDialog, and for modal dialogs, use ShowModal instead of Show.

10.15 wxDialog::Show

If you used **Show** to show a modal dialog or to override the standard modal dialog **Show**, use **ShowModal** instead.

See also

Dialogs and controls (p. 7)

10.16 wxForm

Sorry, this class is no longer available. Try using the wxPropertyListView or wxPropertyFormView class instead, or use .wxr files and validators.

10.17 wxPoint

The old wxPoint is called wxRealPoint, and wxPoint now uses integers.

10.18 wxRectangle

This is now called wxRect.

10.19 wxScrollBar

The function names have changed for this class: please refer to the documentation for wxScrollBar. Instead of setting properties individually, you will call SetScrollbar with several parameters.

10.20 wxText, wxMultiText, wxTextWindow

Change all these to wxTextCtrl. Add the window style wxTE_MULTILINE if you wish to have a multi-line text control.

10.21 wxToolBar

This name is an alias for the most popular form of toolbar for your platform. There is now a family of toolbar classes, with for example wxToolBar95, wxToolBarMSW and wxToolBarSimple classes existing under Windows 95.

Toolbar management is supported by frames, so calling wxFrame::CreateToolBar and adding tools is usually enough, and the SDI or MDI frame will manage the positioning for you. The client area of the frame is the space left over when the menu bar, toolbar and status bar have been taken into account.