An introduction to programming with GTK+ and Glade in ISO C, ISO C++ and Python

Version 1.3.1

Roger Leigh
`rleigh@debian.org`

3rd January 2010

Contents

List of Figures

Listings

1 Introduction

1.1 What is GTK+?

GTK+ is a *toolkit* used for writing graphical applications. Originally written for the X11 windowing system, it has now been ported to other systems, such as Microsoft Windows and the Apple Macintosh, and so may be used for cross-platform software development. GTK+ was written as a part of the *GNU Image Manipulation Program* (GIMP), but has long been a separate project, used by many other free software projects, one of the most notable being the *GNU Network Object Model Environment* (GNOME) Project.

GTK+ is written in C and, because of the ubiquity of the C language, *bindings* have been written to allow the development of GTK+ applications in many other languages. This short tutorial is intended as a simple introduction to writing GTK+ applications in C, C++ and Python, using the current (2.6) version of `libgtk`. It also covers the use of the Glade user interface designer for *rapid application development* (RAD).

It is assumed that the reader is familiar with C and C++ programming, and it would be helpful to work through the "Getting Started" chapter of the GTK+ tutorial before reading further. The GTK+, GLib, libglade, Gtkmm and libglademm API references will be useful while working through the examples. Very little Python knowledge is required, but the Python tutorial and manual, and the PyGTK and Glade API references, will also be useful.

I hope you find this tutorial informative. Please send any corrections or suggestions to `rleigh@debian.org`.

1.2 Building the example code

Several working, commented examples accompany the tutorial. They are also available from `http://people.debian.org/~rleigh/gtk/ogcalc/`. To build them, type:

```
./configure
make
```

This will check for the required libraries and build the example code. Each program may then be run from within its subdirectory.

I have been asked on various occasions to write a tutorial to explain how the GNU autotools work. While this is not the aim of this tutorial, I have converted the build to use the autotools as a simple example of their use.

1.3 Legal bit

2 GTK+ basics

2.1 Objects

GTK+ is an *object-oriented* (OO) toolkit. I'm afraid that unless one is aware of the basic OO concepts (classes, class methods, inheritance, polymorphism), this tutorial (and GTK+ in general) will seem rather confusing. On my first attempt at learning GTK+, I didn't 'get' it, but after I learnt C++, the concepts GTK+ is built on just 'clicked' and I understood it quite quickly.

The C language does not natively support classes, and so GTK+ provides its own object/type system, **GObject**. GObject provides objects, inheritance, polymorphism, constructors, destructors and other facilities such as reference counting and signal emission and handling. Essentially, it provides C++ classes in C. The syntax differs a little from C++ though. As an example, the following C++

```
myclass c;
c.add(2);
```

would be written like this using GObject:

```
myclass *c = myclass_new();
myclass_add(c, 2);
```

The difference is due to the lack of a *this* pointer in the C language (since objects do not exist). This means that class methods require the object pointer passing as their first argument. This happens automatically in C++, but it needs doing 'manually' in C.

Another difference is seen when dealing with polymorphic objects. All GTK+ widgets (the controls, such as buttons, checkboxes, labels, etc.) are derived from `GtkWidget`. That is to say, a `GtkButton` *is a* `GtkWidget`, which *is a* `GtkObject`, which *is a* `GObject`. In C++, one can call member functions from both the class and the classes it is derived from. With GTK+, the object needs explicit casting to the required type. For example

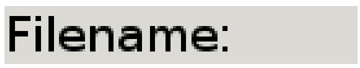
```
GtkButton mybutton;
mybutton.set_label("Cancel");
mybutton.show();
```

would be written as

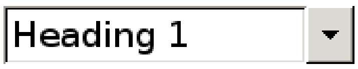
```
GtkButton *mybutton = gtk_button_new();
gtk_button_set_label(mybutton, "Cancel");
gtk_widget_show(GTK_WIDGET(mybutton))
```

In this example, `set_label()` is a method of `GtkButton`, whilst `show()` is a method of `GtkWidget`, which requires an explicit cast. The `GTK_WIDGET()` cast is actually a form of *run-time type identification* (RTTI). This ensures that the objects are of the correct type when they are used.

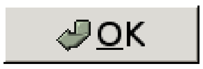
Objects and C work well, but there are some issues, such as a lack of type-safety of callbacks and limited compile-time type checking. Using GObject, deriving new widgets is complex and error-prone. For these, and other, reasons, C++ may be a better language to use. `libsigc++` provides type-safe signal handling, and all of the GTK+ (and GLib, Pango et. al.) objects are available as standard C++ classes. Callbacks may also be class methods, which makes for



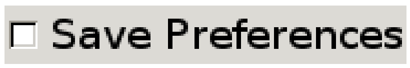
(a) A text label



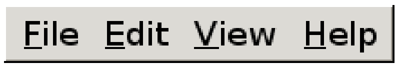
(b) A drop-down selection (combo box)



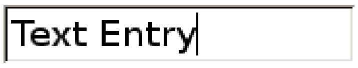
(c) A push button



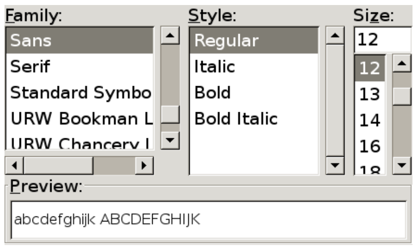
(d) A tick box



(e) A menu bar



(f) A text entry field



(g) A font selection

Figure 1: A selection of GTK+ widgets.

cleaner code since the class can contain object data, removing the need to pass in data as a function argument. These potential problems will become clearer in the next sections.

2.2 Widgets

A user interface consists of different objects with which the user can interact. These include buttons which can be pushed, text entry fields, tick boxes, labels and more complex things such as menus, lists, multiple selections, colour and font pickers. Some example widgets are shown in Figure 1.

Not all widgets are interactive. For example, the user cannot usually interact with a label, or a framebox. Some widgets, such as containers, boxes and event boxes are not even visible to the user (there is more about this in Section 2.3).

Different types of widget have their own unique *properties*. For example, a label widget contains the text it displays, and there are functions to get and set the label text. A checkbox may be ticked or not, and there are functions to get and set its state. An options menu has functions to set the valid options, and get the option the user has chosen.

2.3 Containers

The top-level of every GTK+ interface is the *window*. A window is what one might expect it to be: it has a title bar, borders (which may allow resizing), and it contains the rest of the interface.

In GTK+, a `GtkWindow` *is a* `GtkContainer`. In English, this means that the window is a widget that can contain another widget. More precisely, a `GtkContainer` can contain exactly **one** widget. This is usually quite confusing compared with the behaviour of other graphics toolkits, which allow one to place the controls on some sort of "form".

The fact that a `GtkWindow` can only contain one widget initially seems quite useless. After all, user interfaces usually consist of more than a single button. In GTK+, there are other kinds of `GtkContainer`. The most commonly used are horizontal boxes, vertical boxes, and tables. The structure of these containers is shown in Figure 2.

Figure 2 shows the containers as having equal size, but in a real interface, the containers resize themselves to fit the widgets they contain. In other cases, widgets may be expanded or shrunk to fit the space allotted to them. There are several ways to control this behaviour, to give fine control over the appearance of the interface.

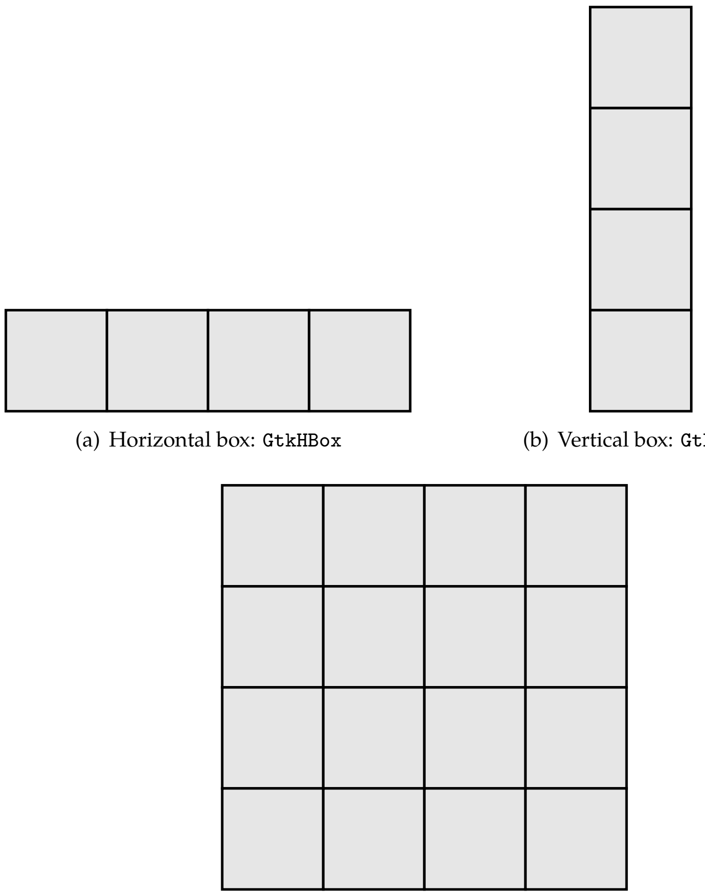
In addition to the containers discussed above, there are more complex containers available, such are horizontal and vertical panes, tabbed notebooks, and viewports and scrolled windows. These are out of the scope of this tutorial, however.

Newcomers to GTK+ may find the concept of containers quite strange. Users of Microsoft Visual Basic or Visual C++ may be used to the free-form placement of controls. The placement of controls at fixed positions on a form has *no* advantages over automatic positioning and sizing. All decent modern toolkits use automatic positioning. This fixes several issues with fixed layouts:

- The hours spent laying out forms, particularly when maintaining existing code.
- Windows that are too big for the screen.
- Windows that are too small for the form they contain.
- Issues with spacing when accommodating translated text.
- Bad things happen when changing the font size from the default.

The nesting of containers results in a *widget tree*, which has many useful properties, some of which will be used later. One important advantage is that they can dynamically resize and accommodate different lengths of text, important for internationalisation when translations in different languages may vary widely in their size.

The Glade user interface designer can be very instructive when exploring how containers and widget packing work. It allows easy manipulation of the interface, and all of the standard GTK+ widgets are available. Modifying an existing interface is trivial, even when doing major reworking. Whole branches of the widget tree may be cut, copied and pasted at will, and a widget's properties may be manipulated using the "Properties" dialogue. While studying the code examples, Glade may be used to interactively build and manipulate the interface, to visually follow how the code is working. More detail about Glade is provided in Section 5, where `libglade` is used to dynamically load a user interface.



(a) Horizontal box: `GtkHBox`

(b) Vertical box: `GtkVBox`

(c) Table: `GtkTable`

Figure 2: GTK+ containers. Each container may contain other widgets in the shaded areas. Containers may contain more containers, allowing them to nest. Complex interfaces may be constructed by nesting the different types of container.

2.4 Signals

Most graphical toolkits are *event-driven*, and GTK+ is no exception. Traditional console applications tend not to be event-driven; these programs follow a fixed path of execution. A typical program might do something along these lines:

- Prompt the user for some input
- Do some work
- Print the results

This type of program does not give the user any freedom to do things in a different order. Each of the above steps might be a single function (each of which might be split into helper functions, and so on).

GTK+ applications differ from this model. The programs must react to *events*, such as the user clicking on a button, or pressing Enter in an text entry field. These widgets emit signals in response to user actions. For each signal of interest, a function defined by the programmer is called. In these functions, the programmer can do whatever needed. For example, in the `ogcalc` program, when the "Calculate" button is pressed, a function is called to read the data from entry fields, do some calculations, and then display the results.

Each event causes a *signal* to be *emitted* from the widget handling the event. The signals are sent to *signal handlers*. A signal handler is a function which is called when the signal is emitted. The signal handler is *connected* to the signal. In C, these functions are known as *callbacks*. The process is illustrated graphically in Figure 3.

A signal may have zero, one or many signal handlers connected (registered) with it. If there is more than one signal handler, they are called in the order they were connected in.

Without signals, the user interface would display on the screen, but would not actually *do* anything. By associating signal handlers with signals one is interested in, events triggered by the user interacting with the widgets will cause things to happen.

2.5 Libraries

GTK+ is comprised of several separate libraries:

`atk` Accessibility Toolkit, to enable use by disabled people.

`gdk` GIMP Drawing Kit (XLib abstraction layer—windowing system dependent part).

`gdk-pixbuf` Image loading and display.

`glib` Basic datatypes and common algorithms.

`gmodule` Dynamic module loader (`libdl` portability wrapper).

`gobject` Object/type system.

`gtk` GIMP Tool Kit (windowing system independent part).

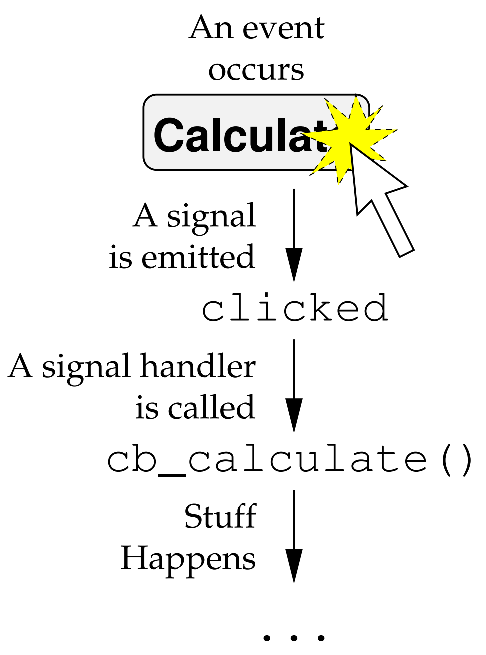


Figure 3: A typical signal handler. When the button is pressed, a signal is emitted, causing the registered callback function to be called.

`pango` Typeface layout and rendering.

When using `libglade` another library is required:

`glade` User Interface description loader/constructor.

Lastly, when using C++, some additional C++ libraries are also needed:

`atkmm` C++ ATK wrapper.

`gdkmm` C++ GDK wrapper.

`gtkmm` C++ GTK+ wrapper.

`glademm` C++ Glade wrapper.

`pangomm` C++ Pango wrapper.

`sigc++` Advanced C++ signalling & event handling (wraps GObject signals).

This looks quite intimidating! However, there is no need to worry, since compiling and linking programs is quite easy. Since the libraries are released together as a set, there are few library interdependency issues.

3 Designing an application

3.1 Planning ahead

Before starting to code, it is necessary to plan ahead by thinking about what the program will do, and how it should do it. When designing a graphical interface, one should pay attention to *how* the user will interact with it, to ensure that it is both easy to understand and efficient to use.

When designing a GTK+ application, it is useful to sketch the interface on paper, before constructing it. Interface designers such as Glade are helpful here, but a pen and paper are best for the initial design.

3.2 Introducing `ogcalc`

As part of the production (and quality control) processes in the brewing industry, it is necessary to determine the alcohol content of each batch at several stages during the brewing process. This is calculated using the density (gravity) in $^{g}/_{cm^3}$ and the refractive index. A correction factor is used to align the calculated value with that determined by distillation, which is the standard required by HM Customs & Excise. Because alcoholic beverages are only slightly denser than water, the PG value is the $(\text{density} - 1) \times 10000$. That is, 1.0052 would be entered as 52.

Original gravity is the density during fermentation. As alcohol is produced during fermentation, the density falls. Traditionally, this would be similar to the PG, but with modern high-gravity brewing (at a higher concentration) it tends to be higher. It is just as important that the OG is within the set limits of the specification for the product as the ABV.

The `ogcalc` program performs the following calculation:

$$O = (R \times 2.597) - (P \times 1.644) - 34.4165 + C \tag{1}$$

If O is less than 60, then

$$A = (O - P) \times 0.130 \tag{2}$$

otherwise

$$A = (O - P) \times 0.134 \tag{3}$$

The symbols have the following meanings:

A	Percentage Alcohol By Volume
C	Correction Factor
O	Original Gravity
P	Present Gravity
R	Refractive Index

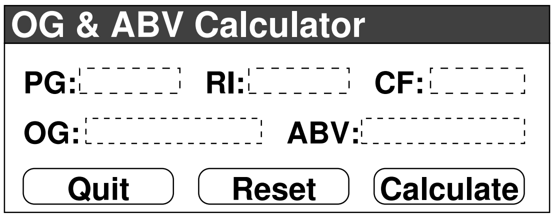


Figure 4: Sketching a user interface. The `ogcalc` main window is drawn simply, to illustrate its functionality. The top row contains three numeric entry fields, followed by two result fields on the middle row. The bottom row contains buttons to quit the program, reset the interface and do the calculation.

3.3 Designing the interface

The program needs to ask the user for the values of C, P, and R. It must then display the results, A and O.

A simple sketch of the interface is shown in Figure 4.

3.4 Creating the interface

Due to the need to build up an interface from the bottom up, due to the containers being nested, the interface is constructed starting with the window, then the containers that fit in it. The widgets the user will use go in last. This is illustrated in Figure 5.

Once a widget has been created, signal handlers may be connected to its signals. After this is completed, the interface can be displayed, and the main *event loop* may be entered. The event loop receives events from the keyboard, mouse and other sources, and causes the widgets to emit signals. To end the program, the event loop must first be left.

4 GTK+ and C

4.1 Introduction

Many GTK+ applications are written in C alone. This section demonstrates the `C/plain/ogcalc` program discussed in the previous section. Figure 6 is a screenshot of the finished application.

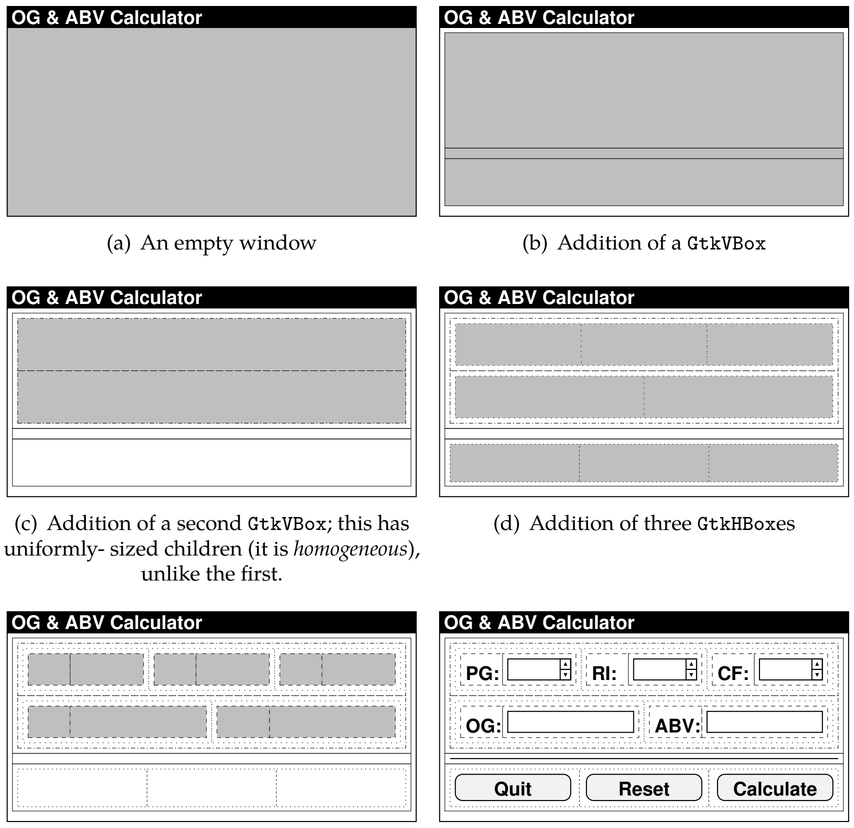
This program consists of five functions:

`on_button_clicked_reset()` Reset the interface to its default state.

`on_button_clicked_calculate()` Get the values the user has entered, do a calculation, then display the results.

`main()` Initialise GTK+, construct the interface, connect the signal handlers, then enter the GTK+ event loop.

`create_spin_entry()` A helper function to create a numeric entry with descriptive label and tooltip, used when constructing the interface.



(a) An empty window

(b) Addition of a `GtkVBox`

(c) Addition of a second `GtkVBox`; this has uniformly- sized children (it is *homogeneous*), unlike the first.

(d) Addition of three `GtkHBoxes`

(e) Addition of five more `GtkHBoxes`, used to ensure visually appealing widget placement

(f) Addition of all of the user-visible widgets

Figure 5: Widget packing. The steps taken during the creation of an interface are shown, demonstrating the use of nested containers to pack widgets.

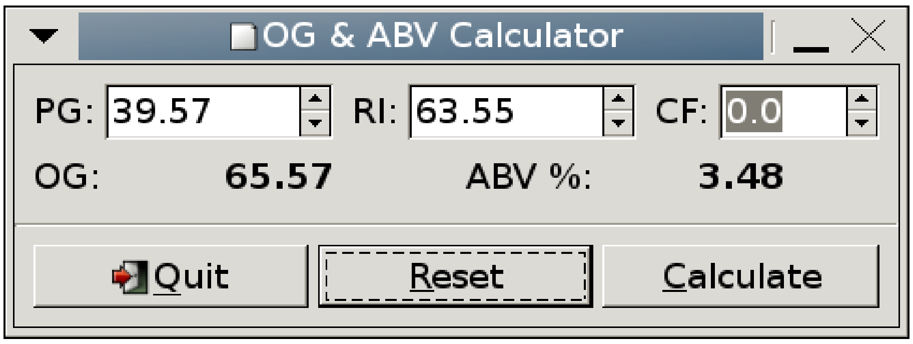


Figure 6: `C/plain/ogcalc` in action.

`create_result_label()` A helper function to create a result label with discriptive label and tooltip, used when constructing the interface.

4.2 Code listing

The program code is listed below. The source code is extensively commented, to explain what is going on.

Listing 1: `C/plain/ogcalc.c`

```c
#include <gtk/gtk.h>

GtkWidget *
create_spin_entry( const gchar    *label_text,
                   const gchar    *tooltip_text,
                   GtkWidget     **spinbutton_pointer,
                   GtkAdjustment  *adjustment,
                   guint           digits );
GtkWidget *
create_result_label(const gchar   *label_text,
                    const gchar   *tooltip_text,
                    GtkWidget    **result_label_pointer );
void on_button_clicked_reset( GtkWidget *widget,
                              gpointer   data );
void on_button_clicked_calculate( GtkWidget *widget,
                                  gpointer   data );

/* This structure holds all of the widgets needed to get all
   the values for the calculation. */
struct calculation_widgets
{
  GtkWidget *pg_val;     /* PG entry widget */
  GtkWidget *ri_val;     /* RI entry widget */
  GtkWidget *cf_val;     /* CF entry widget */
  GtkWidget *og_result;  /* OG result label */
  GtkWidget *abv_result; /* ABV% result label */
};

/* The bulk of the program.  This is nearly all setting up
   of the user interface.  If Glade and libglade were used,
   this would be under 10 lines only! */
int main(int argc, char *argv[])
{
  /* These are pointers to widgets used in constructing the
     interface, and later used by signal handlers. */
  GtkWidget                  *window;
  GtkWidget                  *vbox1,   *vbox2;
  GtkWidget                  *hbox1,   *hbox2;
  GtkWidget                  *quit, *reset, *calculate;
  GtkObject                  *adjustment;
  GtkWidget                  *hsep;
  struct calculation_widgets  cb_widgets;

  /* Initialise GTK+. */
  gtk_init(&argc, &argv);

  /* Create a new top-level window. */
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  /* Set the window title. */
  gtk_window_set_title (GTK_WINDOW(window),
                        "OG & ABV Calculator");
  /* Disable window resizing, since there's no point in this
     case. */
  gtk_window_set_resizable(GTK_WINDOW(window), FALSE);
  /* Connect the window close button ("destroy" event) to
     gtk_main_quit(). */
  g_signal_connect (G_OBJECT(window),
                    "destroy",
                    gtk_main_quit, NULL);

  /* Create a GtkVBox to hold the other widgets. This
     contains other widgets, which are packed in to it
     vertically. */
  vbox1 = gtk_vbox_new (FALSE, 0);
  /* Add the VBox to the Window. A GtkWindow /is a/
     GtkContainer which /is a/ GtkWidget. GTK_CONTAINER
     casts the GtkWidget to a GtkContainer, like a C++
     dynamic_cast. */
  gtk_container_add (GTK_CONTAINER(window), vbox1);
  /* Display the VBox. At this point, the Window has not
     yet been displayed, so the window isn't yet visible. */
  gtk_widget_show(vbox1);

  /* Create a second GtkVBox. Unlike the previous VBox, the
     widgets it will contain will be of uniform size and
     separated by a 5 pixel gap. */
  vbox2 = gtk_vbox_new (TRUE, 5);
  /* Set a 10 pixel border width. */
  gtk_container_set_border_width(GTK_CONTAINER(vbox2), 10);
  /* Add this VBox to our first VBox. */
  gtk_box_pack_start (GTK_BOX(vbox1), vbox2,
                      FALSE, FALSE, 0);
  gtk_widget_show(vbox2);

  /* Create a GtkHBox. This is identical to a GtkVBox
     except that the widgets pack horizontally instead of
     vertically. */
  hbox1 = gtk_hbox_new (FALSE, 10);

  /* Add to vbox2. The function's other arguments mean to
     expand into any extra space alloted to it, to fill the
     extra space and to add 0 pixels of padding between it
     and its neighbour. */
  gtk_box_pack_start (GTK_BOX(vbox2), hbox1, TRUE, TRUE, 0);
  gtk_widget_show (hbox1);


  /* A GtkAdjustment is used to hold a numeric value: the
     initial value, minimum and maximum values, "step" and
```

```
       "page" increments and the "page size". It's used by
       spin buttons, scrollbars, sliders etc.. */
    adjustment = gtk_adjustment_new (0.0, 0.0, 10000.0,
                                     0.01, 1.0, 0);
    /* Call a helper function to create a GtkSpinButton entry
       together with a label and a tooltip. The spin button
       is stored in the cb_widgets.pg_val pointer for later
       use. We also specify the adjustment to use and the
       number of decimal places to allow. */
    hbox2 = create_spin_entry("PG:",
                              "Present Gravity (density)",
                              &cb_widgets.pg_val,
                              GTK_ADJUSTMENT (adjustment), 2);
    /* Pack the returned GtkHBox into the interface. */
    gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
    gtk_widget_show(hbox2);

    /* Repeat the above for the next spin button. */
    adjustment = gtk_adjustment_new (0.0, 0.0, 10000.0,
                                     0.01, 1.0, 0);
    hbox2 = create_spin_entry("RI:",
                              "Refractive Index",
                              &cb_widgets.ri_val,
                              GTK_ADJUSTMENT (adjustment), 2);
    gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
    gtk_widget_show(hbox2);

    /* Repeat again for the last spin button. */
    adjustment = gtk_adjustment_new (0.0, -50.0, 50.0,
                                     0.1, 1.0, 0);
    hbox2 = create_spin_entry("CF:",
                              "Correction Factor",
                              &cb_widgets.cf_val,
                              GTK_ADJUSTMENT (adjustment), 1);
    gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
    gtk_widget_show(hbox2);

    /* Now we move to the second "row" of the interface, used
       to display the results. */

    /* Firstly, a new GtkHBox to pack the labels into. */
    hbox1 = gtk_hbox_new (TRUE, 10);
    gtk_box_pack_start (GTK_BOX(vbox2), hbox1, TRUE, TRUE, 0);
    gtk_widget_show (hbox1);

    /* Create the OG result label, then pack and display. */
    hbox2 = create_result_label("OG:",
                                "Original Gravity (density)",
                                &cb_widgets.og_result);

    gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
    gtk_widget_show(hbox2);

    /* Repeat as above for the second result value. */
```

```
hbox2 = create_result_label("ABV %:",
                            "Percent Alcohol By Volume",
                            &cb_widgets.abv_result);
gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
gtk_widget_show(hbox2);

/* Create a horizontal separator (GtkHSeparator) and add
   it to the VBox. */
hsep = gtk_hseparator_new();
gtk_box_pack_start(GTK_BOX(vbox1), hsep, FALSE, FALSE, 0);
gtk_widget_show(hsep);

/* Create a GtkHBox to hold the bottom row of buttons. */
hbox1 = gtk_hbox_new(TRUE, 5);
gtk_container_set_border_width(GTK_CONTAINER(hbox1), 10);
gtk_box_pack_start(GTK_BOX(vbox1), hbox1, TRUE, TRUE, 0);
gtk_widget_show(hbox1);

/* Create the "Quit" button. We use a "stock"
   button--commonly-used buttons that have a set title and
   icon. */
quit = gtk_button_new_from_stock(GTK_STOCK_QUIT);
/* We connect the "clicked" signal to the gtk_main_quit()
   callback which will end the program. */
g_signal_connect (G_OBJECT (quit), "clicked",
                  gtk_main_quit, NULL);
gtk_box_pack_start(GTK_BOX(hbox1), quit,
                   TRUE, TRUE, 0);
gtk_widget_show(quit);

/* This button resets the interface. */
reset = gtk_button_new_with_mnemonic("_Reset");
/* The "clicked" signal is connected to the
   on_button_clicked_reset() callback above, and our
   "cb_widgets" widget list is passed as the second
   argument, cast to a gpointer (void *). */
g_signal_connect (G_OBJECT (reset), "clicked",
                  G_CALLBACK(on_button_clicked_reset),
                  (gpointer) &cb_widgets);
/* g_signal_connect_swapped is used to connect a signal
   from one widget to the handler of another. The last
   argument is the widget that will be passed as the first
   argument of the callback. This causes
   gtk_widget_grab_focus to switch the focus to the PG
   entry. */
g_signal_connect_swapped
  (G_OBJECT (reset),
   "clicked",
   G_CALLBACK (gtk_widget_grab_focus),
   (gpointer)GTK_WIDGET(cb_widgets.pg_val));
/* This lets the default action (Enter) activate this
   widget even when the focus is elsewhere. This doesn't
   set the default, it just makes it possible to set.*/
GTK_WIDGET_SET_FLAGS (reset, GTK_CAN_DEFAULT);
```

```
gtk_box_pack_start(GTK_BOX(hbox1), reset,
                   TRUE, TRUE, 0);
gtk_widget_show(reset);

/* The final button is the Calculate button. */
calculate = gtk_button_new_with_mnemonic("_Calculate");
/* When the button is clicked, call the
   on_button_clicked_calculate() function.  This is the
   same as for the Reset button. */
g_signal_connect (G_OBJECT (calculate), "clicked",
                  G_CALLBACK(on_button_clicked_calculate),
                  (gpointer) &cb_widgets);
/* Switch the focus to the Reset button when the button is
   clicked. */
g_signal_connect_swapped
  (G_OBJECT (calculate),
   "clicked",
   G_CALLBACK (gtk_widget_grab_focus),
   (gpointer)GTK_WIDGET(reset));
/* As before, the button can be the default. */
GTK_WIDGET_SET_FLAGS (calculate, GTK_CAN_DEFAULT);
gtk_box_pack_start(GTK_BOX(hbox1), calculate,
                   TRUE, TRUE, 0);
/* Make this button the default.  Note the thicker border
   in the interface--this button is activated if you press
   enter in the CF entry field. */
gtk_widget_grab_default (calculate);
gtk_widget_show(calculate);

/* Set up data entry focus movement.  This makes the
   interface work correctly with the keyboard, so that you
   can touch-type through the interface with no mouse
   usage or tabbing between the fields. */

/* When Enter is pressed in the PG entry box, focus is
   transferred to the RI entry. */
g_signal_connect_swapped
  (G_OBJECT (cb_widgets.pg_val),
   "activate",
   G_CALLBACK (gtk_widget_grab_focus),
   (gpointer) GTK_WIDGET(cb_widgets.ri_val));
/* RI -> CF. */
g_signal_connect_swapped
  (G_OBJECT (cb_widgets.ri_val),
   "activate",
   G_CALLBACK (gtk_widget_grab_focus),
   (gpointer) GTK_WIDGET(cb_widgets.cf_val));
/* When Enter is pressed in the RI field, it activates the
   Calculate button. */
g_signal_connect_swapped
  (G_OBJECT (cb_widgets.cf_val),
   "activate",
   G_CALLBACK (gtk_window_activate_default),
   (gpointer) GTK_WIDGET(window));

  /* The interface is complete, so finally we show the
     top-level window. This is done last or else the user
     might see the interface drawing itself during the short
     time it takes to construct. It's nicer this way. */
  gtk_widget_show (window);

  /* Enter the GTK Event Loop. This is where all the events
     are caught and handled. It is exited with
     gtk_main_quit(). */
  gtk_main();

  return 0;
}

/* A utility function for UI construction. It constructs
   part of the widget tree, then returns its root. */
GtkWidget *
create_spin_entry( const gchar    *label_text,
                   const gchar    *tooltip_text,
                   GtkWidget     **spinbutton_pointer,
                   GtkAdjustment  *adjustment,
                   guint           digits )
{
  GtkWidget   *hbox;
  GtkWidget   *eventbox;
  GtkWidget   *spinbutton;
  GtkWidget   *label;
  GtkTooltips *tooltip;

  /* A GtkHBox to pack the entry child widgets into. */
  hbox = gtk_hbox_new(FALSE, 5);

  /* An eventbox. This widget is just a container for
     widgets (like labels) that don't have an associated X
     window, and so can't receive X events. This is just
     used to we can add tooltips to each label. */
  eventbox = gtk_event_box_new();
  gtk_widget_show(eventbox);
  gtk_box_pack_start (GTK_BOX(hbox), eventbox,
                      FALSE, FALSE, 0);
  /* Create a label. */
  label = gtk_label_new(label_text);
  /* Add the label to the eventbox. */
  gtk_container_add(GTK_CONTAINER(eventbox), label);
  gtk_widget_show(label);

  /* Create a GtkSpinButton and associate it with the
     adjustment. It adds/substracts 0.5 when the spin
     buttons are used, and has digits accuracy. */
  spinbutton =
    gtk_spin_button_new (adjustment, 0.5, digits);
  /* Only numbers can be entered. */
  gtk_spin_button_set_numeric
```

```
    (GTK_SPIN_BUTTON(spinbutton), TRUE);
  gtk_box_pack_start(GTK_BOX(hbox), spinbutton,
                     TRUE, TRUE, 0);
  gtk_widget_show(spinbutton);

  /* Create a tooltip and add it to the EventBox previously
     created. */
  tooltip = gtk_tooltips_new();
  gtk_tooltips_set_tip(tooltip, eventbox,
                       tooltip_text, NULL);

  *spinbutton_pointer = spinbutton;
  return hbox;
}

/* A utility function for UI construction.  It constructs
   part of the widget tree, then returns its root. */
GtkWidget *
create_result_label(const gchar   *label_text,
                    const gchar   *tooltip_text,
                    GtkWidget    **result_label_pointer )
{
  GtkWidget   *hbox;
  GtkWidget   *eventbox;
  GtkWidget   *result_label;
  GtkWidget   *result_value;
  GtkTooltips *tooltip;

  /* A GtkHBox to pack the entry child widgets into. */
  hbox = gtk_hbox_new(FALSE, 5);

  /* As before, a label in an event box with a tooltip. */
  eventbox = gtk_event_box_new();
  gtk_widget_show(eventbox);
  gtk_box_pack_start (GTK_BOX(hbox), eventbox,
                      FALSE, FALSE, 0);
  result_label = gtk_label_new(label_text);
  gtk_container_add(GTK_CONTAINER(eventbox), result_label);
  gtk_widget_show(result_label);

  /* This is a label, used to display the OG result. */
  result_value = gtk_label_new (NULL);
  /* Because it's a result, it is set "selectable", to allow
     copy/paste of the result, but it's not modifiable. */
  gtk_label_set_selectable (GTK_LABEL(result_value), TRUE);
  gtk_box_pack_start (GTK_BOX(hbox), result_value,
                      TRUE, TRUE, 0);
  gtk_widget_show(result_value);

  /* Add the tooltip to the event box. */
  tooltip = gtk_tooltips_new();
  gtk_tooltips_set_tip(tooltip, eventbox,
                       tooltip_text, NULL);

  *result_label_pointer = result_value;
  return hbox;
}

/* This is a callback function.  It resets the values of the
   entry widgets, and clears the results.  "data" is the
   calculation_widgets structure, which needs casting back
   to its correct type from a gpointer (void *) type. */
void on_button_clicked_reset( GtkWidget *widget,
                              gpointer   data )
{
  /* Widgets to manipulate. */
  struct calculation_widgets *w;

  w = (struct calculation_widgets *) data;

  gtk_spin_button_set_value (GTK_SPIN_BUTTON(w->pg_val),
                             0.0);
  gtk_spin_button_set_value (GTK_SPIN_BUTTON(w->ri_val),
                             0.0);
  gtk_spin_button_set_value (GTK_SPIN_BUTTON(w->cf_val),
                             0.0);
  gtk_label_set_text (GTK_LABEL(w->og_result), "");
  gtk_label_set_text (GTK_LABEL(w->abv_result), "");
}

/* This callback does the actual calculation.  Its arguments
   are the same as for on_button_clicked_reset(). */
void on_button_clicked_calculate( GtkWidget *widget,
                                  gpointer   data )
{
  gdouble                      pg, ri, cf, og, abv;
  gchar                       *og_string;
  gchar                       *abv_string;
  struct calculation_widgets *w;

  w = (struct calculation_widgets *) data;

  /* Get the numerical values from the entry widgets. */
  pg = gtk_spin_button_get_value
    (GTK_SPIN_BUTTON(w->pg_val));
  ri = gtk_spin_button_get_value
    (GTK_SPIN_BUTTON(w->ri_val));
  cf = gtk_spin_button_get_value
    (GTK_SPIN_BUTTON(w->cf_val));

  /* Do the sums. */
  og = (ri * 2.597) - (pg * 1.644) - 34.4165 + cf;

  if (og < 60)
    abv = (og - pg) * 0.130;
  else
    abv = (og - pg) * 0.134;

  /* Display the results.  Note the <b></b> GMarkup tags to
     make it display in boldface. */
  og_string = g_strdup_printf ("<b>%0.2f</b>", og);
  abv_string = g_strdup_printf ("<b>%0.2f</b>", abv);

  gtk_label_set_markup (GTK_LABEL(w->og_result),
                        og_string);
  gtk_label_set_markup (GTK_LABEL(w->abv_result),
                        abv_string);

  g_free (og_string);
  g_free (abv_string);
}
```

To build the source, do the following:

```
cd C/plain
cc $(pkg-config --cflags gtk+-2.0) -c ogcalc.c
cc $(pkg-config --libs gtk+-2.0) -o ogcalc ogcalc.o
```

4.3 Analysis

The `main()` function is responsible for constructing the user interface, connecting the signals to the signal handlers, and then entering the main event loop. The more complex aspects of the function are discussed here.

```
g_signal_connect (G_OBJECT(window),
                  "destroy",
                  gtk_main_quit, NULL);
```

This code connects the "destroy" signal of *window* to the `gtk_main_quit()` function. This signal is emitted by the window when it is to be destroyed, for example when the "close" button on the titlebar is clicked). The result is that when the window is closed, the main event loop returns, and the program then exits.

```
vbox1 = gtk_vbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER(window), vbox1);
```

vbox1 is a `GtkVBox`. When constructed using `gtk_vbox_new()`, it is set to be non-homogeneous (`FALSE`), which allows the widgets contained within the `GtkVBox` to be of different sizes, and has zero pixels padding space between the container widgets it will contain. The homogeneity and padding space are different for the various `GtkBoxes` used, depending on the visual effect intended.

`gtk_container_add()` packs *vbox1* into the window (a `GtkWindow` object *is a* `GtkContainer`).

```
eventbox = gtk_event_box_new();
gtk_widget_show(eventbox);
gtk_box_pack_start (GTK_BOX(hbox2), eventbox,
                    FALSE, FALSE, 0);
```

Some widgets do not receive events from the windowing system, and hence cannot emit signals. Label widgets are one example of this. If this is required, for example in order to show a tooltip, they must be put into a `GtkEventBox`,

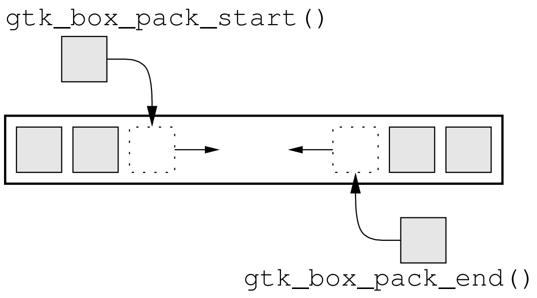


Figure 7: Packing widgets into a `GtkHBox`.

which can receive the events. The signals emitted from the `GtkEventBox` may then be connected to the appropriate handler.

`gtk_widget_show()` displays a widget. Widgets are hidden by default when created, and so must be shown before they can be used. It is typical to show the top-level window *last*, so that the user does not see the interface being drawn.

`gtk_box_pack_start()` packs a widget into a `GtkBox`, in a similar manner to `gtk_container_add()`. This packs *eventbox* into *hbox2*. The last three arguments control whether the child widget should expand into an extra space available, whether it should fill any extra space available (this has no effect if *expand* is `FALSE`), and extra space in pixels to put between its neighbours (or the edge of the box), respectively. Figure 7 shows how `gtk_box_pack_start()` works.

The `create_spin_entry()` function is a helper function to create a numeric entry (spin button) together with a label and tooltip. It is used to create all three entries.

```
label = gtk_label_new(label_text);
```

A new label is created displaying the text *label_text*.

```
spinbutton = gtk_spin_button_new (adjustment, 0.5, 2);
gtk_spin_button_set_numeric
  (GTK_SPIN_BUTTON(spinbutton), TRUE);
```

A `GtkSpinButton` is a numeric entry field. It has up and down buttons to "spin" the numeric value up and down. It is associated with a `GtkAdjustment`, which controls the range allowed, default value, etc.. `gtk_adjustment_new()` returns a new `GtkAdjustment` object. Its arguments are the default value, minimum value, maximum value, step increment, page increment and page size, respectively. This is straightforward, apart from the step and page increments and sizes. The step and page increments are the value that will be added or subtracted when the mouse button 1 or button 2 are clicked on the up or down buttons, respectively. The page size has no meaning in this context (`GtkAdjustments` are also used with scrollbars).

`gtk_spin_button_new()` creates a new `GtkSpinButton`, and associates it with *adjustment*. The second and third arguments set the "climb rate" (rate of change when the spin buttons are pressed) and the number of decimal places to display.

Finally, `gtk_spin_button_set_numeric()` is used to ensure that only numbers can be entered.

```
tooltip = gtk_tooltips_new();
```

```
gtk_tooltips_set_tip(tooltip, eventbox,
                     tooltip_text, NULL);
```

A tooltip (pop-up help message) is created with `gtk_tooltips_new()`. `gtk_tooltips_set_tip()` is used to associate *tooltip* with the *eventbox* widget, also specifying the message it should contain. The fourth argument should typically be `NULL`.

The `create_result_label()` function is a helper function to create a result label together with a descriptive label and tooltip.

```
gtk_label_set_selectable (GTK_LABEL(result_value), TRUE);
```

Normally, labels simply display a text string. The above code allows the text to be selected and copied, to allow pasting of the text elsewhere. This is used for the result fields so the user can easily copy them.

Continuing with the `main()` function:

```
button1 = gtk_button_new_from_stock(GTK_STOCK_QUIT);
```

This code creates a new button, using a *stock widget*. A stock widget contains a predefined icon and text. These are available for commonly used functions, such as "OK", "Cancel", "Print", etc..

```
button2 = gtk_button_new_with_mnemonic("_Calculate");
g_signal_connect (G_OBJECT (button2), "clicked",
                  G_CALLBACK(on_button_clicked_calculate),
                  (gpointer) &cb_widgets);
GTK_WIDGET_SET_FLAGS (button2, GTK_CAN_DEFAULT);
```

Here, a button is created, with the label "Calculate". The *mnemonic* is the '_C', which creates an *accelerator*. This means that when Alt-C is pressed, the button is activated (i.e. it is a keyboard shortcut). The shortcut is underlined, in common with other graphical toolkits.

The "clicked" signal (emitted when the button is pressed and released) is connected to the `on_button_clicked_calculate()` callback. A pointer to the *cb_widgets* structure is passed as the argument to the callback.

Lastly, the `GTK_CAN_DEFAULT` attribute is set. This attribute allows the button to be the default widget in the window.

```
g_signal_connect_swapped
  (G_OBJECT (cb_widgets.pg_val),
   "activate",
   G_CALLBACK (gtk_widget_grab_focus),
   (gpointer)GTK_WIDGET(cb_widgets.ri_val));
```

This code connects signals in the same way as `gtk_signal_connect()`. The difference is the fourth argument, which is a `GtkWidget` pointer. This allows the signal emitted by one widget to be received by the signal handler for another. Basically, the *widget* argument of the signal handler is given *cb_widgets.ri_val* rather than *cb_widgets.pg_val*. This allows the focus (where keyboard input is sent) to be switched to the next entry field when Enter is pressed in the first.

```
g_signal_connect_swapped
  (G_OBJECT (cb_widgets.cf_val),
   "activate",
   G_CALLBACK (gtk_window_activate_default),
   (gpointer) GTK_WIDGET(window));
```

This is identical to the last example, but in this case the callback is the function `gtk_window_activate_default()` and the widget to give to the signal handler is *window*. When Enter is pressed in the CF entry field, the default "Calculate" button is activated.

```
gtk_main();
```

This is the GTK+ event loop. It runs until `gtk_main_quit()` is called.

The signal handlers are far simpler than the interface construction. The function `on_button_clicked_calculate()` reads the user input, performs a calculation, and then displays the result.

```
void on_button_clicked_calculate( GtkWidget *widget,
                                  gpointer   data )
{
  struct calculation_widgets *w;
  w = (struct calculation_widgets *) data;
```

Recall that a pointer to *cb_widgets*, of type `struct calculation_widgets`, was passed to the signal handler, cast to a `gpointer`. The reverse process is now applied, casting *data* to a pointer of type `struct calculation_widgets`.

```
gdouble pg;
pg = gtk_spin_button_get_value
  (GTK_SPIN_BUTTON(w->pg_val));
```

This code gets the value from the `GtkSpinButton`.

```
gchar *og_string;
og_string = g_strdup_printf ("<b>%0.2f</b>", og);
gtk_label_set_markup (GTK_LABEL(w->og_result),
                      og_string);
g_free (og_string);
```

Here the result *og* is printed to the string *og_string*. This is then set as the label text using `gtk_label_set_markup()`. This function sets the label text using the *Pango Markup Format*, which uses the '`<b>`' and '`</b>`' tags to embolden the text.

```
gtk_spin_button_set_value (GTK_SPIN_BUTTON(w->pg_val),
                           0.0);
gtk_label_set_text (GTK_LABEL(w->og_result), "");
```

`on_button_clicked_reset()` resets the input fields to their default value, and blanks the result fields.

5 GTK+ and Glade

5.1 Introduction

In the previous section, the user interface was constructed entirely "by hand". This might seem to be rather difficult to do, as well as being messy and time-consuming. In addition, it also makes for rather unmaintainable code, since changing the interface, for example to add a new feature, would be rather hard. As interfaces become more complex, constructing them entirely in code becomes less feasible.

The Glade user interface designer is an alternative to this. Glade allows one to design an interface visually, selecting the desired widgets from a palette and placing them on windows, or in containers, in a similar manner to other interface designers. Figure 8 shows some screenshots of the various components of Glade.

The file `C/glade/ogcalc.glade` contains the same interface constructed in `C/plain/ogcalc`, but designed in Glade. This file can be opened in Glade, and changed as needed, without needing to touch any code.

Even signal connection is automated. Examine the "Signals" tab in the "Properties" dialogue box.

The source code is listed below. This is the same as the previous listing, but with the following changes:

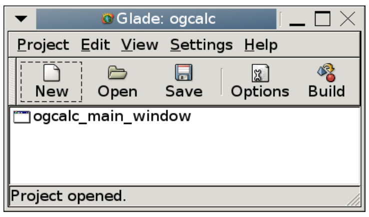
- The `main()` function does not construct the interface. It merely loads the `ogcalc.glade` interface description, auto-connects the signals, and shows the main window.
- The `cb_widgets` structure is no longer needed: the callbacks are now able to query the widget tree through the Glade XML object to locate the widgets they need. This allows for greater encapsulation of data, and signal handler connection is simpler.
- The code saving is significant, and there is now separation between the interface and the callbacks.

The running `C/glade/ogcalc` application is shown in Figure 9. Notice that it is identical to `C/plain/ogcalc`, shown in Figure 6. (No, they are *not* the same screenshot!)

5.2 Code listing

Listing 2: `C/glade/ogcalc.c`

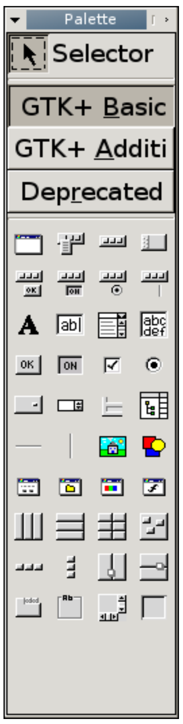
```
#include <gtk/gtk.h>
#include <glade/glade.h>

void
on_button_clicked_reset( GtkWidget *widget,
                         gpointer   data );
void
on_button_clicked_calculate( GtkWidget *widget,
                             gpointer   data );

/* The bulk of the program.  Since Glade and libglade are
   used, this is just 9 lines! */
int main(int argc, char *argv[])
{
  GladeXML *xml;
  GtkWidget *window;

  /* Initialise GTK+. */
  gtk_init(&argc, &argv);

```



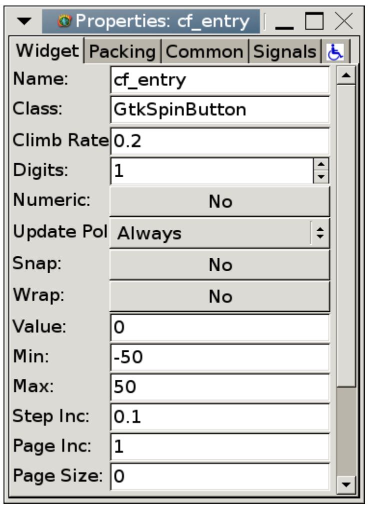
ogcalc_main_window

Project opened.

(a) Main window

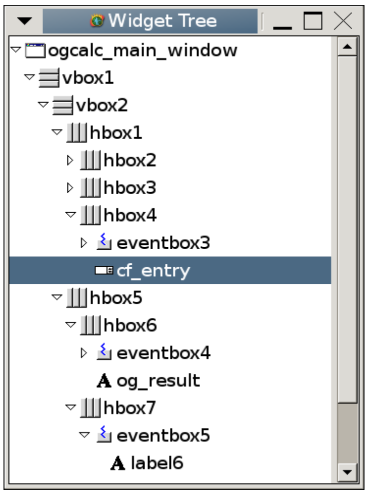


(b) Palette for widget selection



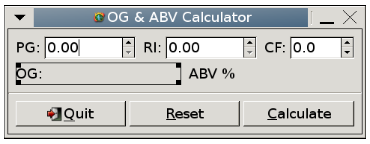
Name:	cf_entry
Class:	GtkSpinButton
Climb Rate	0.2
Digits:	1
Numeric:	No
Update Pol	Always
Snap:	No
Wrap:	No
Value:	0
Min:	-50
Max:	50
Step Inc:	0.1
Page Inc:	1
Page Size:	0

(c) Widget properties dialogue



```
ogcalc_main_window
  vbox1
    vbox2
      hbox1
        hbox2
        hbox3
        hbox4
          eventbox3
          cf_entry
      hbox5
        hbox6
          eventbox4
          og_result
        hbox7
          eventbox5
            label6
```

(d) Widget tree



PG: 0.00 RI: 0.00 CF: 0.0

OG: ABV %

(e) The program being designed

Figure 8: The Glade user interface designer.

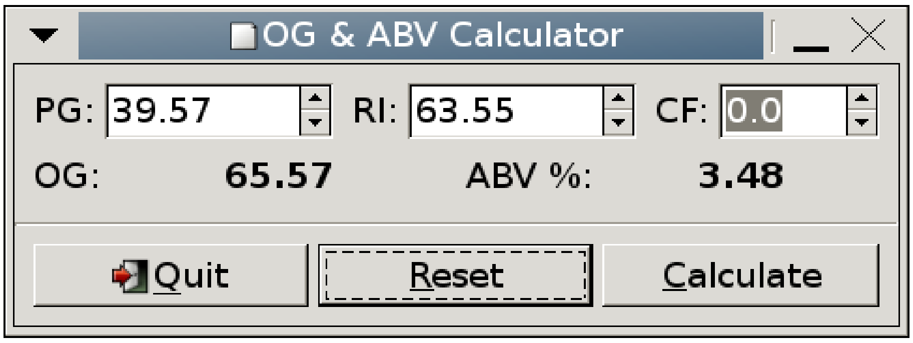


Figure 9: `C/glade/ogcalc` in action.

```
  /* Load the interface description. */
  xml = glade_xml_new("ogcalc.glade", NULL, NULL);

  /* Set up the signal handlers. */
  glade_xml_signal_autoconnect(xml);

  /* Find the main window (not shown by default, ogcalcmm.cc
     needs it to be hidden initially) and then show it. */
  window = glade_xml_get_widget (xml, "ogcalc_main_window");
  gtk_widget_show(window);

  /* Enter the GTK Event Loop.  This is where all the events
     are caught and handled.  It is exited with
     gtk_main_quit(). */
  gtk_main();

  return 0;
}

/* This is a callback.  This resets the values of the entry
   widgets, and clears the results. */
void on_button_clicked_reset( GtkWidget *widget,
                              gpointer   data )
{
  GtkWidget *pg_val;
  GtkWidget *ri_val;
  GtkWidget *cf_val;
  GtkWidget *og_result;
  GtkWidget *abv_result;

  GladeXML *xml;

  /* Find the Glade XML tree containing widget. */
  xml = glade_get_widget_tree (GTK_WIDGET (widget));

  /* Pull the other widgets out the the tree. */
  pg_val = glade_xml_get_widget (xml, "pg_entry");
  ri_val = glade_xml_get_widget (xml, "ri_entry");
  cf_val = glade_xml_get_widget (xml, "cf_entry");
  og_result = glade_xml_get_widget (xml, "og_result");
  abv_result = glade_xml_get_widget (xml, "abv_result");

  gtk_spin_button_set_value (GTK_SPIN_BUTTON(pg_val), 0.0);
  gtk_spin_button_set_value (GTK_SPIN_BUTTON(ri_val), 0.0);
  gtk_spin_button_set_value (GTK_SPIN_BUTTON(cf_val), 0.0);
  gtk_label_set_text (GTK_LABEL(og_result), "");
  gtk_label_set_text (GTK_LABEL(abv_result), "");
}

/* This callback does the actual calculation. */
void on_button_clicked_calculate( GtkWidget *widget,
                                  gpointer   data )
{
  GtkWidget *pg_val;
  GtkWidget *ri_val;
  GtkWidget *cf_val;
  GtkWidget *og_result;
  GtkWidget *abv_result;

  GladeXML *xml;

  gdouble pg, ri, cf, og, abv;
  gchar *og_string;
  gchar *abv_string;

  /* Find the Glade XML tree containing widget. */
  xml = glade_get_widget_tree (GTK_WIDGET (widget));

  /* Pull the other widgets out the the tree. */
  pg_val = glade_xml_get_widget (xml, "pg_entry");
  ri_val = glade_xml_get_widget (xml, "ri_entry");
  cf_val = glade_xml_get_widget (xml, "cf_entry");
  og_result = glade_xml_get_widget (xml, "og_result");
  abv_result = glade_xml_get_widget (xml, "abv_result");

  /* Get the numerical values from the entry widgets. */
  pg = gtk_spin_button_get_value (GTK_SPIN_BUTTON(pg_val));
  ri = gtk_spin_button_get_value (GTK_SPIN_BUTTON(ri_val));
  cf = gtk_spin_button_get_value (GTK_SPIN_BUTTON(cf_val));

  og = (ri * 2.597) - (pg * 1.644) - 34.4165 + cf;

  /* Do the sums. */
  if (og < 60)
    abv = (og - pg) * 0.130;
  else
    abv = (og - pg) * 0.134;

  /* Display the results. Note the <b></b> GMarkup tags to
     make it display in Bold. */
  og_string = g_strdup_printf ("<b>%0.2f</b>", og);
  abv_string = g_strdup_printf ("<b>%0.2f</b>", abv);

  gtk_label_set_markup (GTK_LABEL(og_result), og_string);
  gtk_label_set_markup (GTK_LABEL(abv_result), abv_string);

  g_free (og_string);
  g_free (abv_string);
}
```

To build the source, do the following:

```
cd C/glade
cc $(pkg-config --cflags libglade-2.0 gmodule-2.0) -c ogcalc.c
cc $(pkg-config --libs libglade-2.0 gmodule-2.0)
  -o ogcalc ogcalc.o
```

5.3 Analysis

The most obvious difference between this listing and the previous one is the huge reduction in size. The `main()` function is reduced to just these lines:

```
GladeXML *xml;
GtkWidget *window;

xml = glade_xml_new("ogcalc.glade", NULL, NULL);

glade_xml_signal_autoconnect(xml);

window = glade_xml_get_widget (xml, "ogcalc_main_window");
gtk_widget_show(window);
```

`glade_xml_new()` reads the interface from the file `ogcalc.glade`. It returns the interface as a pointer to a `GladeXML` object, which will be used later. Next, the signal handlers are connected with `glade_xml_signal_autoconnect()`. Windows users may require special linker flags because signal autoconnection requires the executable to have a dynamic symbol table in order to dynamically find the required functions.

The signal handlers are identical to those in the previous section. The only difference is that `struct calculation_widgets` has been removed. No information needs to be passed to them through the *data* argument, since the widgets they need to use may now be found using the `GladeXML` interface description.

```
GtkWidget *pg_val;
GladeXML *xml;
xml = glade_get_widget_tree (GTK_WIDGET (widget));
pg_val = glade_xml_get_widget (xml, "pg_entry");
```

Firstly, the `GladeXML` interface is found, by finding the widget tree containing the widget passed as the first argument to the signal handler. Once *xml* has been set, `glade_xml_get_widget()` may be used to obtain pointers to the `GtkWidgets` stored in the widget tree.

Compared with the pure C GTK+ application, the code is far simpler, and the signal handlers no longer need to get their data as structures cast to `gpointer`, which was ugly. The code is far more understandable, cleaner and maintainable.

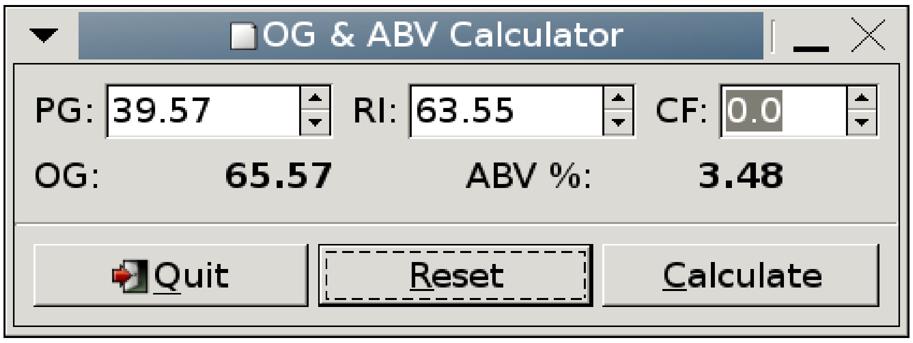


Figure 10: `C/gobject/ogcalc` in action.

6 GTK+ and GObject

6.1 Introduction

In the previous sections, the user interface was constructed entirely by hand, or automatically using libglade. The callback functions called in response to signals were simple C functions. While this mechanism is simple, understandable and works well, as a project gets larger the source will become more difficult to understand and manage. A better way of organising the source is required.

One very common way of reducing this complexity is *object-orientation*. The GTK+ library is already made up of many different objects. By using the same object mechanism (GObject), the ogcalc code can be made more understandable and maintainable.

The ogcalc program consists of a `GtkWindow` which contains a number of other `GtkWidgets` and some signal handler functions. If our program was a class (`Ogcalc`) which derived from `GtkWindow`, the widgets the window contains would be member variables and the signal handlers would be member functions (methods). The user of the class wouldn't be required to have knowledge of these details, they just create a new `Ogcalc` object and show it.

By using objects one also gains *reusability*. Previously only one instance of the object at a time was possible, and `main()` had explicit knowledge of the creation and workings of the interface.

This example bears many similarities with the C++ Glade example in Section 7. Some of the features offered by C++ may be taken advantage of using plain C and GObject.

6.2 Code listing

Listing 3: `C/gobject/ogcalc.h`

```
#include <gtk/gtk.h>

/* The following macros are GObject boilerplate. */

/* Return the GType of the Ogcalc class. */
#define OGCALC_TYPE \
          (ogcalc_get_type ())
```

```

/* Cast an object to type Ogcalc.  The object must be of
   type Ogcalc, or derived from Ogcalc for this to work.
   This is similar to a C++ dynamic_cast<>. */
#define OGCALC(obj) \
  (G_TYPE_CHECK_INSTANCE_CAST ((obj), \
                               OGCALC_TYPE, \
                               Ogcalc))

/* Cast a derived class to an OgcalcClass. */
#define OGCALC_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_CAST ((klass), \
                            OGCALC_TYPE, \
                            OgcalcClass))

/* Check if an object is an Ogcalc. */
#define IS_OGCALC(obj) \
  (G_TYPE_CHECK_TYPE ((obj), \
                      OGCALC_TYPE))

/* Check if a class is an OgcalcClass. */
#define IS_OGCALC_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_TYPE ((klass), \
                            OGCALC_TYPE))

/* Get the OgcalcClass class. */
#define OGCALC_GET_CLASS(obj) \
  (G_TYPE_INSTANCE_GET_CLASS ((obj), \
                              OGCALC_TYPE, \
                              OgcalcClass))

/* The Ogcalc object instance type. */
typedef struct _Ogcalc Ogcalc;
/* The Ogcalc class type. */
typedef struct _OgcalcClass OgcalcClass;

/* The definition of Ogcalc. */
struct _Ogcalc
{
  GtkWindow parent; /* The object derives from GtkWindow. */
  /* Widgets contained within the window. */
  GtkSpinButton *pg_val;
  GtkSpinButton *ri_val;
  GtkSpinButton *cf_val;
  GtkLabel *og_result;
  GtkLabel *abv_result;
  GtkButton* quit_button;
  GtkButton* reset_button;
  GtkButton* calculate_button;
};

struct _OgcalcClass
{
  /* The class derives from GtkWindowClass. */
  GtkWindowClass parent;
  /* No other class properties are required (e.g. virtual
     functions). */
};

/* The following functions are described in ogcalc.c */

GType ogcalc_get_type (void);

Ogcalc *
ogcalc_new (void);

gboolean
ogcalc_on_delete_event( Ogcalc   *ogcalc,
                        GdkEvent *event,
                        gpointer  data );

void
ogcalc_reset( Ogcalc    *ogcalc,
              gpointer   data );

void
ogcalc_calculate( Ogcalc    *ogcalc,
                  gpointer   data );

#endif /* OGCALC_H */
```

Listing 4: C/gobject/ogcalc.c

```
#include "ogcalc.h"

/* Declare class and instance initialisation functions and
   an ogcalc_get_type function to get the GType of Ogcalc.
   This has the side effect of registering Ogcalc as a new
   GType if it has not already been registered. */
G_DEFINE_TYPE(Ogcalc, ogcalc, GTK_TYPE_WINDOW);

GtkWidget *
_ogcalc_create_spin_entry( const gchar    *label_text,
                           const gchar    *tooltip_text,
                           GtkSpinButton **spinbutton_pointer,
                           GtkAdjustment  *adjustment,
                           guint           digits );
GtkWidget *
_ogcalc_create_result_label(const gchar   *label_text,
                            const gchar   *tooltip_text,
                            GtkLabel     **result_label_pointer );

static void
ogcalc_finalize( Ogcalc *self );

/* This is the class initialisation function.  It has no
   comparable C++ equivalent, since this is done by the
   compliler. */
static void
```

```
ogcalc_class_init ( OgcalcClass *klass )
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);

  /* Override the virtual finalize method in the GObject
     class vtable (which is contained in OgcalcClass). */
  gobject_class->finalize = (GObjectFinalizeFunc) ogcalc_finalize;
}

/* This is the object initialisation function.  It is
   comparable to a C++ constructor.  Note the similarity
   between "self" and the C++ "this" pointer. */
static void
ogcalc_init( Ogcalc *self )
{
  /* Set the window title */
  gtk_window_set_title(GTK_WINDOW (self),
                       "OG & ABV Calculator");
  /* Don't permit resizing */
  gtk_window_set_resizable(GTK_WINDOW (self), FALSE);

  /* Connect the window close button ("destroy-event") to
     a callback. */
  g_signal_connect(G_OBJECT (self), "delete-event",
                   G_CALLBACK (ogcalc_on_delete_event),
                   NULL);

  GtkWidget                  *vbox1,   *vbox2;
  GtkWidget                  *hbox1,   *hbox2;
  GtkObject                  *adjustment;
  GtkWidget                  *hsep;

  /* Create a GtkVBox to hold the other widgets.  This
     contains other widgets, which are packed in to it
     vertically. */
  vbox1 = gtk_vbox_new (FALSE, 0);
  /* Add the VBox to the Window.  A GtkWindow /is a/
     GtkContainer which /is a/ GtkWidget.  GTK_CONTAINER
     casts the GtkWidget to a GtkContainer, like a C++
     dynamic_cast. */
  gtk_container_add (GTK_CONTAINER(self), vbox1);
  /* Display the VBox.  At this point, the Window has not
     yet been displayed, so the window isn't yet visible. */
  gtk_widget_show(vbox1);

  /* Create a second GtkVBox.  Unlike the previous VBox, the
     widgets it will contain will be of uniform size and
     separated by a 5 pixel gap. */
  vbox2 = gtk_vbox_new (TRUE, 5);
  /* Set a 10 pixel border width. */
  gtk_container_set_border_width(GTK_CONTAINER(vbox2), 10);
  /* Add this VBox to our first VBox. */
  gtk_box_pack_start (GTK_BOX(vbox1), vbox2,
                      FALSE, FALSE, 0);
```

```
gtk_widget_show(vbox2);

/* Create a GtkHBox.  This is identical to a GtkVBox
   except that the widgets pack horizontally instead of
   vertically. */
hbox1 = gtk_hbox_new (FALSE, 10);

/* Add to vbox2.  The function's other arguments mean to
   expand into any extra space alloted to it, to fill the
   extra space and to add 0 pixels of padding between it
   and its neighbour. */
gtk_box_pack_start (GTK_BOX(vbox2), hbox1, TRUE, TRUE, 0);
gtk_widget_show (hbox1);

/* A GtkAdjustment is used to hold a numeric value: the
   initial value, minimum and maximum values, "step" and
   "page" increments and the "page size".  It's used by
   spin buttons, scrollbars, sliders etc.. */
adjustment = gtk_adjustment_new (0.0, 0.0, 10000.0,
                                 0.01, 1.0, 0);
/* Call a helper function to create a GtkSpinButton entry
   together with a label and a tooltip.  The spin button
   is stored in the cb_widgets.pg_val pointer for later
   use. We also specify the adjustment to use and the
   number of decimal places to allow. */
hbox2 = _ogcalc_create_spin_entry("PG:",
                                  "Present Gravity (density)",
                                  &self->pg_val,
                                  GTK_ADJUSTMENT (adjustment), 2);
/* Pack the returned GtkHBox into the interface. */
gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
gtk_widget_show(hbox2);

/* Repeat the above for the next spin button. */
adjustment = gtk_adjustment_new (0.0, 0.0, 10000.0,
                                 0.01, 1.0, 0);
hbox2 = _ogcalc_create_spin_entry("RI:",
                                  "Refractive Index",
                                  &self->ri_val,
                                  GTK_ADJUSTMENT (adjustment), 2);
gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
gtk_widget_show(hbox2);

/* Repeat again for the last spin button. */
adjustment = gtk_adjustment_new (0.0, -50.0, 50.0,
                                 0.1, 1.0, 0);
hbox2 = _ogcalc_create_spin_entry("CF:",
                                  "Correction Factor",
                                  &self->cf_val,
                                  GTK_ADJUSTMENT (adjustment), 1);
gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
gtk_widget_show(hbox2);

/* Now we move to the second "row" of the interface, used
```

```
       to display the results. */

   /* Firstly, a new GtkHBox to pack the labels into. */
   hbox1 = gtk_hbox_new (TRUE, 10);
   gtk_box_pack_start (GTK_BOX(vbox2), hbox1, TRUE, TRUE, 0);
   gtk_widget_show (hbox1);

   /* Create the OG result label, then pack and display. */
   hbox2 = _ogcalc_create_result_label("OG:",
                                       "Original Gravity (density)",
                                       &self->og_result);

   gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
   gtk_widget_show(hbox2);

   /* Repeat as above for the second result value. */
   hbox2 = _ogcalc_create_result_label("ABV %:",
                                       "Percent Alcohol By Volume",
                                       &self->abv_result);
   gtk_box_pack_start(GTK_BOX(hbox1), hbox2, TRUE, TRUE, 0);
   gtk_widget_show(hbox2);

   /* Create a horizontal separator (GtkHSeparator) and add
      it to the VBox. */
   hsep = gtk_hseparator_new();
   gtk_box_pack_start(GTK_BOX(vbox1), hsep, FALSE, FALSE, 0);
   gtk_widget_show(hsep);

   /* Create a GtkHBox to hold the bottom row of buttons. */
   hbox1 = gtk_hbox_new(TRUE, 5);
   gtk_container_set_border_width(GTK_CONTAINER(hbox1), 10);
   gtk_box_pack_start(GTK_BOX(vbox1), hbox1, TRUE, TRUE, 0);
   gtk_widget_show(hbox1);

   /* Create the "Quit" button.  We use a "stock"
      button——commonly-used buttons that have a set title and
      icon. */
   self->quit_button = GTK_BUTTON(gtk_button_new_from_stock(GTK_STOCK_QUIT));
   gtk_box_pack_start(GTK_BOX(hbox1), GTK_WIDGET(self->quit_button),
                      TRUE, TRUE, 0);
   gtk_widget_show(GTK_WIDGET(self->quit_button));

   /* This button resets the interface. */
   self->reset_button = GTK_BUTTON(gtk_button_new_with_mnemonic("_Reset"));
```

Listing 5: C/gobject/ogcalc-main.c

```
#include <gtk/gtk.h>
#include <glade/glade.h>

#include "ogcalc.h"

/* This main function merely instantiates the ogcalc class
   and displays its main window. */
int
```

```
main (int argc, char *argv[])
{
  /* Initialise GTK+. */
  gtk_init(&argc, &argv);

  /* Create an Ogcalc object. */
  Ogcalc *ogcalc = ogcalc_new();
  /* When the widget is hidden, quit the GTK+ main loop. */
  g_signal_connect(G_OBJECT (ogcalc), "hide",
                   G_CALLBACK (gtk_main_quit), NULL);

  /* Show the object. */
  gtk_widget_show(GTK_WIDGET (ogcalc));

  /* Enter the GTK Event Loop.  This is where all the events
     are caught and handled.  It is exited with
     gtk_main_quit(). */
  gtk_main();

  /* Clean up. */
  gtk_widget_destroy(GTK_WIDGET (ogcalc));

  return 0;
}
```

To build the source, do the following:

```
cd C/gobject
cc $(pkg-config --cflags libglade-2.0 gmodule-2.0) \
  -c ogcalc.c
cc $(pkg-config --cflags libglade-2.0 gmodule-2.0) \
  -c ogcalc-main.c
cc $(pkg-config --libs libglade-2.0 gmodule-2.0) \
  -o ogcalc ogcalc.o ogcalc-main.o
```

6.3 Analysis

The bulk of the code is the same as in previous sections, and so describing what the code does will not be repeated here. The `Ogcalc` class is defined in `C/gobject/ogcalc.h`. This header declares the object and class structures and some macros common to all GObject-based objects and classes. The macros and internals of GObject are out of the scope of this document, but suffice it to say that this boilerplate is required, and is identical for all GObject classes bar the class and object names.

The object structure (`_Ogcalc`) has the object it derives from as the first member. This is very important, since it allows casting between types in the inheritance hierarchy, since all of the object structures start at an offset of 0 from the start address of the object. The other members may be in any order. In this case it contains the Glade XML interface object and the widgets required to be manipulated after object and interface construction. The class structure (`_OgcalcClass`) is identical to that of the derived class (GtkWindowClass). For more complex classes, this might contain virtual function pointers. It has many similarities to a C++ vtable. Finally, the header defines the public member functions of the class.

The implementation of this class is found in `C/gobject/ogcalc.c`. The major difference to previous examples is the class registration and the extra functions for object construction, initialisation and notification of destruction. The body of the methods to reset and calculate are identical to previous examples.

The macro `G_DEFINE_TYPE` is used for convenience. Its parameters are the class name to register, the prefix used by methods of this class and the GType of the parent type we are inheriting from. It prototypes the initialisation functions defined in the source below, and it defines the function `ogcalc_get_type()`, which is used to get the the typeid (`GType`) of the class. As a side effect, this function triggers registration of the class with the GType type system. GType is a *dynamic* type system. Unlike languages like C++, where the types of all classes are known at compile-time, the majority of all the types used with GTK+ are registered on demand, except for the primitive data types and the base class `GObject` which are registered as *fundamental* types. As a result, in addition to being able to specify constructors and destructors for the *object* (or *initialisers* and *finalisers* in GType parlance), it is also possible to have initialisation and finalisation functions for both the *class* and *base*. For example, the class initialiser could be used to fix up the vtable for overriding virtual functions in derived classes. In addition, there is also an *instance_init* function, which is used in this example to initialise the class. It's similar to the constructor, but is called after object construction.

All these functions are specified in a `GTypeInfo` structure which is passed to `g_type_register_static()` to register the new type.

`ogcalc_class_init()` is the class initialisation function. This has no C++ equivalent, since this is taken care of by the compiler. In this case it is used to override the `finalize()` virtual function in the `GObjectClass` base class. This is used to specify a virtual destructor (it's not specified in the `GTypeInfo` because the destructor cannot be run until after an instance is created, and so has no place in object construction). With C++, the vtable would be fixed up automatically; here, it must be done manually. Pure virtual functions and default implementations are also possible, as with C++.

`ogcalc_init()` is the object initialisation function (C++ constructor). This does a similar job to the `main()` function in previous examples, namely contructing the interface (using Glade) and setting up the few object properties and signal handlers that could not be done automatically with Glade. In this example, a second argument is passed to `glade_xml_new()`; in this case, there is no need to create the window, since our `Ogcalc` object *is a* window, and so only the interface rooted from "ogcalc_main_vbox" is loaded.

`ogcalc_finalize()` is the object finalisation function (C++ destructor). It's used to free resources allocated by the object, in this case the `GladeXML` interface description. `g_object_unref()` is used to decrease the reference count on a `GObject`. When the reference count reaches zero, the object is destroyed and its destructor is run. There is also a `dispose()` function called prior to `finalize()`, which may be called multiple times. Its purpose is to safely free resources when there are cyclic references between objects, but this is not required in this simple case.

An important difference with earlier examples is that instead of connecting the window "destroy" signal to `gtk_main_quit()` to end the application by ending the GTK+ main loop, the "delete" signal is connected to `ogcalc_on_delete_event()`

instead. This is because the default action of the "delete" event is to trigger a "destroy" event. The object should not be destroyed, so by handling the "delete" signal and returning `TRUE`, destruction is prevented. Both the "Quit" button and the "delete" event end up calling `gtk_widget_hide()` to hide the widget rather than `gtk_main_quit()` as before.

Lastly, `C/gobject/ogcalc-main.c` defines a minimal `main()`. The sole purpose of this function is to create an instance of `Ogcalc`, show it, and then destroy it. Notice how simple and understandable this has become now that building the UI is where it belongs—in the object construction process. The users of `Ogcalc` need no knowledge of its internal workings, which is the advantage of encapsulating complexity in classes.

By connecting the "hide" signal of the `Ogcalc` object to `gtk_main_quit()` the GTK+ event loop is ended when the user presses "Quit" or closes the window. By not doing this directly in the class it is possible to have as many instances of it as ones likes in the same program, and control over termination is entirely in the hands of the user of the class—where it should be.

7 GTK+ and C++

7.1 Introduction

In the previous section, it was shown that Glade and GObject could make programs much simpler, and hence increase their long-term maintainability. However, some problems remain:

- Much type checking is done at run-time. This might mean errors only show up when the code is in production use.
- Although object-oriented, using objects in C is a bit clunky. In addition, it is very difficult (although not impossible) to derive new widgets from existing ones using GObject, or override a class method or signal. Most programmers do not bother, or just use "compound widgets", which are just a container containing more widgets.
- Signal handlers are not type safe. This could result in undefined behaviour, or a crash, if a signal handler does not have a signature compatible with the signal it is connected to.
- Signal handlers are functions, and there is often a need to resort to using global variables and casting structures to type `gpointer` to pass complex information to a callback though its *data* argument. If Glade or GObject are used, this can be avoided, however.

Gtkmm offers solutions to most of these problems. Firstly, all of the GTK+ objects are available as native C++ classes. The object accessor functions are now normal C++ *class methods*, which prevents some of the abuse of objects that could be accomplished in C. The advantage is less typing, and there is no need to manually cast between an object's types to use the methods for different classes in the inheritance hierarchy.

The Gtkmm classes may be used just like any other C++ class, and this includes deriving new objects from them through inheritance. This also enables

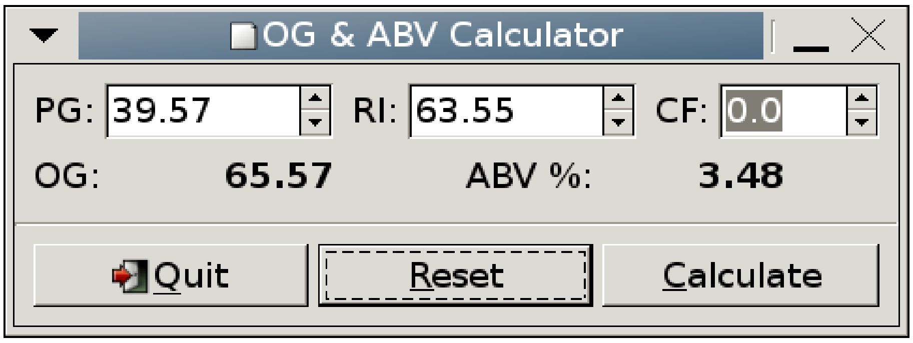


Figure 11: `C++/glade/ogcalc` in action.

all the type checking to be performed by the compiler, which results in more robust code, since object type checking is not deferred until run-time.

Signal handling is also more reliable. Gtkmm uses the `libsigc++` library, which provides a templated signalling mechanism for type-safe signal handling. The `mem_fun` objects allow signal handlers with a different signature than the signal requires to be bound, which gives greater flexibility than the C signals allow. Perhaps the most notable feature is that signal handlers may be class methods, which are recommended over global functions. This results in further encapsulation of complexity, and allows the signal handlers to access the member data of their class. Unlike the *Qt* library, Gtkmm does not require any source preprocessing, allowing plain ISO C++ to be used without extensions.

`libglademm` is a C++ wrapper around libglade, and may be used to dynamically load user interfaces as in the previous section. It provides similar functionality, the exception being that signals must be connected manually. This is because the `libsigc++` signals, connecting to the methods of individual objects, cannot be connected automatically.

`C++/glade/ogcalc`, shown in Figure 11, is identical to the previous examples, both in appearance and functionality. However, internally there are some major differences.

Firstly, the `main()` function no longer knows anything about the user interface. It merely instantiates an instance of the `ogcalc` class, similarly to `C/gobject/ogcalc`.

The `ogcalc` class is derived from the `Gtk::Window` class, and so contains all of the functionality of a `Gtk::Window`, plus its own additional functions and data. `ogcalc` contains methods called `on_button_clicked_calculate()` and `on_button_clicked_reset()`. These are the equivalents of the functions `on_button_clicked_calculate()` and `on_button_clicked_reset()` used in the previous examples. Because these functions are class methods, they have access to the class member data, and as a result are somewhat simpler than previously.

Two versions are provided, one using the basic C++ classes and methods to construct the interface, the other using `libglademm` to load and construct the interface as for the previous examples using Glade. Only the latter is discussed here. There are a great many similarities between the C and C++ versions not using Glade, and the C Gobject version and the C++ Glade version. It is left as an exercise to the reader to compare and contrast them.

7.2 Code Listing

Listing 6: C++/glade/ogcalc.h

```
#include <gtkmm.h>
#include <libglademm.h>

class ogcalc : public Gtk::Window
{
public:
  ogcalc();
  virtual ~ogcalc();

protected:
  // Calculation signal handler.
  virtual void on_button_clicked_calculate();
  // Reset signal handler.
  virtual void on_button_clicked_reset();

  // The widgets that are manipulated.
  Gtk::SpinButton* pg_entry;
  Gtk::SpinButton* ri_entry;
  Gtk::SpinButton* cf_entry;
  Gtk::Label* og_result;
  Gtk::Label* abv_result;
  Gtk::Button* quit_button;
  Gtk::Button* reset_button;
  Gtk::Button* calculate_button;

  // Glade interface description.
  Glib::RefPtr<Gnome::Glade::Xml> xml_interface;
};
```

Listing 7: C++/glade/ogcalc.cc

```
#include <iomanip>
#include <sstream>

#include <sigc++/retype_return.h>

#include "ogcalc.h"

ogcalc::ogcalc()
{
  // Set the window title.
  set_title("OG & ABV Calculator");
  // Don't permit resizing.
  set_resizable(false);

  // Get the Glade user interface and add it to this window.
  xml_interface =
    Gnome::Glade::Xml::create("ogcalc.glade",
                              "ogcalc_main_vbox");
  Gtk::VBox *main_vbox;
  xml_interface->get_widget("ogcalc_main_vbox", main_vbox);
  add(*main_vbox);

  // Pull all of the widgets out of the Glade interface.
  xml_interface->get_widget("pg_entry", pg_entry);
  xml_interface->get_widget("ri_entry", ri_entry);
  xml_interface->get_widget("cf_entry", cf_entry);
  xml_interface->get_widget("og_result", og_result);
  xml_interface->get_widget("abv_result", abv_result);
  xml_interface->get_widget("quit_button", quit_button);
  xml_interface->get_widget("reset_button", reset_button);
  xml_interface->get_widget("calculate_button",
                            calculate_button);

  // Set up signal handers for buttons.
  quit_button->signal_clicked().connect
    ( sigc::mem_fun(*this, &ogcalc::hide) );
  reset_button->signal_clicked().connect
    ( sigc::mem_fun(*this, &ogcalc::on_button_clicked_reset) );
  reset_button->signal_clicked().connect
    ( sigc::mem_fun(*pg_entry, &Gtk::Widget::grab_focus) );
  calculate_button->signal_clicked().connect
    ( sigc::mem_fun(*this,
                    &ogcalc::on_button_clicked_calculate) );
  calculate_button->signal_clicked().connect
    ( sigc::mem_fun(*reset_button, &Gtk::Widget::grab_focus) );

  // Set up signal handlers for numeric entries.
  pg_entry->signal_activate().connect
    ( sigc::mem_fun(*ri_entry, &Gtk::Widget::grab_focus) );
  ri_entry->signal_activate().connect
    ( sigc::mem_fun(*cf_entry, &Gtk::Widget::grab_focus) );
  cf_entry->signal_activate().connect
    ( sigc::hide_return
      ( sigc::mem_fun(*this,
                      &Gtk::Window::activate_default) ) );

  // Ensure calculate is the default.  The Glade default was
  // lost since it was not packed in a window when set.
  calculate_button->grab_default();
}

ogcalc::~ogcalc()
{
}

void
ogcalc::on_button_clicked_calculate()
{
  // PG, RI, and CF values.
  double pg = pg_entry->get_value();
  double ri = ri_entry->get_value();
  double cf = cf_entry->get_value();

  // Calculate OG.
  double og = (ri * 2.597) - (pg * 1.644) - 34.4165 + cf;

  // Calculate ABV.
  double abv;
  if (og < 60)
    abv = (og - pg) * 0.130;
  else
    abv = (og - pg) * 0.134;

  std::ostringstream output;
  // Use the user's locale for this stream.
  output.imbue(std::locale(""));
  output << "<b>" << std::fixed << std::setprecision(2)
         << og << "</b>";
  og_result->set_markup(Glib::locale_to_utf8(output.str()));
  output.str("");
  output << "<b>" << std::fixed << std::setprecision(2)
         << abv << "</b>";
  abv_result->set_markup
    (Glib::locale_to_utf8(output.str()));
}

void
ogcalc::on_button_clicked_reset()
{
  pg_entry->set_value(0.0);
  ri_entry->set_value(0.0);
  cf_entry->set_value(0.0);
  og_result->set_text("");
  abv_result->set_text("");
}
```

Listing 8: C++/glade/ogcalc-main.cc

```
#include <gtk/gtk.h>
#include <glade/glade.h>

#include "ogcalc.h"

// This main function merely instantiates the ogcalc class
// and displays it.
int
main (int argc, char *argv[])
{
  Gtk::Main kit(argc, argv); // Initialise GTK+.

  ogcalc window;    // Create an ogcalc object.
  kit.run(window); // Show window; return when it's closed.

  return 0;
}
```

To build the source, do the following:

```
cd C++/glade
c++ $(pkg-config --cflags libglademm-2.4) -c ogcalc.cc
```

```
c++ $(pkg-config --cflags libglademm-2.4) -c ogcalc-main.cc
c++ $(pkg-config --libs libglademm-2.4) -o ogcalc ogcalc.o \
                                          ogcalc-main.o
```

Similarly, for the plain C++ version, which is not discussed in the tutorial:

```
cd C++/plain
c++ $(pkg-config --cflags gtkmm-2.4) -c ogcalc.cc
c++ $(pkg-config --cflags gtkmm-2.4) -c ogcalc-main.cc
c++ $(pkg-config --libs gtkmm-2.4) -o ogcalc ogcalc.o \
                                            ogcalc-main.o
```

7.3 Analysis

7.3.1 ogcalc.h

The header file declares the ogcalc class.

```
class ogcalc : public Gtk::Window
```

ogcalc is derived from Gtk::Window

```
virtual void on_button_clicked_calculate();
virtual void on_button_clicked_reset();
```

on_button_clicked_calculate() and on_button_clicked_reset() are the signal handling functions, as previously. However, they are now class *member functions*, taking no arguments.

```
Gtk::SpinButton* pg_entry;
Glib::RefPtr<Gnome::Glade::Xml> xml_interface;
```

The class data members include pointers to the objects needed by the callbacks (which can access the class members like normal class member functions). Note that Gtk::SpinButton is a native C++ class. It also includes a pointer to the XML interface description. Glib::RefPtr is a templated, reference-counted, "smart pointer" class, which will take care of destroying the pointed-to object when ogcalc is destroyed.

7.3.2 ogcalc.cc

The constructor ogcalc::ogcalc() takes care of creating the interface when the class is instantiated.

```
set_title("OG & ABV Calculator");
set_resizable(false);
```

The above code uses member functions of the Gtk::Window class. The global functions gtk_window_set_title() and gtk_window_set_resizable() were used previously.

```
xml\_interface =
  Gnome::Glade::Xml::create("ogcalc.glade",
                            "ogcalc\_main\_vbox");
Gtk::VBox *main\_vbox;
xml\_interface->get\_widget("ogcalc\_main\_vbox", main\_vbox);
add(*main\_vbox);
```

The Glade interface is loaded using `Gnome::Glade::Xml::create()`, in a similar manner to the GObject example, and then the main VBox is added to the `Ogcalc` object.

```
xml_interface->get_widget("pg_entry", pg_entry);
```

Individual widgets may be obtained from the widget tree using the static member function `Gnome::Glade::Xml::get_widget()`.

Because Gtkmm uses `libsigc++` for signal handling, which uses class member functions as signal handlers (normal functions may also be used, too), the signals cannot be connected automatically, as in the previous example.

```
quit_button->signal_clicked().connect
  ( sigc::mem_fun(*this, &ogcalc::hide) );
```

This complex-looking code can be broken into several parts.

```
sigc::mem_fun(*this, &ogcalc::hide)
```

creates a `sigc::mem_fun` (function object) which points to the `ogcalc::hide()` member function of this object.

```
quit_button->signal_clicked()
```

returns a `Glib::SignalProxy0` object (a signal taking no arguments). The `connect()` method of the signal proxy is used to connect `ogcalc::hide()` to the "clicked" signal of the `Gtk::Button`.

```
calculate_button->signal_clicked().connect
  ( sigc::mem_fun(*this,
                 &ogcalc::on_button_clicked_calculate) );
calculate_button->signal_clicked().connect
  ( sigc::mem_fun(*reset_button, &Gtk::Widget::grab_focus) );
```

Here two signal handlers are connected to the same signal. When the "Calculate" button is clicked, `ogcalc::on_button_clicked_calculate()` is called first, followed by `Gtk::Widget::grab_focus()`.

```
cf_entry->signal_activate().connect
  ( sigc::hide_return
    ( sigc::mem_fun(*this,
                   &Gtk::Window::activate_default) ) );
```

`sigc::hide_return` is a special `sigc::mem_fun` used to mask the boolean value returned by `activate_default()`. The `mem_fun` created is incompatible with with the `mem_fun` type required by the signal, and this "glues" them together.

In the `ogcalc::on_button_clicked_calculate()` member function,

```
double pg
pg = pg_entry->get_value();
```

the member function `Gtk::SpinButton::get_value()` was previously used as `gtk_spin_button_get_value()`.

```
std::ostringstream output;
output.imbue(std::locale(""));
output << "<b>" << std::fixed << std::setprecision(2)
       << og << "</b>";
og_result->set_markup(Glib::locale_to_utf8(output.str()));
```

This code sets the result field text, using an output stringstream and Pango markup.

In the `ogcalc::on_button_clicked_reset()` member function,

```
pg_entry->set_value(0.0);
og_result->set_text("");
pg_entry->grab_focus();
```

class member functions are used to reset and clear the widgets as in previous examples.

7.3.3 `ogcalc-main.cc`

This file contains a very simple `main()` function.

```
Gtk::Main kit(argc, argv); // Initialise GTK+.
ogcalc window;
kit.run(window);
```

A `Gtk::Main` object is created, and then an `ogcalc` class, *window*, is instantiated. Finally, the interface is run, using `kit.run()`. This function will return when *window* is hidden, and then the program will exit.

8 Python

8.1 Introduction

Python is a popular scripting language, particularly with beginners to programming, but also used by many veteran developers. It has a clear and simple syntax, coupled with decent support for both procedural and object-oriented programming. Unlike C and C++, Python is an interpreted language, and so compilation is not necessary. This has some advantages, for example development is faster, particularly when prototyping new code. There are also disadvantages, such as programs running much slower than machine code. Worse, all code paths must be run in order to verify they are syntactically correct, and simple typing mistakes can result in a syntactically correct, but dysfunctional, program. A good C or C++ compiler would catch these errors, but Python cannot. There are tools, such as `pychecker`, which help with this. The purpose of this document is not to advocate any particular tool, however. The pros and cons of each language have been discussed at length in many other places.

Python has a language binding for GTK+, pyGTK, which allows the creation of GTK+ user interfaces directly, including the ability to derive new classes from the standard GTK+ classes, and use Python functions and object methods as callbacks. The functionality provided by `libglade` in C is also similarly available.

In the next section, examples show the use of pyGTK to create the `ogcalc` interface, using both plain GTK+ and Glade. The author wrote the Python scripts with only a few hours of Python experience, directly from the original C source, which demonstrates just how easy Python is to get into.

8.2 Code listing

Listing 9: python/plain/ogcalc

```
import pygtk
pygtk.require('2.0')
import gtk

# A utility widget for UI construction.
class OgcalcSpinEntry(gtk.HBox):

    def __init__(self, label_text, tooltip_text,
                 adjustment, digits):
        gtk.HBox.__init__(self, False, 5)

        # An eventbox.  This widget is just a container for
        # widgets (like labels) that don't have an
        # associated X window, and so can't receive X
        # events.  This is just used to we can add tooltips
        # to each label.
        eventbox = gtk.EventBox()
        eventbox.show()
        self.pack_start(eventbox, False, False)
        # Create a label.
        label = gtk.Label(label_text)
        # Add the label to the eventbox.
        eventbox.add(label)
        label.show()

        # Create a GtkSpinButton and associate it with the
        # adjustment. It adds/substracts 0.5 when the spin
        # buttons are used, and has digits accuracy.
        self.spinbutton = gtk.SpinButton(adjustment, 0.5,
                                         digits)
        # Only numbers can be entered.
        self.spinbutton.set_numeric(True)
        self.pack_start(self.spinbutton)
        self.spinbutton.show()

        # Create a tooltip and add it to the EventBox
        # previously created.
        tooltip = gtk.Tooltips()
        tooltip.set_tip(eventbox, tooltip_text)

# A utility widget for UI construction.
class OgcalcResult(gtk.HBox):

    def __init__(self, label_text, tooltip_text):
        gtk.HBox.__init__(self, False, 5)

        # As before, a label in an event box with a tooltip.
        eventbox = gtk.EventBox()
        eventbox.show()
        self.pack_start(eventbox, False, False)

        label = gtk.Label(label_text)
        eventbox.add(label)
        label.show()

        # This is a label, used to display the OG result.
        self.result_value = gtk.Label()
        # Because it's a result, it is set "selectable", to
        # allow copy/paste of the result, but it's not
        # modifiable.
        self.result_value.set_selectable(True)
        self.pack_start(self.result_value)
        self.result_value.show()

        # Add the tooltip to the event box.
        tooltip = gtk.Tooltips()
        tooltip.set_tip(eventbox, tooltip_text, None)

# The main widget (a top-level window).
class Ogcalc(gtk.Window):

    # This is a callback function.  It resets the values
    # of the entry widgets, and clears the results.
    # "data" is the calculation_widgets structure, which
    # needs casting back to its correct type from a
    # gpointer (void *) type.
    def on_button_clicked_reset(self, data=None):
        self.pg_entry.spinbutton.set_value(0.0)
        self.ri_entry.spinbutton.set_value(0.0)
        self.cf_entry.spinbutton.set_value(0.0)
        self.og_result.result_value.set_text("")
        self.abv_result.result_value.set_text("")

        # This callback does the actual calculation.  Its
        # arguments are the same as for
        # on_button_clicked_reset().

    def on_button_clicked_calculate(self, data=None):
        # Get the numerical values from the entry widgets.
        pg = self.pg_entry.spinbutton.get_value()
        ri = self.ri_entry.spinbutton.get_value()
        cf = self.cf_entry.spinbutton.get_value()

        # Do the sums.
        og = (ri * 2.597) - (pg * 1.644) - 34.4165 + cf

        if (og < 60):
            abv = (og - pg) * 0.130
        else:
            abv = (og - pg) * 0.134

        # Display the results.  Note the <b></b> GMarkup
        # tags to make it display in boldface.
        self.og_result.result_value.set_markup \
        ("<b>%(result)0.2f</b>" %{'result': og})
        self.abv_result.result_value.set_markup \
        ("<b>%(result)0.2f</b>" %{'result': abv})

    def __init__(self):
        gtk.Window.__init__(self, gtk.WINDOW_TOPLEVEL)
        self.set_title("OG & ABV Calculator")

        # Disable window resizing, since there's no point in
        # this case.
        self.set_resizable(False)

        # Connect the window close button ("destroy" event)
        # to gtk_main_quit().
        self.connect("destroy", gtk.main_quit, None)

        # Create a GtkVBox to hold the other widgets. This
        # contains other widgets, which are packed in to it
        # vertically.
        vbox1 = gtk.VBox()

        # Add the VBox to the Window. A GtkWindow /is a/
        # GtkContainer which /is a/ GtkWidget.
        # GTK_CONTAINER casts the GtkWidget to a
        # GtkContainer, like a C++ dynamic_cast.
        self.add(vbox1)
        # Display the VBox. At this point, the Window has
        # not yet been displayed, so the window isn't yet
        # visible.
        vbox1.show()

        # Create a second GtkVBox. Unlike the previous
        # VBox, the widgets it will contain will be of
        # uniform size and separated by a 5 pixel gap.
        vbox2 = gtk.VBox(True, 5)
        # Set a 10 pixel border width.
        vbox2.set_border_width(10)
        # Add this VBox to our first VBox.
        vbox1.pack_start(vbox2, False, False)
        vbox2.show()

        # Create a GtkHBox. This is identical to a GtkVBox
        # except that the widgets pack horizontally instead
        # of vertically.
        hbox1 = gtk.HBox(False, 10)

        # Add to vbox2. The function's other arguments mean
        # to expand into any extra space alloted to it, to
        # fill the extra space and to add 0 pixels of
        # padding between it and its neighbour.
        vbox2.pack_start(hbox1)
        hbox1.show()

        # A GtkAdjustment is used to hold a numeric value:
        # the initial value, minimum and maximum values,
        # "step" and "page" increments and the "page size".
        # It's used by spin buttons, scrollbars, sliders
        # etc..
        adjustment = gtk.Adjustment(0.0, 0.0, 10000.0,
                                    0.01, 1.0, 0)

        # Use a helper widget to create a GtkSpinButton
        # entry together with a label and a tooltip.  The
        # spin button is stored in the cb_widgets.pg_val
        # pointer for later use. We also specify the
        # adjustment to use and the number of decimal places
        # to allow.
        self.pg_entry = \
        OgcalcSpinEntry("PG:", "Present Gravity (density)",
                        adjustment, 2)

        # Pack the returned widget into the interface.
        hbox1.pack_start(self.pg_entry)
        self.pg_entry.show()

        # Repeat the above for the next spin button.
        adjustment = gtk.Adjustment(0.0, 0.0, 10000.0,
                                    0.01, 1.0, 0)
        self.ri_entry = \
        OgcalcSpinEntry("RI:", "Refractive Index",
                         adjustment, 2)
        hbox1.pack_start(self.ri_entry)
        self.ri_entry.show()

        # Repeat again for the last spin button.
        adjustment = gtk.Adjustment(0.0, -50.0, 50.0,
                                    0.1, 1.0, 0)
        self.cf_entry = \
        OgcalcSpinEntry("CF:", "Correction Factor",
                        adjustment, 1)
        hbox1.pack_start(self.cf_entry)
        self.cf_entry.show()

        # Now we move to the second "row" of the interface,
        # used display the results.

        # Firstly, a new GtkHBox to pack the labels into.
        hbox1 = gtk.HBox(True, 10)
        vbox2.pack_start(hbox1)
        hbox1.show()

        # Create the OG result label, then pack and display.
        self.og_result = \
        OgcalcResult("OG:", "Original Gravity (density)")

        hbox1.pack_start(self.og_result)
        self.og_result.show()

        # Repeat as above for the second result value.
        self.abv_result = \
        OgcalcResult("ABV %:", "Percent Alcohol By Volume")
        hbox1.pack_start(self.abv_result)
        self.abv_result.show()

        # Create a horizontal separator (GtkHSeparator) and
        # add it to the VBox.
        hsep = gtk.HSeparator()
        vbox1.pack_start(hsep, False, False)
        hsep.show()

        # Create a GtkHBox to hold the bottom row of
        # buttons.
        hbox1 = gtk.HBox(True, 5)
        hbox1.set_border_width(10)
        vbox1.pack_start(hbox1)
        hbox1.show()

        # Create the "Quit" button.  We use a "stock"
        # button--commonly-used buttons that have a set
        # title and icon.
        button1 = gtk.Button(None, gtk.STOCK_QUIT, False)
        # We connect the "clicked" signal to the
        # gtk_main_quit() callback which will end the
        # program.
        button1.connect("clicked", gtk.main_quit, None)
        hbox1.pack_start(button1)
        button1.show()

        # This button resets the interface.
        button1 = gtk.Button("_Reset", None, True)
        # The "clicked" signal is connected to the
        # on_button_clicked_reset() callback above, and our
        # "cb_widgets" widget list is passed as the second
        # argument, cast to a gpointer (void *).
        button1.connect_object("clicked",
            Ogcalc.on_button_clicked_reset, self)
        # connect_object is used to connect a signal from
        # one widget to the handler of another.  The last
        # argument is the widget that will be passed as the
        # first argument of the callback.  This causes
        # gtk_widget_grab_focus to switch the focus to the
        # PG entry.
        button1.connect_object("clicked",
            gtk.Widget.grab_focus, self.pg_entry.spinbutton)
        # This lets the default action (Enter) activate this
        # widget even when the focus is elsewhere.  This
        # doesn't set the default, it just makes it possible
        # to set.
        button1.set_flags(gtk.CAN_DEFAULT)
        hbox1.pack_start(button1)
        button1.show()

        # The final button is the Calculate button.
        button2 = gtk.Button("_Calculate", None, True)
        # When the button is clicked, call the
        # on_button_clicked_calculate() function.  This is
        # the same as for the Reset button.
        button2.connect_object("clicked",
            Ogcalc.on_button_clicked_calculate, self)
        # Switch the focus to the Reset button when the
        # button is clicked.
        button2.connect_object("clicked",
            gtk.Widget.grab_focus, button1)
        # As before, the button can be the default.
        button2.set_flags(gtk.CAN_DEFAULT)
        hbox1.pack_start(button2)
        # Make this button the default.  Note the thicker
        # border in the interface--this button is activated
        # if you press enter in the CF entry field.
        button2.grab_default()
        button2.show()

        # Set up data entry focus movement.  This makes the
        # interface work correctly with the keyboard, so
        # that you can touch-type through the interface with
        # no mouse usage or tabbing between the fields.

        # When Enter is pressed in the PG entry box, focus
        # is transferred to the RI entry.
        self.pg_entry.spinbutton.connect_object("activate",
            gtk.Widget.grab_focus, self.ri_entry.spinbutton)

        # RI -> CF.
        self.ri_entry.spinbutton.connect_object("activate",
            gtk.Widget.grab_focus, self.cf_entry.spinbutton)
        # When Enter is pressed in the RI field, it
        # activates the Calculate button.
        self.cf_entry.spinbutton.connect_object("activate",
            gtk.Window.activate_default, self)

if __name__ == "__main__":
    ogcalc = Ogcalc()
    ogcalc.show()
    gtk.main()
```

Listing 10: python/glade/ogcalc

```
import pygtk
pygtk.require('2.0')
import gtk
import gtk.glade

class Ogcalc(gtk.Window):

    # This function is called when the window is about to be
    # destroyed (e.g. if the close button on the window was
    # clicked).  It is not a destructor.
    def on_delete_event(self, event, data=None):
        self.hide()
        return True

    # Reset the interface.
    def reset(self, data=None):
        self.pg_val.set_value(0.0)
        self.ri_val.set_value(0.0)
        self.cf_val.set_value(0.0)
        self.og_result.set_text("")
        self.abv_result.set_text("")

    # Peform the calculation.
    def calculate(self, data=None):
        pg = self.pg_val.get_value()
        ri = self.ri_val.get_value()
        cf = self.cf_val.get_value()

        og = (ri * 2.597) - (pg * 1.644) - 34.4165 + cf;

        # Do the sums.
        if og < 60:
            abv = (og - pg) * 0.130;
        else:
            abv = (og - pg) * 0.134;

        # Display the results.  Note the <b></b> GMarkup
        # tags to make it display in boldface.
        self.og_result.set_markup("<b>%(result)0.2f</b>"
                                  %{'result': og})
        self.abv_result.set_markup("<b>%(result)0.2f</b>"
                                   %{'result': abv})

    def __init__(self):
        gtk.Window.__init__(self, gtk.WINDOW_TOPLEVEL)
        self.set_title("OG & ABV Calculator")

        # Disable window resizing, since there's no point in
        # this case.
        self.set_resizable(False)

        self.connect("delete-event",
                     Ogcalc.on_delete_event, None)

        # Load the interface description.
        self.xml = gtk.glade.XML("ogcalc.glade",
                                 "ogcalc_main_vbox", None);

        # Get the widgets.
        self.pg_val = self.xml.get_widget("pg_entry");
        self.ri_val = self.xml.get_widget("ri_entry");
        self.cf_val = self.xml.get_widget("cf_entry");
        self.og_result = self.xml.get_widget("og_result");
        self.abv_result = self.xml.get_widget("abv_result");
        self.quit_button = \
            self.xml.get_widget("quit_button");
        self.reset_button = \
            self.xml.get_widget("reset_button");
        self.calculate_button = \
            self.xml.get_widget("calculate_button");

        self.cf_val.connect_object("activate",
            gtk.Window.activate_default, self)
        self.calculate_button.connect_object("clicked",
            Ogcalc.calculate, self)
        self.calculate_button.connect_object("clicked",
            gtk.Widget.grab_focus, self.reset_button)
        self.reset_button.connect_object("clicked",
            Ogcalc.reset, self)
        self.reset_button.connect_object("clicked",
            gtk.Widget.grab_focus, self.pg_val)
        self.quit_button.connect_object("clicked",
            gtk.Widget.hide, self)

        # Set up signal handlers for numeric entries.
        self.pg_val.connect_object("activate",
            gtk.Widget.grab_focus, self.ri_val)
        self.ri_val.connect_object("activate",
            gtk.Widget.grab_focus, self.cf_val)
        self.cf_val.connect_object("activate",
            gtk.Window.activate_default, self)

        # Get the interface root and pack it into our
        # window.
        self.add(self.xml.get_widget("ogcalc_main_vbox"))

        # Ensure calculate is the default.  The Glade
        # default was lost since it wasn't in a window when
        # the default was set.
        self.calculate_button.grab_default()

if __name__ == "__main__":
    ogcalc = Ogcalc()
    ogcalc.connect("hide", gtk.main_quit, None)
    ogcalc.show()
    gtk.main()
```

8.3 Analysis

What the GTK+ classes and methods do here will not be discussed, having been covered in the previous sections. Instead, the Python-specific differences will be examined.

```
import pygtk
pygtk.require('2.0')
import gtk
```

This preamble imports the pyGTK modules for us, and checks that the GTK+ version is correct.

```
class OgcalcSpinEntry(gtk.HBox):
    def __init__(self, label_text, tooltip_text,
                 adjustment, digits):
        gtk.HBox.__init__(self, False, 5)
        ...

class OgcalcResult(gtk.HBox):
    def __init__(self, label_text, tooltip_text):
        gtk.HBox.__init__(self, False, 5)
        ...
```

These two simple classes derive from `GtkHBox`. They are the Python equivalents of the `create_spin_entry()` and `create_result_label()` functions in Section 4. They are mostly identical to the C code in terms of the objects created and the object methods used. The main difference is that `create_spin_entry()` has a *spinbutton_pointer* argument which has been dropped here. The same difference applies to `create_result_label()` for *result_label_pointer*. In Python, we can't pass pointers as easily as in C, however we can access the spinbutton as a member of the `OgcalcSpinEntry` object instead (`object.spinbutton`).

Note that because the object is derived, the `__init__()` initialiser (constructor) has to manually chain up to the parent initialiser in order to correctly initialise the class instance.

```
class Ogcalc(gtk.Window):
```

is our main application object. It derives from `gtk.Window`.

```
    def on_button_clicked_reset(self, data=None):
        self.pg_entry.spinbutton.set_value(0.0)
        ...
        self.abv_result.result_value.set_text("")
```

This function resets the interface to its initial state. Note that all the member variables are accessed through *self*, which is the class instance, and that the spinbutton and value label to be manipulated are contained within the helper objects defined above.

```
    def on_button_clicked_calculate(self, data=None):
        ...
        self.og_result.result_value.set_markup \
        ("<b>%(result)0.2f</b>" %{'result': og})
```

This function does the calculation. Note the substitution of the result value into the string, which is rather simpler than both the C and the C++ code used to construct the result string.

```
    def __init__(self):
        gtk.Window.__init__(self, gtk.WINDOW_TOPLEVEL)
        self.set_title("OG & ABV Calculator")
```

This is the initialiser for the `Ogcalc` class. It starts by chaining up the `gtk.Window` initialiser, and then calls the `set_title()` `gtk.Window` method to set the window title.

```
        self.connect("destroy", gtk.main_quit, None)
```

This connects the "destroy" signal to the `gtk.main_quit()` function. There's far less to type than the C and C++ equivalents, and hence it's rather more readable.

```
        self.pg_entry = \
        OgcalcSpinEntry("PG:", "Present Gravity (density)",
                        adjustment, 2)
```

Here we create a helper object for entering the PG value.

```
        self.abv_result = \
        OgcalcResult("ABV %:", "Percent Alcohol By Volume")
```

Here we create a helper object for displaying the ABV result.

```
        button1 = gtk.Button(None, gtk.STOCK_QUIT, False)
        button1 = gtk.Button("_Reset", None, True)
        button2 = gtk.Button("_Calculate", None, True)
```

This code creates the buttons. Unlike C and C++, where different functions or overloaded constructors were used to create an object with different parameters, Python only has a single initialiser function, which is used for both stock and non-stock widgets. Depending on whether a stock or non-stock widget is being created, the first and third, or the second arguments are redundant, respectively.

```
        button1.connect_object("clicked",
            Ogcalc.on_button_clicked_reset, self)
```

This connects the "clicked" signal to the `Ogcalc` `on_button_clicked_reset()` method of the *self* object.

```
        self.pg_entry.spinbutton.connect_object("activate",
            gtk.Widget.grab_focus, self.ri_entry.spinbutton)
```

This connects the "activate" signal to the `Ogcalc` `grab_focus()` method of the *self.ri_entry.spinbutton* object.

```
if __name__ == "__main__":
    ogcalc = Ogcalc()
    ogcalc.show()
    gtk.main()
```

The classes are intended for use as a module in a larger program. When run as a standalone script from the command-line, we "run" the class by creating an instance of it, showing it, and then run the GTK+ main loop.

The Glade code is identical, except for loading the Glade interface:

```
        self.xml = gtk.glade.XML("ogcalc.glade",
                                 "ogcalc_main_vbox", None);
```

Here the Glade interface is loaded, rooted at the "ogcalc_main_vbox" widget,

```
        self.pg_val = self.xml.get_widget("pg_entry");
```

and now a specific widget is pulled out of the XML interface description.

9 Conclusion

Which method of programming one chooses is dependent on many different factors, such as:

- The languages one is familiar with.
- The size and nature of the program to be written.
- The need for long-term maintainability.
- The need for code reuse.

For simple programs, such as `C/plain/ogcalc`, there is no problem with writing in plain C, but as programs become more complex, Glade can greatly ease the effort needed to develop and maintain the code. The code reduction and de-uglification achieved through conversion to Glade/`libglade` is beneficial even for small programs, however, so I would recommend that Glade be used for all but the most trivial code.

The C++ code using Gtkmm is slightly more complex than the code using Glade. However, the benefits of type and signal safety, encapsulation of complexity and the ability to re-use code through the derivation of new widgets make Gtkmm and `libglademm` an even better choice. Although it is possible to write perfectly good code in C, Gtkmm gives the programmer security through compiler type checking that plain GTK+ cannot offer. In addition, improved code organisation is possible, because inheritance allows encapsulation.

GObject provides similar facilities to C++ in terms of providing classes, objects, inheritance, constructors and destructors etc., and is certainly very capable (it is, after all, the basis of the whole of GTK+!). The code using GObject is very similar to the corresponding C++ code in terms of its structure. However, C++ still provides facilities such as RAII (Resource Acquisition is Initialisation) and automatic destruction when an object goes out of scope that C cannot provide.

Depending on whether the speed and safety tradeoffs are acceptable, Python may also be a valid choice. While Python code is certainly clearer and simpler, the speed of execution and lack of compile-time type checking are a concern.

There is no "best solution" for everyone. Choose based on your own preferences and capabilities. In addition, Glade is not the solution for every problem. The author typically uses a mixture of custom widgets and Glade interfaces (and your custom widgets can *contain* Glade interfaces!). Really dynamic interfaces must be coded by hand, since Glade interfaces are not sufficiently flexible. Use what is best for each situation.

10 Further Reading

The GTK+ Tutorial, and the GTK+ documentation are highly recommended. These are available from `http://www.gtk.org/`. The Gtkmm documentation is available from `www.gtkmm.org`. Unfortunately, some parts of these manuals are as yet incomplete. I hope that they will be fully documented in the future, since without good documentation, it will not be possible to write programs that take advantage of all the capabilities of GTK+ and Gtkmm, without having to read the original source code. While there is nothing wrong with reading the source, having good documentation is essential for widespread adoption of GTK+.

Documentation and examples of GObject are scarce, but Mathieu Lacage has written an excellent tutorial which is available from `http://le-hacker.org/papers/gobject/`.