Programming in GNOME with C++

Paulo Pinto

19 May 2002

As published in C/C++ Users Journal, 2002 May issue

1 Introduction

When choosing a programming language in which to develop an application for the GNOME enviroment, C is the natural choice. This happens because C is the language used to develop GNOME. So what options are available to C++ programmers? One of them is to create your own framework, wrapping the C API that you need for your application. But it would be better if such framework already existed. In fact, it does, and it is accepted as the official C++ binding for GNOME. It is known as Gnomemm and will be the subject of this article.

2 Overview

As with any other framework, Gnomemm is composed of several parts, shown in Figure 1. The main parts are Gtkmm which wraps Gtk+ funcionality, Gnomemm for the GUI part of GNOME, and libsigc++ for signal/slot handling. Table 1 shows all the major components and their funcionality.

Gtkmm	Wraps all the Gtk+ funcionality
Gnomemm	Wraps the Gnome UI funcionality
libsigc++	Handles signal/slot invocations and declarations
Bakery	Document/View framework for Gnome– with XML support using Xerces
Glade–, Gladecc	Plugins for Glade so that it outputs Gtk–/Gnome– code for the UI
GConf–	Wraps the GConf functionality for Gnome– applications
GtkExtra–	Wraps the GtkExtra widgets for Gtk– applications
GtkGLArea–	Wraps the GtkGLArea widget for Gtk– applications

Table 1: Gnomemm components and funcionality

To build Gnomemm applications you need to have at least three components:

- libsigc++;
- Gtkmm;
- Gnomemm

Gtkmm/Gnomemm[1] make heavy use on the abstraction features provided by C++, such as namespaces, templates and STL. This means that you need to have a C++ compiler that supports ANSI C++ as closely as possible. For g++ this means at least version 2.95.2.

3 Advantages

Experienced C++ developers will be able to anticipate the advantages of wrapping a C API like Gtk+ with an OO framework. But, just in case, here's a list:

- Clearer API:
  - better cohesion between the functions and data structures;
  - type-specific parameters. e.g. no mysterious void* struct members, which supress errors caused by bad parameters;
  - No need to use several arguments just to pass state around, like handles.

[1] Gtkmm/Gnomemm were previously known as Gtk–/Gnome–

- Less code:
  - No need for repetitive cast macros, like GTK_WIDGET. C++ type system will handle conversions between widgets;
- Increased type-safety:
  - more code problems found at compile time.
- Incredibly simple derivation of new widgets, using C++ inheritance:
  - Derivation of new widgets is complex and error-prone in normal GTK+, because it forces the user to create several structures with a specific layout and with complex initializations.
- Type-safe callbacks:
  - The compiler will tell you if your callback (signal handler) does not match the signal. In Gtk+/GNOME it is possible to associate a callback to a signal with the wrong signature.
- More organised code:
  - The simpler code and ease of derivation remove obstacles to code refactoring.
- Simpler memory management.
  - No worrying about when a char* or GList* should be deallocated. Let the constructors and destructors of the widgets handle the allocation/deallocation of data.

4 Creating a window

It's time to show how a Gtkmm application looks. So lets start by creating a window with "Hello World" as the caption, using the Gtk+ part of the framework.

Start by including common headers:

```
#include <gtk--/main.h>
#include <gtk--/window.h>
```

Initialize the framework and create the window:

```
int main (int argc, char *argv [])
{
  Gtk::Main kit (argc, argv);
  Gtk::Window win;
```

Set the window caption to "Hello World":

```
  win.set_title ("Hello World");
```

The destroy signal must be handled or the user won't be able to close the window by clicking on the 'X' button:

```
  win.destroy.connect (Gtk::Main::quit.slot ());
```

Show the window and begin the event processing loop:

```
  win.show ();

  kit.run ();
}
```

Listing A shows the complete source code for the program.

To compile the application you will need to use the following command

```
g++ -o hello hello.cpp ‘gtkmm-config --cflags --libs‘ -lgnomemm
```

For those who are curious about gtkmm-config, it is a shell script that gives the compiler the correct flags for the platform where it is run.

When running the application, a window like the one in figure 2 will appear, which you can close by clicking in the X part of the window.

5 What are signals and slots ?

Many readers may not know what are signals and slots if they haven’t looked to Gtk+, or even QT before. In GUI toolkits like Gtk+, the events that are sent to the application are known as signals. The callbacks associated with those signals are called slots.

Every signal can have multiple slots, and a slot can respond to several different signals. In Gtk+ slots are functions. In Gtkmm/Gnomemm they can be simple functions or member functions. In the case of member functions, they can be virtual or static.

6 A better wc

The hello world program is fine for a first example, but now it’s time to do something more substantial. Let’s next create a GNOME version of the popular Unix command line program wc [2]. This program will allow the user to select a file and then display the lines, words and bytes count of the file, just like wc. In figure 3 you can see gwc after it is launched. In figure 4, gwc is displaying the results of having processed a file.

In Gnomemm all GNOME applications inherit from Gnome::App, and so will gwc. The gwc window will be composed of a menu, a toolbar and a text label for displaying results. These items will be created in the constructor of the application window, as shown below.

First the label is created and used as the contents for the application window:

```
m_text = new Gtk::Label ("No file counted");
set_contents (*m_text);
```

The created label will have to be explicitly deleted. Compare this with the use of the manage() call, described latter on.

[2]For those who don’t know what wc is, it is a very common program in Unix, that counts the number of word, lines and bytes in a file

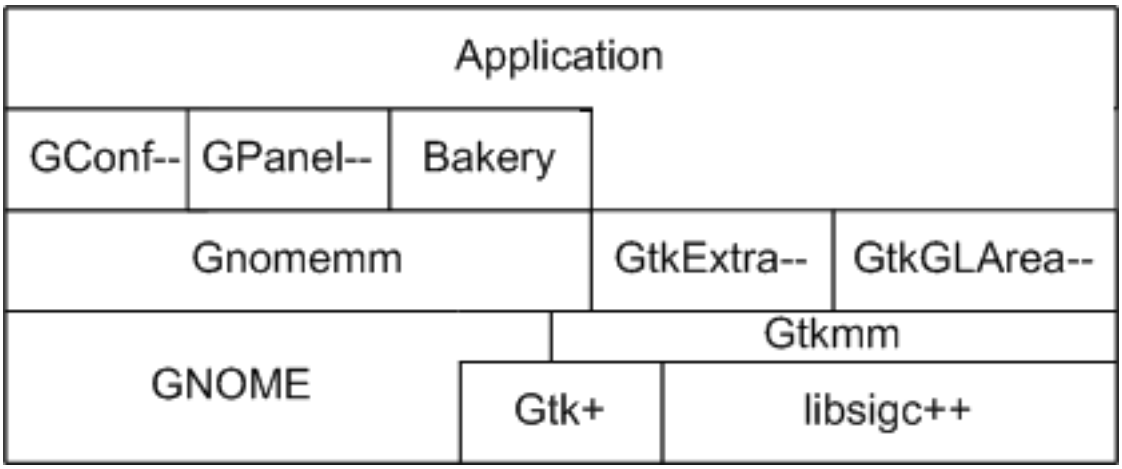

Figure 1: Gnomemm framework

Figure 2: Hello World window

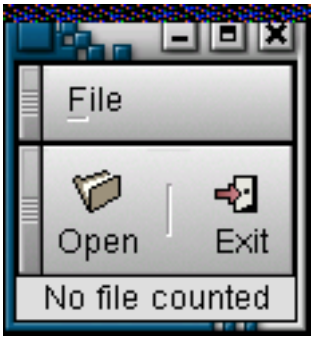

Figure 3: gwc application

Then a file selection dialog is created so that whenever gwc needs to ask the user for a file, it is already there. Note that this is just an application design choice. You could alternatively choose to create the dialog every time the user tries to open a file.

```
// Allocate the dialog and let the library
// handle the memory used by it
m_filesel = manage (new Gtk::FileSelection ("Open File"));

// Connect the ok_button to file_ok_sel function
m_filesel->get_ok_button()->clicked.connect(slot(this,
                                  &GWCApp::process_file));

// Connect the cancel_button to hide the window
m_filesel->get_cancel_button()->clicked.connect(m_filesel->hide.slot());
```

There are some things to be emphasized here. The first thing to note is that a slot must be conected to the signal that is triggered when the user presses the OK button. This is the best way to process the information from the dialog. The other solution would be to set a flag so that in the code that calls the dialog knows that the user has selected a file.

The second noteworthy point has to do with the manage () call. In Gtkmm/Gnomemm you can create widgets on the stack, inside class objects, or on the heap. If you created them on the heap, you are expected to delete the widgets unless you pass the ownership of the widget to the Gtkmm/Gnomemm library, by calling manage(). In that case it will be the library that will delete the widget when it will be safe to do so. There are some subleties in using manage (), so some care must be taken in its use. Just don't use it like it's the solution for all your memory problems. In fact, just use it for widgets that are added to a containers[3] and not for top-level windows, or dialogs for that matter. Think of it as letting the widget be 'managed' by the container that owns it.

Next, how are the menus constructed ? You can use the wrapped Gtk+/GNOME APIs, the native Gtk+/GNOME APIs, or the menu helpers. Our sample gwc program will use the last solution because it is simple to understand, and to me it's also the most natural choice. Helpers are special objects that manage the creation of widgets that must be inserted into Gtk::Container widgets. Their use makes the task of building multi-item widgets like menus less error-prone.

The "File" menu that gwc has is built like this:

```
UI::Info file_menu[] = { MenuItems::Open (
                                  slot (this,
                                        &GWCApp::on_menu_open)),
                         UI::Separator(),
                         MenuItems::Exit(Gtk::Main::quit.slot ())
                        };
```

The first and third array entries create standard menu options and specify which slots will be invoked when the user selects that option. As for the second entry, that's a normal horizontal line that's commonly used to separate different sets of menu options. But a menu will be useless if it isn't attached to a menu bar, so let's also declare the menu bar:

```
UI::SubTree menus[] = { Menus::File(file_menu) };
```

Now that we have a menu bar, the application must generate the menu layout from it, and attach the result to the application window.

[3]Widgets that can own other widgets. For example a window is a container for any kind of widget that can be placed inside it.

```
create_menus(menus);
```

I must emphasize that although the menu helpers are based on templates, they are quite easy to use.

You may have noted that the code listing doesn't connect a callback to the delete event, which would seem to mean that the users won't be able to close the gwc window. Well that isn't quite true, because the user will be able to close it.

How? If you take a closer look to listing B you will see a declaration for delete_event_impl (), and that method is responsible for terminating the application. All widgets have virtual member functions for their signals, with the same name as the signal and a _impl suffix. So it is possible to process signals without having to connect the signal to a slot. That was what happened to the delete event. This way it is easier to alter the behaviour of some signal handlers.

Finally, no modern GUI application would be complete without a toolbar, so gwc also ought to have one. The process of creating toolbars is the same as for menus, so a vector is declared with an item for each toolbar buttom, and an icon for that button[4]. After that, a name for the button is added, the slot to call when the user presses the button, and the text for the tooltip. Simple, isn't it ?

```
UI::Info toolbar[] = {UI::Item(UI::Icon(GNOME_STOCK_PIXMAP_OPEN),
                               "Open", slot(this, &GWCApp::on_menu_open),
                               "Open a file"),
                      UI::Separator (),
                      UI::Item(UI::Icon(GNOME_STOCK_PIXMAP_EXIT),
                               "Exit", Gtk::Main::quit.slot (),
                               "Exit the application")
                     };
```

After initializing the toolbar vector, it must be added to the application window with a:

```
create_toolbar(toolbar);
```

In this case we just used a built-in array of UI::Info, but create_toolbar() accepts other types. For example, std::vector<> or std::list<> with an UI::Info argument can also be used. It all depends on the needs of your application.

7 Current Gtkmm/Gnomemm state

At this time, Gtkmm wraps all of the widgets provided by the stable Gtk+ 1.2.x. Support for the upcomming Gtk+ 2.0, with Pango and GObject, is now available in CVS as of version 1.3.x. Gnomemm wraps all the gnome-libs 1.2.x widgets. Unfortunally Gnomemm still lacks support for Bonobo components, the XML handling framework [5], the audio handling and some other things. But don't be demoralized by this, after all the Gtkmm/Gnomemm are already very usable and you can always wrap the functionality that you need. And who knows, you could even give that work back to the community and be helping Gtkmm in doing so.

[4] In Gtk+/GNOME the icons in the toolbars are optional

[5] But you could use any other C++ XML parser, such as Xerces from xml.apache.org. Bakery uses Xerces-C++ to provide a Document/View for Gnome– which uses XML

8 Conclusion

This article is intended to provide a very quick overview about Gtkmm, and to show that C++ programmers are supported and welcome in GNOME. Of course many things are still lacking and if you can help, the Gtkmm team will surely apreciate your help.

I hope that you will give Gtkmm a try in you next project.

9 Acknowledgements

I wish to thank to Murray Cumming, one of the Gtkmm developers, for his input to this document. I also dedicate this article to the memory to my grandfathers.

10 References

Gtkmm - http://gtkmm.sourceforge.net/

Gtkmm mailing list - http://gtkmm.sourceforge.net/mailinglist.html

Gtkmm FAQ - http://gtkmm.sourceforge.net/docs/gtkmm-faq.html

libsigc++ - http://libsigc.sourceforge.net/

Bakery - http://bakery.sourceforge.net

Glade– - http://home.wtal.de/petig/Gtk/

Gladecc - http://www.geocities.com/SiliconValley/Bit/8083/gladecc.htm

GConf– - http://gconfmm.sourceforge.net/

GtkExtra– - http://gtkextramm.sourceforge.net/

GtkGLArea– - http://www.ece.ucdavis.edu/ kenelson/gtkglareamm/

Gtk+ - http://www.gtk.org

GNOME - http://www.gnome.org

A Listing 1

```
/*
 * Paulo Pinto (pjmlp_pt@yahoo.com) (c) 2001
 * A very simple hello world to
 * show basic Gtk-- usage.
 */
#include <gtk--/main.h>
#include <gtk--/window.h>

// Please note that in ANSI C++ the
// return is optional
int main (int argc, char *argv [])
{
  // Initialize the framework
  Gtk::Main kit (argc, argv);
  Gtk::Window win;

  // Set the main window title
  win.set_title ("Hello World");

  // Connect the signal that is generated when the
  // user tries to close the window, to the
  // quit slot. This way the user will be able to
  // close the window by clicking on the 'X' mark.
  win.destroy.connect (Gtk::Main::quit.slot ());

  // Make the window appear
  win.show ();

  // Enter into the event loop
  kit.run ();
}
```

B Listing 2

```
/*
 * Paulo Pinto (pjmlp_pt@yahoo.com) (c) 2001
 * An GUI version of the wc program
 */
#include <gtk--/label.h>
#include <gtk--/fileselection.h>
#include <gnome--/main.h>
#include <gnome--/app.h>
#include <gnome--/stock.h>

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cctype>

//Bring the slot binding command to global scope
using SigC::slot;

// Application
// A minimalist graphical view of wc
class GWCApp: public Gnome::App {
public:
  GWCApp ();
  virtual ~GWCApp ();

protected:
  gint delete_event_impl (GdkEventAny*);

  //Signal handlers:
  virtual void on_menu_open ();

  void process_file ();

private:
  bool count_words (const std::string &filename, int &words, int &lines, int &bytes);
  void init_menus ();
  void init_toolbar ();

  Gtk::Label *m_text;
  Gtk::FileSelection *m_filesel;
};

// The program entry point.
// Just create a GWCApp instance and
// launch it.
// Please note that in ANSI C++ the
// return is optional
int main (int argc, char *argv [])
{
  // Initialize the framework
  Gnome::Main kit ("gwc", "0.0.1", argc, argv);
  GWCApp app;

  // Make the window appear
  app.show ();

  // Enter into the event loop
  kit.run ();
}
```

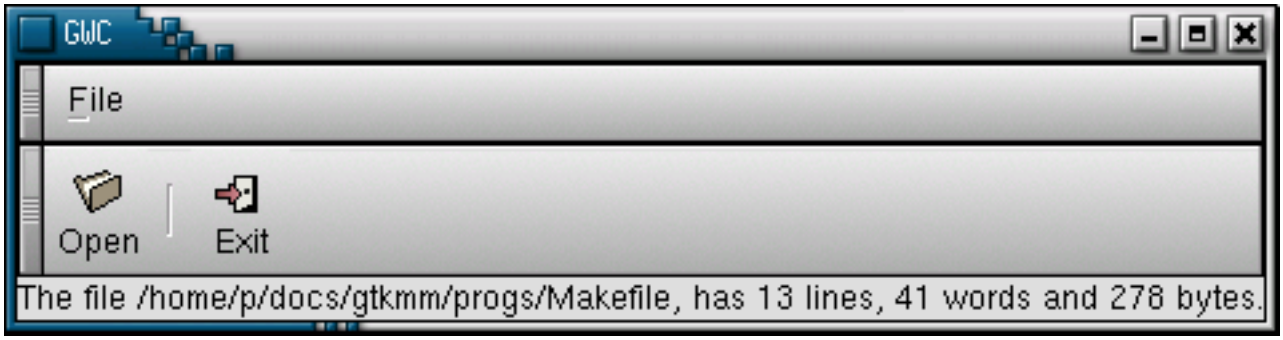

Figure 4: gwc after processing a file