Programming GTK in Free Pascal

Florian Klämpfl
and
Michaël Van Canneyt

July 2000

1 Introduction

The GTK library is a popular widget library for the X-Windows system. It is used as the basis for the GIMP graphical manipulation program and for the GNOME application framework. With its ports to Microsoft Windows and BeOS, it allows to program a graphical interface for any application in a platform independent way.

GTK is implemented in C, but it is possible to access its functionality from Free Pascal. For this, its headers have been translated to Pascal, so a program written in Free Pascal can make use of the functionality offered by GTK and its companion libraries GDK and GLIB. In fact, there is an open source project (Lazarus) which makes use of GTK in order to build an open-source alternative to the visual classes offered by Inprise's Delphi.

This article intends to present an introduction to programming GTK in Free Pascal. It by no means covers all of the functionality that GTK offers, as this would probably require a complete manual.

The first section gives some general considerations on the GTK toolkit.

2 GTK is a C library

Since GTK is an external library, some import units describing the calls in the libraries are needed. Three libraries make up the GTK programming kit:

glib this library contains some general programming tools, and defines platform independent types, which are used throughout the other libraries. To use this library, it is sufficient to include the glib unit in your **uses** clause.

gdk encapsulates the Windowing system (X or Windows) underlying GTK. It contains routines to draw on the screen, and react to various mouse or keyboard events. To use these routines, the gdk unit must be included in the **uses** clause of a unit or program.

gtk contains the widget library. This is a series of controls such as edit boxes, drop-down lists and many more, which are organised in an OOP structure. Since the library is written in C, there is no programming language support for this structure.

All definitions of the gtk library are contained in the gtk unit, which must be included in the **uses** clause of any program or unit that needs their functionality.

The GTK toolkit was programmed in C. This has some consequences for the Pascal interface, since some C constructs do not port easily to Pascal. When using the Pascal translation of the C headers, the following must be kept in mind:

1. Reserved words: Pascal reserved words in types, record element names etc. have been prepended with the word 'the'. For example **label** becomes thelabel.

2. Functions and procedures have been kept with the same names.

3. Types have been prepended with T, that is, the C type GtkWidget has become TGtkWidget.

4. Pointers to types have been defined as the type name, prepended with a P. GtkWidget * becomes PGtkWidget.

5. Records with bit-size elements: C allows to store parts of a record in individual bits; whereas in Pascal, the minimum size of an element in a record is a byte. To accommodate this, functions were defined to retrieve or set single bits from a record. The functions to retrieve a bit have the name of the record field. The procedure to set a bit has the name of the field prepended with 'set_'. For example

```
struct SomeStruct
{
  gchar *title;
  guint visible        : 1;
  guint resizeable     : 1;
};
```

translates to

```
TSomeStruct = record
 title : Pgchar;
 flag0 : word;
end;
function  visible(var a: TGtkCListColumn): guint;
procedure set_visible(var a: TGtkCListColumn; __visible: guint);
function  resizeable(var a: TGtkCListColumn): guint;
procedure set_resizeable(var a: TGtkCListColumn;__resizeable: guint);
```

6. Macros. Many C macros have not been translated. The typecasting macros have been dropped, since they're useless under Pascal. Macros to access record members have been translated, but they are to be considered as read-only. So they can be used to retrieve a value, but not to store one. e.g.

```
function GTK_WIDGET_FLAGS(wid : pgtkwidget) : longint;
```

can be used to retrieve the widget flags, but not to set them. so things like

```
GTK_WIDGET_FLAGS(wid):=GTK_WIDGET_FLAGS(wid) and someflag;
```

will not work, since this is a function, and NOT a macro as in C.

7. Calling conventions: A C compiler uses another calling convention than the Free Pascal compiler. Since many GTK functions need a callback, these callback must use the C calling convention. This means that every function that is called by GTK code, should have the **cdecl** modifier as a part of its declaration.

Compiling a GTK application is no different than compiling any other Free Pascal application. The only thing that needs to be done is to tell the free Pascal compiler where the gtk, gdk and glib libraries are located on your system. This can be done with the `-Fl` command-line switch. For example, supposing the gtk library is located in `/usr/X11/lib`, the following command-line could be used to compile your application:

```
ppc386 -Fl/usr/X11/lib mygtkapp.pp
```

This example supposes that the gtk unit is be in your unit search path. If it is not, you can add it with the `-Fu` switch.

3 The bricks of a GTK application

The building-blocks of a a GTK application are the *widgets*. Widgets are the equivalent of Delphi's controls. And although GTK is not an object oriented library, the library defines a record TGtkWidget which contains all settings common to all widgets; all widgets start with this record, and add their own specific data to it. This creates a tree-like structure with all the widgets present in the GTK library, to which your own widgets can be added.

All functions that create a particular widget return a pointer to a TGtkWidget record. It is not recommended to manipulate the contents of the widget record directly; GTK offers many functions to manipulate the members of the record, e.g. gtk_widget_set_parent or gtk_widget_get_name. To this set of functions, each new widget adds a few functions that are specific to this particular widget.

Each widget has a specific function and a specific look; there are many widgets to choose from. A complete list of widgets is outside the scope of this article; the GTK reference manual offers an overview of available widgets. In general it can be said that most widgets one would expect are present in the GTK library: Edit fields, buttons, check-boxes, various lists, menus, combo-boxes, tree views, and some pre-defined dialogs.

Any of these widgets is created with a gtk_WIDGET NAME_new call. This call can accept arguments; The number and type of arguments depend on the widget. For example, to create a button that displays a text, the call is defined as follows:

```
gtk_button_new_with_label(ALAbel : PChar)
```

All widgets can be destroyed with the gtk_widget_destroy call, irrespective of their type.

4 Showing things on the screen

To show things on the screen, it is necessary to create a window. A window is created with the the gtk_window_new call. This call accepts as an argument the type of window to be created.

Creating a window creates it's structure in memory, but doesn't show it on screen. To show this window on the screen,a call to the gtk_widget_show function is needed, as can been seen in example 1.

```
program ex1;

{$mode objfpc}

uses
 glib, gtk;

procedure destroy(widget : pGtkWidget ; data: pgpointer ); cdecl;
begin
  gtk_main_quit();
end;

var
  window :  pGtkWidget;
```

```
begin
  gtk_init (@argc, @argv);
  window := gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_signal_connect (pGTKOBJECT (window), 'destroy',
                      GTK_SIGNAL_FUNC (@destroy), NULL);
  gtk_widget_show (window);
  gtk_main ();
end.
```

If the window contains widgets, the gtk_widget_show function must be called for each widget.

Looking at example 1, one notices 2 special calls: gtk_init and gtk_main. These calls should be present in any program that uses the GTK library.

The first call initialises the GTK library. Among other things, it reads the command-line to see e.g. which display should be used.

The second call is the heart of the GTK widget library: It starts the message loop of GTK. This call will not return, unless somewhere else in the program gtk_main_quit is called. As long as the call doesn't return, GTK will wait for events such as mouse clicks, key-presses and so on. It will handle these events, but it will not notify you of any of these events except if you specifically ask for it.

A window by itself is of course not very interesting. To make it more interesting, some elements should be added.

Adding a widget to a parent is done with the gtk_container_add call. This call places a widget in a container. A container is a widget which can contain other widgets; not all widgets are containers, however.

Example 2 shows how to add a widget (a button) to a container (the window in this case). It also shows that the container has some specific properties, which can be manipulated as well (in this case, the border width). Since not each widget is a container, the window pointer must be typecasted to GTK_CONTAINER in order to be accepted by the container handling calls.

```
program ex2;

{$mode objfpc}

uses
 glib,gtk;

procedure destroy(widget : pGtkWidget; data: pgpointer ); cdecl;
begin
  gtk_main_quit();
end;

var
  window : PGtkWidget;
  button : PGtkWidget;

begin
  gtk_init (@argc, @argv);
  window := gtk_window_new (GTK_WINDOW_TOPLEVEL);
  button := gtk_button_new_with_label('Click_me');
  gtk_container_set_border_width(GTK_CONTAINER(Window),5);
  gtk_container_add(GTK_Container(window),button);
  gtk_signal_connect (PGTKOBJECT (window), 'destroy',
                      GTK_SIGNAL_FUNC (@destroy), NULL);
  gtk_widget_show (button);
  gtk_widget_show (window);
```

```
  gtk_main ();
end.
```

Adding more than 1 widget to a container is not trivial in GTK. The reason for this is that GTK has not been designed to set widgets at a specific location in their parent widget. Instead, GTK asks that you 'pack' your objects in their parent widget. This means that if the parent widget is resized, it's child widgets are resized as well, depending on the packing options that were set.

One of the reasons that the GTK library was set up this way, is that the size of a widget is not well-defined. For instance, the size of a button depends on whether it is the default widget of the window or not. Given that this is so, the placement of such a button is not well-defined either.

The most common ways of packing widgets in a parent widget are the following:

1. using a vertical box.
2. using a horizontal box.
3. using a table.

We'll discuss these ways in the subsequent. There are other ways, but these are probably the most important ones.

4.1 Using boxes

A horizontal or vertical box can be used to contain a row or column of widgets. Various options can be set to modify the spacing between the widgets, the alignment of the widgets in the box, or the behaviour of the box when the user resizes the parent widget. Boxes work only in one direction. The widgets inside a horizontal box always have the height of the box, and widgets in a vertical box always have the width of the vertical box.

You can create a horizontal box with the gtk_hbox_new call. It accepts 2 arguments: The first one is a boolean. It tells GTK whether the children should have the same space in the box. The second one is an integer, which tells GTK how much space to leave between the widgets in the box. Likewise, a vertical box can be created with the gtk_vbox_new call. This call accepts the same arguments as the first box.

Adding widgets to a box happens with the gtk_box_pack_start or gtk_box_pack_end calls. The former adds a widget at the start of the box, the latter adds a widget at the end of the box. Both functions accept the same arguments:

```
(Box : PGtkBox; Widget: PGtkWidget;
 expand gboolean; fill : gboolean;padding : guint);
```

The expand argument tells the box whether it should take the size of it's parent widget, or whether it should resize itself so that it is just large enough to fit the widgets. The latter allows to justify the widgets in the box (but only if the box is *not* homogeneous. If the box should keep the size of it's parent, then the fill argument decides what is done with the extra space available.

If fill is True then the extra space is divided over the widgets. If fill is False then the extra space is put in between the widgets.

The padding adding tells the box to add extra space for this particular widget.

The following program shows the use of a box:

```
program ex3;

{$mode objfpc}
```

```
uses
 glib , gtk;

function newbutton(ALabel : PChar) : PGtkWidget;

begin
  Result:=gtk_button_new_with_label(ALabel);
  gtk_widget_show(result);
end;

procedure destroy(widget : pGtkWidget ; data: pgpointer ); cdecl;
begin
  gtk_main_quit();
end;

var
  window,
  totalbox,
  hbox,vbox : PgtkWidget;

begin
  gtk_init (@argc, @argv);
  window := gtk_window_new (GTK_WINDOW_TOPLEVEL);
  // Box to divide window in 2 halves:
  totalbox := gtk_vbox_new(true,10);
  gtk_widget_show(totalbox);
  // A box for each half of the screen:
  hbox := gtk_hbox_new(false,5);
  gtk_widget_show(hbox);
  vbox := gtk_vbox_new(true,5);
  gtk_widget_show(vbox);
  // Put boxes in their halves
  gtk_box_pack_start(GTK_BOX(totalbox),hbox,true,true,0);
  gtk_box_pack_start(GTK_BOX(totalbox),vbox,true,true,0);
  // Now fill boxes with buttons.
  // Horizontal box
  gtk_box_pack_start(GTK_BOX(hbox),newbutton('Button 1'),false,false,0);
  gtk_box_pack_start(GTK_BOX(hbox),newbutton('Button 2'),false,false,0);
  gtk_box_pack_start(GTK_BOX(hbox),newbutton('Button 3'),false,false,0);
  // Vertical box
  gtk_box_pack_start(GTK_BOX(vbox),newbutton('Button A'),true,true,0);
  gtk_box_pack_start(GTK_BOX(vbox),newbutton('Button B'),true,true,0);
  gtk_box_pack_start(GTK_BOX(vbox),newbutton('Button C'),true,true,0);
  // Put totalbox in window
  gtk_container_set_border_width(GTK_CONTAINER(Window),5);
  gtk_container_add(GTK_Container(window),totalbox);
  gtk_signal_connect (PGTKOBJECT (window), 'destroy',
                      GTK_SIGNAL_FUNC (@destroy), NULL);
  gtk_widget_show (window);
  gtk_main ();
end.
```

What the program does is the following: It creates a window, which it splits up in two halves by means of the totalbox widget. This is a vertical box, which will contain two other boxes: a vertical box and a horizontal box. Each of these two boxes is filled with buttons. The behaviour of the boxes can be seen when the window is resized.

The effect of the various arguments to the pack calls can be seen by changing