Table of Contents

Introduction

When I first started learning to program using GTK+3.0, I found that the available information was, either for a different version or, in some areas, too detailed and technical, and somewhat confusing. In other areas, the detail I required, as a beginner, was presumably regarded as obvious and was glossed over.

After many hours struggling with limited information, and a lot of trial and error, I finally managed to write a complete, fully working program using GTK+3.0.

During this process I made a lot of notes and, when I had finished, I realised that I had enough information to write a book to help others learn what I have found out.

It is not my intention to provide a detailed programming manual, but to set out enough information to enable the reader to begin programming. Also, I have given pointers to where further, more detailed information can be found if required.

The first three chapters should be read by all programmers whatever their requirements, as they contain basic information needed for all programs. The other chapters deal with a selection of the available widgets which may or may not be required for any particular program, and therefore are designed to be dipped into as required. Full details of all of the widgets are available in the API documentation.

I have assumed that the reader has sufficient experience to handle files, and create and run programs using the terminal. I also assume that they have a basic knowledge of the 'C' language.

I trust that I have provided enough clear information to enable the reader to start writing programs without too much difficulty, and in a reasonably short time.

1 Getting Started

Full details of the GTK+3.0 Application Programming Interface (API) are contained in the GTK+3.0 API documentation, and this will be frequently referred to in future chapters, so it is recommended that a copy is downloaded from the GTK+ web-site before going any further so that the relevant parts can be accessed at any point while reading this guide.

Throughout this guide examples have been included to illustrate the point being made at the time. Each one may be compiled and run to see the effect of the instructions being discussed.

In each program, where a name of the form ….._name is used, the instruction may be compiled as it stands, but it is assumed that the programmer will replace it by a more meaningful name.

Compiling

To compile any of the examples, the simplest method is to use :-

```
gcc program_name.c -o program_name `pkg-config –cflags gtk+-3.0 –libs gtk+-3.0`
```

First, please note the use of backward sloping apostrophies. Using the upright type will not work. Also, if the compiler cannot find the pkg-config program, then it will be necessary to load and install it before continuing.

Basic Program

All GTK+3.0 programs contain the same basic instructions, and the following program is the minimum required to create a basic window.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ), NULL );
      gtk_widget_show_all( window_name );

      gtk_main();

      return 0;
```

```
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
      gtk_main_quit();

      return FALSE;
}
```

The first line of the program results in all of the required libraries being included.

The next is a declaration of the callback routine required to delete the program when the window is closed. For more information on callback routines, refer to Chapter 2.

The next line is a standard C main routine call which is followed by a declaration of a widget called 'window_name', after which gtk_init is called to carry out all of the necessary initialisation of the gtk system.

A window is then created using gtk_window_new(). For further details about the many window setting routines, see Chapter 4.

The next line connects the “delete_event” signal to the delete_event callback routine, about which there are more details in chapter 2.

Any of the items used to populate a window, and the window itself are referred to as widgets. The widgets which have been created, in this case a basic window only, are made visible by the gtk_widget_show_all() routine.

After all of the windows, and other widgets have been set up, gtk_main is called to transfer control to the GTK+ system. This will retain control until gtk_main_quit() is called, in this case in the delete_event callback routine then, when the window is destroyed, the gtk_main program will be terminated

2 Signals and Callbacks

While the program is under control of the GTK+ system, widgets can be made to generate signals, usually as a result of action by the user. These signals can be used to initiate execution of instructions contained in a function referred to as a callback routine. The instructions can be anything specified by the programmer.

There are two types of signal. The first is usually emitted by a widget as a result of user action e.g. clicking on a button, and the second is emitted as a result of action by an input device, e.g. clicking a mouse button. The latter are referred to as events.

Widget Signals

The process for adding a widget to a window and setting up a callback routine is the same for all widgets. To demonstrate this, a button will be added to the window and a callback will be used to change the label in the button when the button is clicked.

First the callback routine should be declared, but to do this it is necessary to determine the appropriate format for the declaration. Most widget callback routines are of similar form but it is best to be sure, so search the API documentation for the GtkButton widget. Click on this and then search for the list of signals. In this case the clicked signal will be used. Select clicked to find that the callback routine is of the form :-

```
void user_function (GtkButton *button, gpointer user_data)
```

Therefore, at the start of the program, before the main routine is started, add the declaration :-

```
static void callback_name( GtkWidget*, gpointer );
```

Then, at the start of the main routine add a declaration of the button :-

```
GtkWidget   *button_name;
```

After the window has been created and its callback routine connected, create a button with a label. The format of this command is again shown in the API documentation as one of the many routines listed under Functions in the GtkButton section. This shows that the format of the command is :-

```
GtkWidget * gtk_button_new_with_label (const gchar c*label);
```

Therefore the button is created by :-

```
button_name = gtk_button_new_with_label( “Off” );
```

Clicking on the button must now be connected to the required callback routine using a command of the form :-

```
g_signal_connect( widget, signal, callback, data );
```

Where widget is the name of the widget generating the signal,
signal is the signal which has been generated,
callback is the name of the callback routine,
data is a pointer to data used by the callback routine.

This is the general format for all signals generated by widgets, those generated by input devices, known as events, have a different form shown below.

For the purposes of the example, the command should be :-

```
g_signal_connect( button_name, “clicked”, G_CALLBACK( callback_name,
      NULL );
```

The pointer to the data is NULL in this case as the callback does not require any data. Where data is required, it can be convenient to create a data structure and use the name of the structure as a pointer to all of the data contained in it.

The button now should be added to the window using:-

```
gtk_container_add( GTK_CONTAINER( window_name ), button_name );
```

Finally the callback needs to be defined :-

```
static void callback_name( GtkWidget *widget, gpointer data )
{
      if ( strcmp( gtk_button_get_label( GTK_BUTTON( widget ) ), “Off” )
            == 0 )
            gtk_button_set_label( GTK_BUTTON( widget ), “On” );
      else
            gtk_button_set_label( GTK_BUTTON( widget ), “Off” );
}
```

The details of the button_set_label and button_get_label commands are available in the GtkButton section of the API documentation.

Note that the pointer to the button is the first parameter passed to the callback and has type widget, therefore this needs to be retyped to button for the button_get and button_set

routines as they are expecting a pointer of type button.

Putting all of the above together gives :-

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );
static void callback_name( GtkWidget*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name, *button_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );

      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ),
                        NULL );
      button_name = gtk_button_new_with_label( "Off" );
      g_signal_connect( button_name, "clicked", G_CALLBACK( callback_name ), NULL );
      gtk_container_add( GTK_CONTAINER( window_name ), button_name );

      gtk_widget_show_all( window_name );

      gtk_main();

      return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
      gtk_main_quit();

      return FALSE;
}

static void callback_name( GtkWidget *widget, gpointer data )
{
      if ( strcmp( gtk_button_get_label( GTK_BUTTON( widget ) ), "Off" ) == 0 )
            gtk_button_set_label( GTK_BUTTON( widget ), "On" );
      else
            gtk_button_set_label( GTK_BUTTON( widget ), "Off" );
}
```

Events

Events are signals which are usually emitted when an input device, such as a mouse, has generated a signal. They are part of the Gdk system and therefore the full details of all of the available events are detailed in the Gdk API documentation. In general, events are treated in a similar manner to widget signals but there are differences which will be highlighted in the following example.

In the example, pressing the left mouse button generates an event which passes the mouse position to the callback routine. This will be used to set the label's text to show the position..

First, as with signals, the callback routine should be declared. Event callbacks have an additional parameter to pass event data to the callback. In this case it will be the mouse x and y position when the button is pressed.

```
static void callback_name( GtkWidget*, GdkEvent*, gpointer );
```

As there is no need to pass other data to the callback routine in this example, the third parameter will be used to pass a pointer to the label i.e.

```
static void callback_name( GtkWidget*, GdkEvent*, GtkLabel* );
```

There is no need for a button to be created for this example, but a label will be used to display the position of the mouse when the button is pressed, therefore, in this example, add the label in a similar manner to the button in the previous example..

```
GtkWidget   *label_name;
```

And create the label using :-

```
label_name = gtk_label_new( “Position” );
```

A drawing area has also been used for the mouse pointer to move over when the mouse button is pressed. This is added and created in a similar manner but a request to determine its size is sent to the window manager to make a suitably large area for the mouse to move over :-

```
GtkWidget   *drawing_area_name;

drawing_area_name = gtk_drawing_area_new();
gtk_widget_set_size_request( drawing_area_name, 300, 300 );
```

These now need to be added to the window. As there is more than one widget, it will be necessary to use one of the forms of packing boxes, in this case a vertical box. Full details of packing widgets into windows are given in chapter 3.

```
gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
```

The required events must now be added to the window. The usual ones for a mouse are GDK_BUTTON_PRESS_MASK | GDK_BUTTON_MOTION_MASK | GDK_BUTTON_RELEASE_MASK. The first is used when the mouse button is pressed, the second when the mouse is moved, and the third when the button is released, the event data providing the x and y position of the mouse in each case In this example, only the

BUTTON_PRESS_MASK is required.

This event must now be connected to the callback in a similar manner to a signal.

```
g_signal_connect( G_OBJECT( window ), “button_press_event”,
            G_CALLBACK( callback_name ), ( gpointer ) label_name );
```

Note that the fourth parameter is a pointer to the label in order to pass it to the callback.

Putting all of the above together gives :-

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );
static gboolean callback_name( GtkWidget*, GdkEventButton*, GtkLabel*  );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name, *v_box_name, *drawing_area_name, *label_name,
                  *separator_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ),
                  NULL );

      drawing_area_name = gtk_drawing_area_new();
      gtk_widget_set_size_request( drawing_area_name, 300, 300 );
      separator_name = gtk_separator_new( GTK_ORIENTATION_HORIZONTAL );
      label_name = gtk_label_new( "Position" );

      v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
      gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
      gtk_box_pack_start( GTK_BOX( v_box_name ), drawing_area_name, FALSE, FALSE, 0 );
      gtk_box_pack_start( GTK_BOX( v_box_name ), separator_name, FALSE, FALSE, 0 );
      gtk_box_pack_start( GTK_BOX( v_box_name ), label_name, FALSE, FALSE, 20 );

      gtk_widget_add_events( drawing_area_name, GDK_BUTTON_PRESS_MASK );
      g_signal_connect( G_OBJECT( drawing_area_name ), "button_press_event",
      G_CALLBACK( callback_name ), ( gpointer ) label_name );

      gtk_widget_show_all( window_name );

      gtk_main();

      return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
      gtk_main_quit();

      return FALSE;
}
```

```
static gboolean callback_name( GtkWidget *area, GdkEventButton *event, GtkLabel *label )
{
      gint x = event->x, y = event->y, i, width, height;

      width = gtk_widget_get_allocated_width( area );
      height = gtk_widget_get_allocated_height( area );
      if ( x <= width / 2 )
      {
            if ( y <= height / 2 )
                  gtk_label_set_text( GTK_LABEL( label ), "Top left" );
            else
                  gtk_label_set_text( GTK_LABEL( label ), "Bottom Left" );
      }
      else
      {
            if ( y <= height / 2 )
                  gtk_label_set_text( GTK_LABEL( label ), "Top right" );
            else
                  gtk_label_set_text( GTK_LABEL( label ), "Bottom right" );
      }

return FALSE;
}
```

Note that, if the program is copied and pasted into an editor, the minus sign becomes a non_printing character, therefore this needs to be deleted and a proper minus sign inserted to get the required effect.

Swapped Widget Callbacks

There is a third variety of call to a callback routine which is used when it is required for the signal from one widget to be used in place of the signal from another. The most common occurrence is when a button is clicked in order to close a window. To do this the g_signal_connect_swapped command is used.

Returning to the basic program, alter it as follows :-

```
      GtkWidget   *window_name, *button_name;

      gtk_init ...

      window_name = ...
      g_signal_connect( ...

      button_name = gtk_button_new_with_label( "Exit" );
      gtk_container_add( GTK_CONTAINER( window_name ), button_name );
      g_signal_connect_swapped( button_name, "clicked", G_CALLBACK
            ( delete_event ), window_name );

      gtk_widget_show_all ...
      .
      .
```

The g_signal_connect_swapped command, in this instance, results in the “clicked” signal from the button being used instead of the “delete_event” signal to call the delete_event callback routine for the window.

3 Packing

Because the gtk+ window manager automatically resizes and positions widgets as it thinks is suitable, it is not possible to position widgets using absolute coordinates, therefore a system of packing widgets into windows has been developed using theoretical boxes. There is a variety of specialised types of box to choose from, depending on requirements, but there is a basic box which is commonly used and which exists in either horizontal or vertical form.

It should be remembered that a container, such as a box, is a widget, and boxes have widgets placed within them, therefore containers may be nested within each other. Thus a notebook container may have a grid container in one of its pages, and the grid could have a box within one of its cells.

Boxes

In an horizontal box, widgets are packed in order either from left to right, or from right to left. Equally, for a vertical box, widgets will be packed either from top to bottom, or from bottom to top.

A box is created using :-

```
gtk_box_new( orientation, spacing );
```

orientation is either GTK_ORIENTATION_HORIZONTAL or
GTK_ORIENTATION_VERTICAL
spacing = number of pixels between children.

A child widget is added to a box using :-

```
gtk_box_pack_start( box, child, expand, fill, padding );
```

or

```
gtk_box_pack_end( box, child, expand, fill, padding );
```

depending on whether the box is to be packed from the beginning, forward, or from the end, backward.

If expand is TRUE,any spare space allocated to box will be evenly divided between all children for which expand is set to TRUE.

If fill is TRUE, space given to the child by the expand option is allocated to the child itself and causes the child to expand. It has no effect if expand is FALSE.

Padding is the amount of space, in pixels, to be added equally either side of each child widget.

The following examples should make things clearer.

In the first example, the basic program has a vertical box added to the window, and a horizontal box placed within the vertical box. Three buttons are added to the horizontal box, starting at the left. The boxes have the spacing parameter set to 0, the expand and fill packing parameters are set to FALSE and the padding parameter is 0.

In the basic program it will be necessary to declare the boxes and the buttons :-

```
GtkWidget   *window_name, *h_box_name, *v_box_name;
GtkWidget   *button_1_name, *button_2_name, *button_3_name;
```

In this example, after the window has been created, the size should be requested to be 600 x 600 pixels. This is to provide spare space to be allocated to the widgets. The size is requested using :-

```
gtk_window_set_default_size( GTK_WINDOW( window_name ), 600, 600 );
```

The GTK_WINDOW cast is required because the window has been declared as a widget, but the command requires the window name to be of type window.

After all of the window creation commands, but before the window show command, add instructions to create the vertical box, and add the horizontal box to it :-

```
v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
h_box_name = gtk_box_new( GTK_ORIENTATION_HORIZONTAL, 0 );
gtk_box_pack_start( GTK_BOX( v_box_name ), h_box_name, FALSE, FALSE, 0 );
```

Then create the three buttons, adding each one in turn to the horizontal box :-

```
button_1_name = gtk_button_new_with_label( "Button 1" );
gtk_box_pack_start( GTK_BOX( h_box_name ), button_1_name, FALSE, FALSE, 0 );
button_2_name = gtk_button_new_with_label( "Button 2" );
gtk_box_pack_start( GTK_BOX( h_box_name ), button_2_name, FALSE, FALSE, 0 );
button_3_name = gtk_button_new_with_label( "Button 3" );
gtk_box_pack_start( GTK_BOX( h_box_name ), button_3_name, FALSE, FALSE, 0 );
```

Running the program should show a square window with three buttons at the top edge with no spacing between them.

Now set the spacing parameter of the h_box_name to 40 and gaps will appear between the buttons, but button 1 will still be placed at the left edge. This is because the spacing parameter only provides spacing between widgets but does not provide any before the first, or after the last widget in the box.

Reset the spacing to 0 but now set the padding parameter for each button to 20. This time a space of 20 pixels appears before button 1, 2 spaces of 20 pixels between buttons 1 and 2, and between buttons 2 and 3, and the remaining space to the right of the buttons. This is because the padding parameter adds the space of 20 pixels to either side of each button.

Reset the padding parameter to 0 and, for each button, set the Expand parameter to TRUE. This will take the spare space to the right of the buttons and distribute it evenly, either side of each button i.e. if there were 60 pixels to the right of the third button, 10 would be placed before the first button, 20 between each of the buttons, and 10 after the third button. This is useful to lay out the buttons equally spaced across a window.

Leaving the Expand parameter set to TRUE, also set the Fill parameter to true for button 2. Button 2 will now spread to fill the space allocated to it. Note that the Fill parameter has no effect if the Expand parameter is FALSE.

Grids

If it is required to lay out widgets in rows and columns, it may be more convenient to use a grid container. This will make it easier to align the widgets both vertically and horizontally.

The use of a grid is similar to that for boxes but, having created the grid, the widgets are added using a command of the form :-

```
gtk_grid_attach( *grid, *widget, column, row, width, height );
```

where grid is the created grid,
widget is the widget to be added,
column is the column number,
row is the row number,
width is the number of columns to be spanned by the widget,
height is the number of rows to be spanned by the widget.

In the previous box example, replace the declarations of the boxes :-

```
GtkWidget *window_name, *grid_name;
```

The buttons and window should be created as before, but replace the commands to create the boxes as follows :-

```
grid_name = gtk_grid_new();
gtk_container_add( GTK_CONTAINER( window_name ), grid_name );
```

and then add the buttons :-

```
button_1_name = gtk_button_new_with_label( "Button 1" );
gtk_grid_atatach( GTK_GRID( grid_name ), button_1_name, 1, 1, 1, 1 );
button_2_name = gtk_button_new_with_label( "Button 2" );
gtk_grid_atatach( GTK_GRID( grid_name ), button_2_name, 2, 2, 1, 1 );
button_3_name = gtk_button_new_with_label( "Button 3" );
gtk_grid_atatach( GTK_GRID( grid_name ), button_3_name, 3, 3, 1, 1 );
```

Others

There are several other special purpose containers of which the Listbox, Stack, Overlay, Paned and Notebook containers may be worth considering. The API documentation gives a basic description of the operation of each container, and then experimenting to achieve the required effect is probably the best approach.

4 Windows

Windows are created, having been declared as a widget, using :-

```
window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
```

The function parameter can be either GTK_WINDOW_TOPLEVEL or GTK_WINDOW_POPUP. Most windows should be TOPLEVEL ones and, if this is not appropriate, a dialog should be considered before using a POPUP window.

The window's title may be set using :-

```
gtk_window_set_title( GTK_WINDOW( window_name ), "Title_name" );
```

Note that the window_name, which was declared as a widget, has to be cast as a window for this function.

A window's size and position are always determined by the window manager, but requests can be made for the initial values, which the window manager usually accepts.
The initial size can be set using :-

```
gtk_window_set_default_size( GTK_WINDOW( window_name ), width, height );
```

The size is set to the number of pixels specified by the width and height parameters.

and the window may be positioned using :-

```
gtk_window_move( GTK_WINDOW( window_name ), across, down );
```

The window_name must again be recast. The window will be positioned by the number of pixels specified in the across and down parameters relative to the top left corner of the screen.
The main window of a program should have its delete event, which is triggered when the window is closed by clicking on the cross at the top right hand corner, connected to a callback which calls the gtk_main_quit function, in order to close the program when the window is closed, otherwise the program will continue to run until the computer is shut down. This is shown in the example of a basic program detailed in chapter 1.

The window title, the resizing and closing buttons and the drag handles are all referred to as window decorations. If a plain window is required without decorations, it may be set up using :-

```
gtk_window_set_decorated( GTK_WINDOW( window_name ), FALSE );
```

but remember to create some method of closing the window, or hiding it, unless it is required at all times.

When a widget is created it is not initially visible and therefore, when a program is initially set up it is usual to call the gtk_widget_show_all(window_name) function immediately before the gtk_main() call in order to make all widgets visible. Widgets may be set as visible or hidden using :-

```
gtk_widget_show( widget_name ); or gtk_widget_hide( widget_name );
```

Any child widget made visible will not be shown until its parent widget is made visible, and equally, when a parent widget is made visible, any hidden child widgets will not be shown.

Where a second window has been created as a child of an existing window, the child window may be made modal. This means that, while the child window exists, the main window will not accept any input. When the child window has been closed, the main window again becomes active. A window may be made modal by :-

```
gtk_window_set_modal( GTK_WINDOW( window_name ), TRUE );
```

The following example shows the features described above in a complete program. The points to note are that, in the g_signal_connect call to the show_window callback, the fourth parameter is a pointer to the modal window in order to pass it to the callback routine, also, if the modal window is visible, it is not possible to close the main window by clicking on its cross. To close the program the Exit button on the modal window must be clicked.

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

static void show_window( GtkWidget*, GtkWidget* );
static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name, *modal_window_name;
      GtkWidget   *v_box_name, *h_box_name, *main_button_name, *modal_button_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ), NULL );

      v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
      gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
      h_box_name = gtk_box_new( GTK_ORIENTATION_HORIZONTAL, 0 );
      gtk_box_pack_start( GTK_BOX( v_box_name ), h_box_name, FALSE, FALSE, 70 );
      main_button_name = gtk_button_new_with_label( "Show modal window" );
      gtk_box_pack_start( GTK_BOX( h_box_name ), main_button_name, TRUE, FALSE, 0 );
```

```
	gtk_widget_show_all( window_name );

	modal_window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
	gtk_window_set_decorated( GTK_WINDOW( modal_window_name ), FALSE );
	gtk_window_set_default_size( GTK_WINDOW( modal_window_name ), 200, 120 );
	gtk_window_move( GTK_WINDOW( modal_window_name ), 250, 450 );
	gtk_window_set_modal( GTK_WINDOW( modal_window_name ), TRUE );

	v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
	gtk_container_add( GTK_CONTAINER( modal_window_name ), v_box_name );
	h_box_name = gtk_box_new( GTK_ORIENTATION_HORIZONTAL, 0 );
	gtk_box_pack_start( GTK_BOX( v_box_name ), h_box_name, FALSE, FALSE, 40 );
	modal_button_name = gtk_button_new_with_label( "Exit" );
	gtk_box_pack_start( GTK_BOX( h_box_name ), modal_button_name, TRUE, FALSE, 40 );

	g_signal_connect( main_button_name, "clicked", G_CALLBACK( show_window ),
	                  ( gpointer )modal_window_name );
	g_signal_connect_swapped( modal_button_name, "clicked", G_CALLBACK( delete_event ),
	                  NULL );

	gtk_main();

	return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
	gtk_main_quit();

	return FALSE;
}

static void show_window( GtkWidget *widget, GtkWidget *window )
{
	gtk_widget_show_all( window );
}
```

5 Dialogs

A dialog is a pop-up window which is usually used to prompt the user for input. A dialog is split into two sections, the top section which is referred to as the 'content area' is a v_box which is normally used to contain a label which passes a request to the user. The lower section is referred to as the 'action area' and usually contains buttons to perform actions such as cancel or ok. Dialogs are normally modal so that an action must be activated before any further activity on the main window can be carried out.

To create a dialog use :-

```
dialog_name = gtk_dialog_new_with_buttons( "Title_name", window_name, flags, button responses, NULL );
```

where Title_name is the title to be diplayed at the top of the dialog,
window_name is the name of the parent window,
flags can be one or both of GTK_DIALOG_MODAL or
GTK_DIALOG_DELETE_WITH_PARENT,
the button responses are pair combinations of a button and a response ID number.
When any button is clicked, a dialog issues a “response” signal with the ID number which has been associated with the button, so that the button that has been clicked can be identified. A list of all of the required buttons, with their IDs, should be given, separated by commas. After the last button pair, NULL should be added.

The following example shows the usual main window, this time with a button labelled “Exit”. Clicking on this button will call the show_dialog callback, and it should be noted that the g_signal_connect command has, as its fourth parameter, the main window name, so that the dialog can be associated with it.

In the callback routine, the dialog is created as modal with two buttons, a Cancel button with an ID of 1, and an OK button with an ID of 2. A message is also placed in the content area.

Finally the dialog is run and, if the Cancel button has been clicked, the dialog is destroyed, or if the OK button has been clicked, the program is destroyed.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );
static void show_dialog( GtkWidget*, GtkWidget* );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name, *dialog_name;
      GtkWidget   *v_box_name, *h_box_name, *main_button_name;
```

```
	gtk_init( &argc, &argv );

	window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
	gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
	gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
	gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
	g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ),
		NULL );
	v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
	gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
	h_box_name = gtk_box_new( GTK_ORIENTATION_HORIZONTAL, 0 );
	gtk_box_pack_start( GTK_BOX( v_box_name ), h_box_name, FALSE, FALSE, 70 );
	main_button_name = gtk_button_new_with_label( "Exit" );
	gtk_box_pack_start( GTK_BOX( h_box_name ), main_button_name, TRUE, FALSE, 0 );

	gtk_widget_show_all( window_name );

	g_signal_connect( main_button_name, "clicked", G_CALLBACK( show_dialog ),
		( gpointer )window_name );

	gtk_main();

	return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
	gtk_main_quit();

	return FALSE;
}

static void show_dialog( GtkWidget *widget_name, GtkWidget *new_window_name )
{
	gint result_name;
	GtkWidget *dialog_name, *content_area_name, *label_name;

	dialog_name = gtk_dialog_new_with_buttons( "Dialog", GTK_WINDOW( new_window_name ),
			GTK_DIALOG_MODAL, ( "Cancel" ), 1, ( "OK" ), 2, NULL );
	gtk_window_set_default_size( GTK_WINDOW( dialog_name ), 200, 120 );
	content_area_name = gtk_dialog_get_content_area( GTK_DIALOG( dialog_name ) );
	label_name = gtk_label_new( "Do you really wish to exit" );
	gtk_container_add( GTK_CONTAINER( content_area_name ), label_name );
	gtk_widget_show_all( dialog_name );

	result_name = gtk_dialog_run( GTK_DIALOG( dialog_name ) );
	switch ( result_name )
	{
		case 1 :
			gtk_widget_destroy( dialog_name );
		break;
		case 2 :
			gtk_main_quit();
		break;
		default :
		break;
	}
}
```

6 Labels

Labels usually contain a small amount of text to provide information to the user. They may also be used to add identification text to a widget such as a button.

To create a label use :-

```
label_name = gtk_label_name( "label_text" );
```

or for a button use :-

```
button_name = gtk_button_new_with_label( "label_text" );
```

Labels may contain underlined characters, known as mnemonics, which may be used in conjunction with the ALT key to activate the widget containing the label. For example, a button containing a label may have a mnemonic character which, when pressed with the ALT key will have the same result as if the button has been clicked. The underline does not become visible until the ALT key is pressed.

To create a button with a mnemonic use :-

```
button_name = gtk_button_new_with_mnemonic( "_labeltext" );
```

This is shown in the following example.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name;
      GtkWidget   *v_box_name, *h_box_name, *button_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
      v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
      gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
      h_box_name = gtk_box_new( GTK_ORIENTATION_HORIZONTAL, 0 );
      gtk_box_pack_start( GTK_BOX( v_box_name ), h_box_name, FALSE, FALSE, 70 );
      button_name = gtk_button_new_with_mnemonic( "_Labeltext" );
      gtk_box_pack_start( GTK_BOX( h_box_name ), button_name, TRUE, FALSE, 0 );
      g_signal_connect_swapped( button_name, "clicked", G_CALLBACK( delete_event ),
                  window_name );
      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ), NULL );
```

```
	gtk_widget_show_all( window_name );

	gtk_main();

	return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
	gtk_main_quit();

	return FALSE;
}
```

7 Images

Images may be displayed in a window, the usual variety being a pixbuf loaded from a file. This can be achieved using :-

```
image_name = gtk_image_new_from_file( "image.png" );
```

If the image is directly available, it may be loaded using :-

```
image_name = gtk_image_new_from_pixbuf( pixbuf_name );
```

For other types of image, reference should be made to the API documentation.

The following is an example of loading the image from a file. To run this example, it is necessary to have the image file in the same folder as the example program.

To the basic program declarations add :-

```
GtkWidget   *v_box_name, *image_name;
```

and after the window has been created add :-

```
v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
image_name = gtk_image_new_from_file( "image.png" );
gtk_box_pack_start( GTK_BOX( v_box_name ), image_name, FALSE, FALSE, 20 );
```

8 Buttons

Basic Buttons

A button is usually used to trigger a callback function, either when it is clicked, or when the mnemonic key is pressed on the keyboard at the same time as the ALT key.

To create a button without a mnemonic use :-

```
button_name = gtk_button_new_with_label( "label_text" );
```

and for one with a mnemonic use :-

```
button_name = gtk_button_new_with_mnemonic( "_Labeltext" );
```

either of these, when activated, will cause the button to emit the “clicked” signal, which should be connected to a callback using :-

```
g_signal_connect( button_name, "clicked", G_CALLBACK( callback_name ), NULL ):
```

The callback should be declared as :-

```
static void callback_name( GtkWidget*, gpointer );
```

and should normally have the form :-

```
static void callback_name( GtkWidget *widget_name, gpointer data_name )
{
      \\ callback content
}
```

The following example has a different callback format as it is swapped to the “delete_event” used to delete the main window.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name;
      GtkWidget   *v_box_name, *h_box_name, *button_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
```

```
	v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
	gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
	h_box_name = gtk_box_new( GTK_ORIENTATION_HORIZONTAL, 0 );
	gtk_box_pack_start( GTK_BOX( v_box_name ), h_box_name, FALSE, FALSE, 70 );
	button_name = gtk_button_new_with_mnemonic( "_Labeltext" );
	gtk_box_pack_start( GTK_BOX( h_box_name ), button_name, TRUE, FALSE, 0 );
	g_signal_connect_swapped( button_name, "clicked", G_CALLBACK( delete_event ),
	                  window_name );
	g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ), NULL );
	gtk_widget_show_all( window_name );

	gtk_main();

	return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
	gtk_main_quit();

	return FALSE;
}
```

Toggle Buttons

A toggle button is one which, when clicked, stays activated until it is clicked again. It is created using :-

```
	toggle_button_name = gtk_toggle_button_new();.
```

Similarly to an ordinary button, it may also be created with a label or a mnemonic.

To read the current state of the toggle button use :-

```
	state = gtk_toggle_button_get_active( GTK_TOGGLE_BUTTON( toggle_button_name ));
```

or set it with :-

```
	gtk_toggle_button_set_active( GTK_TOGGLE_BUTTON( toggle_button_name ), state );
```

where state is TRUE or FALSE.

Whenever the toggle button is clicked, a “toggled” signal is emitted which should be connected to a callback function using :-

```
	g_signal_connect( toggle_button_name, "toggled", G_CALLBACK( callback_name ),
	                  NULL );
```

the callback function having the form :-

```
	void callback_name( GtkToggleButton *toggle_button, gpointer data )
```

```
{
      \\ callback_content;
}
```

Check Buttons

A check button is a small toggle button which is usually placed next to its label. It is created using :-

```
check_button_name = gtk_check_button_new();
```

In all other respects it is the same as a toggle button in that it may be created with a label or a mnemonic, and also emits the “toggled” signal when clicked, which may be connected to a callback function.

Radio Buttons

Radio buttons are check buttons which have been collected together into a group such that, when one is selected, all other radio buttons in the same group are deselected.

The first radio button in a group should be created using :-

```
radio_button_name = gtk_radio_button_new( NULL );
```

and the group determined using :-

```
group_name = gtk_radio_button_get_group( GTK_RADIO_BUTTON( radio_button_name ));
```

All subsequent radio buttons in the same group should then be created replacing the NULL argument by group_name.

The button to be initially set active should be selected using :-

```
gtk_toggle_button_set_active( GTK_TOGGLE_BUTTON( radio_button_name ));
```

In all other respects they are the same as toggle buttons in that they may be created with a label or a mnemonic, and also emit the “toggled” signal when clicked, which may be connected to a callback function.

9 Entry

The entry widget is a single line text entry widget which, if the entered text is larger than the allocated space, will scroll to the end of the text.

An entry is created using :-

```
entry_name = gtk_entry_name();
```

After entering text, if the Enter key is pressed, the entry emits the “activate” signal which may be connected to a callback function as follows :-

```
g_signal_connect( entry_name, "activate", G_CALLBACK( entry_changed ), NULL );
```

The text in the entry may be accessed using :-

```
text_buffer = gtk_entry_get_text( GTK_ENTRY( entry_name ) );
```

In the following example, text which has been entered will be transferred to a label when the Enter key is pressed.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );
static void entry_changed( GtkWidget*, GtkWidget* );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name;
      GtkWidget   *v_box_name, *h_box_name, *entry_name, *label_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
      v_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
      gtk_container_add( GTK_CONTAINER( window_name ), v_box_name );
      entry_name = gtk_entry_new();
      gtk_box_pack_start( GTK_BOX( v_box_name ), entry_name, FALSE, FALSE, 40 );
      label_name = gtk_label_new( "Label" );
      gtk_box_pack_start( GTK_BOX( v_box_name ), label_name, FALSE, FALSE, 0 );
      g_signal_connect( entry_name, "activate", G_CALLBACK( entry_changed ),
                  ( gpointer )label_name );
      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ), NULL );

      gtk_widget_show_all( window_name );

      gtk_main();
```

```
	return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
	gtk_main_quit();

	return FALSE;
}

static void entry_changed( GtkWidget *widget, GtkWidget *label_name )
{
	const gchar *text_name;

	text_name = gtk_entry_get_text( GTK_ENTRY( widget ) );
	gtk_label_set_text( GTK_LABEL( label_name ), text_name );
}
```

10 Menus

Menu Bars

A complete menu consists of a menu bar on which a collection of menus are placed. Each menu drops down to show a list of menu items, each of which may be activated, by clicking on it, to trigger a callback function.

Assuming that a window has been created to which a vertical box has been added, then a menu bar can be created from :-

```
menu_bar_name = gtk_menu bar new();
```

When the menu bar has been populated with various menus, each of which have had menu items attached, the menu bar may be placed in the vertical box using :-

```
gtk_box_pack_start( GTK_BOX( vertical_box_name ), menu_bar_name, FALSE, FALSE, 0 );
```

Before the menu bar is packed into the vertical box, individual menus are created on the menu bar. Each of these menus is a sub menu which, having been created, may then be populated with menu items, each of which may then be connected to a callback function.

To create a menu use :-

```
menu_name = gtk_menu_item_new_with_label( "Label_text" );
```

The label text is the text that will appear on the menu bar.

Then create a sub menu to contain the menu items using -

```
sub_menu_name = gtk_menu_new();
```

The sub menu is then attached to the menu using :-

```
gtk_menu_item_set_submenu( GTK_MENU_ITEM( menu_name ), sub_menu_name );
```

The sub menu may now be populated with the required menu items.
To create, connect and attach each menu item use the following :-

```
menu_item_name = gtk_menu_item_new_with_label( "menu_item_text" );
g_signal_connect( menu_item_name, "activate", G_CALLBACK( callback_name ), NULL );
gtk_menu_shell_append( GTK_MENU_SHELL( sub_menu_name ), menu_item_name );
```

This may be repeated for each menu item which is to be attached to the sub menu. When all of the menu items have been attached, the sub menu should be placed on the menu bar :-

```
	gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar_name ), sub_menu_name );
```

The whole process may be repeated as many times as necessary to place the required number of sub menus, each with their associated menu items, on the menu bar. When this has been completed, the menu bar should be placed in the vertical box using :-

```
	gtk_box_pack_start( GTK_BOX( vertical_box_name ), menu_bar_name, FALSE, FALSE, 0);
```

The following example shows how to create a basic menu. The various parts are created starting with the menu bar onto which are added submenus. Each submenu is then populated with menu items. The sensitivity menu item shows that a menu item may be made inactive and greyed out by setting the sensitivity to FALSE.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );

static void sensitivity_selected( GtkWidget*, gpointer );
static void about_selected( GtkWidget*, GtkWidget* );

int main( int argc, char *argv[] )
{
	GtkWidget   *window_name;
	GtkWidget   *vertical_box_name, *menu_bar_name, *file, *file_menu_name,
	            *sensitivity, *exit;
	GtkWidget   *help, *help_menu_name, *about ;

	gtk_init( &argc, &argv );

	window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
	gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
	gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
	gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
	vertical_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
	gtk_container_add( GTK_CONTAINER( window_name ), vertical_box_name );

//	Create a new menu bar
	menu_bar_name = gtk_menu_bar_new();

//	Create a submenu with the title "File"
	file = gtk_menu_item_new_with_label( "File" );
	file_menu_name = gtk_menu_new();
	gtk_menu_item_set_submenu( GTK_MENU_ITEM( file ), file_menu_name );

//	Create a menu item called "Sensitivity" and add it to the File sub menu
	sensitivity = gtk_menu_item_new_with_label( "Sensitivity" );
	g_signal_connect( sensitivity, "activate", G_CALLBACK( sensitivity_selected ),
	            NULL );
	gtk_menu_shell_append( GTK_MENU_SHELL( file_menu_name ), sensitivity );

//	Create a menu item called "Exit" and add it to the File sub menu
	exit = gtk_menu_item_new_with_label( "Exit" );
	g_signal_connect( exit, "activate", G_CALLBACK( delete_event ), NULL );
	gtk_menu_shell_append( GTK_MENU_SHELL( file_menu_name ), exit );
```

```
//    Create a second sub menu with the title "Help"
      help = gtk_menu_item_new_with_label( "Help" );
      help_menu_name = gtk_menu_new();
      gtk_menu_item_set_submenu( GTK_MENU_ITEM( help ), help_menu_name );

//    Create a menu item called "About" and add it to the Help sub menu
      about = gtk_menu_item_new_with_label( "About" );
      g_signal_connect( about, "activate", G_CALLBACK( about_selected ),
                  ( gpointer )vertical_box_name );
      gtk_menu_shell_append( GTK_MENU_SHELL( help_menu_name ), about );

//    Add the File and Help sub menus to the menu bar
      gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar_name ), file );
      gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar_name ), help );

//    Add the menu bar to the vertical box in the main window
      gtk_box_pack_start( GTK_BOX( vertical_box_name ), menu_bar_name, FALSE, FALSE, 0 );

      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ), NULL );

      gtk_widget_show_all( window_name );

      gtk_main();

      return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
      gtk_main_quit();

      return FALSE;
}

static void sensitivity_selected( GtkWidget *widget, gpointer data )
{
      gtk_widget_set_sensitive( widget, FALSE );
}

static void about_selected( GtkWidget *widget, GtkWidget *v_box )
{
      GtkWidget *button_name;

      button_name = gtk_button_new_with_label( "Exit" );
      gtk_box_pack_start( GTK_BOX( v_box ), button_name, FALSE, FALSE, 40 );
      g_signal_connect( button_name, "clicked", G_CALLBACK( delete_event ), NULL );
      gtk_widget_show( button_name );
}
```

11 File Chooser

A File Chooser Dialog is a special purpose dialog that allows a hard disk, or other memory device, to be browsed in order to select a file to be opened, or for saving a file.

To create a file chooser use :-

```
dialog_name = gtk_file_chooser_dialog_new( "Title_name", window_name,
            action, first_button_text, first_response, ......, NULL );
```

where :-

"Title_name" is the title of the dialog, or NULL,
window_name is the name of the parent window, or NULL,
action is the operating mode of the dialog which may be :-
GTK_FILE_CHOOSER_ACTION_OPEN or
GTK_FILE_CHOOSER_ACTION_SAVE or
GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER.

This is followed by pairs of button text and button response for as many buttons as are required, the list being terminated by NULL.

File chooser to open file

In the following example, there are two buttons, one for "Cancel" and the other for "Open". The chooser is run using gtk_dialog_run(GTK_DIALOG(dialog_name)) which returns the result of a button being clicked.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name, *vertical_box_name;
      GtkWidget   *dialog_name, *label_name, *opened_label_name;
      gint        res;
      char        *filename;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
      vertical_box_name = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0 );
      gtk_container_add( GTK_CONTAINER( window_name ), vertical_box_name );

      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ),
                  NULL );
```

```
	dialog_name = gtk_file_chooser_dialog_new( ( "Open" ), GTK_WINDOW( window_name ),
		GTK_FILE_CHOOSER_ACTION_OPEN, ( "Cancel" ), GTK_RESPONSE_CANCEL,
		( "Open" ), GTK_RESPONSE_ACCEPT, NULL );
	res = gtk_dialog_run( GTK_DIALOG( dialog_name ) );
	if ( res == GTK_RESPONSE_ACCEPT )
	{
		filename = gtk_file_chooser_get_filename( GTK_FILE_CHOOSER( dialog_name ) );
		label_name = gtk_label_new( filename );
		gtk_box_pack_start( GTK_BOX( vertical_box_name ), label_name, FALSE, FALSE,
				0 );
		opened_label_name = gtk_label_new( "opened" );
		gtk_box_pack_start( GTK_BOX( vertical_box_name ), opened_label_name, FALSE,
				FALSE, 0 );
	}
	gtk_widget_destroy( dialog_name );

	gtk_widget_show_all( window_name );

	gtk_main();

	return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
	gtk_main_quit();

	return FALSE;
}
```

File chooser to save file

The next example shows how to use a file chooser to save a new file. If it is required to overwrite an existing file, it is necessary to add :-

```
	gtk_file_chooser_set_do_overwrite_confirmation( dialog_name, TRUE );
```

and instead of the command to set the current name,

```
	gtk_file_chooser_set_filename( dialog_name, existing filename );
```

Other than the above, the example is very similar to the previous one, but with the action set to :-

GTK_FILE_CHOOSER_ACTION_SAVE

Further details can be obtained from the API documentation.

```
	.
	.
	.
	GtkWidget   *dialog_name;
```

```
gint  res;

.
.
.
dialog_name = gtk_file_chooser_dialog_new( "Save file", parent_window,
            GTK_FILE_CHOOSER_ACTION_SAVE, ( "Cancel" ), GTK_RESPONSE_CANCEL,
                  ( "Save" ), GTK_RESPONSE_ACCEPT, NULL );
gtk_file_chooser_set_do_overwrite_confirmation( GTK_FILE_CHOOSER( dialog_name ),
            TRUE );
if ( user_edited_a_new_document )
      gtk_file_chooser_set_current_name( GTK_FILE_CHOOSER( dialog_name )
                  ( "Untitled document" ) );
else
      gtk_file_chooser_set_filename( GTK_FILE_CHOOSER( dialog_name ),
                  existing_filename );

res = gtk_dialog_run( GTK_DIALOG( dialog ) );
if ( res == GTK_RESPONSE_ACCEPT )
{
      char  *filename;

      filename = gtk_file_chooser_get_filename( chooser );
      save_to_file( filename );
      g_free( filename );
}

gtk_widget_destroy( dialog );
.
.
.
```

12 Drawing Area

A Drawing Area is a blank widget which may be added to any window or dialog to provide an area on which drawing can take place using Cairo drawing commands. The drawing area may also be connected to mouse and button press signals which may be input. An example of the use of mouse signals is shown in the section on events.

When the drawing area is created, a “draw” signal is emitted, which may be connected to a callback to allow items to be drawn on the area. The “draw” signal may also be triggered using :-

```
gtk_widget_queue_draw_area( drawing_area_name, x, y, width, height );
```

where :-

x and y define the top - left corner of the rectangular area to be redrawn,
width and height define the size of the rectangular area.

This is used to redraw only the required portion of the drawing area which is to be updated. Further details of Drawing Areas are given in the API documentation.

Creating a Drawing Area is very straightforward, but the example shows how Cairo commands may be used to draw on the drawing area.

The Drawing Area may be created using :-

```
drawing_area_name = gtk_drawing_area_new();
gtk_widget_set_size_request( drawing_area_name, 300, 200 );
g_signal_connect( G_OBJECT( drawing_area_name ), "draw",
                  G_CALLBACK( draw_callback ), NULL );
gtk_container_add( GTK_CONTAINER( window_name ), drawing_area_name );
```

The draw_callback function must be of the form :-

```
gboolean draw_callback( GtkWidget*, cairo_t*, gpointer );
```

cairo_t is a pointer to an area of memory in which cairo stores all of the information required to create the various items on the drawing area. This pointer is known as the cairo context which, in this case must be connected to the drawing area so that cairo knows where to draw. For a drawing area, this connection is effected by the callback connect command which connects the drawing area to the callback function. The context is used by all of the cairo commands, and is usually given the name cr.

Cairo Commands

Before any lines are drawn, the colour and thickness of the lines should be set using :-

```
cairo_set_source_rgb( cr, r, g, b );
cairo_set_line width( cr, width );
```

Then, to draw a line use :-

```
cairo_move_to( cr, x, y );
cairo_line_to( cr, x, y );
cairo_stroke( cr );
```

where move_to defines the start point of the line, line_to defines the end point, and stroke causes the line to be drawn.

For a rectangle use :-

```
cairo_rectangle( cr, x, y, width, height );
```

to define the rectangle and then either :-

```
cairo_stroke( cr ); or cairo_fill( cr );
```

where stroke will draw a rectangle using the defined colour while fill will fill the rectangle with that colour.

For text it is first necessary to select a font using :-

```
cairo_select_font_face( cr, "Font_name", CAIRO_FONT_SLANT_NORMAL,
                  CAIRO_FONT_WIEGHT_BOLD );
cairo_set_font_size( cr, size );
```

then to write the text :-

```
cairo_move_to( cr, x, y );
cairo_show_text( cr, text_name );
```

More details of various aspects of cairo are available in the cairo API documentation.

Note that, in the following example, #include< cairo.h > has been added to include the cairo headers. The example also shows how the width and height of the text can be used to position text in the drawing area. These are known as cairo text extents. Also, if the example is cut and pasted into an editor, the minus signs may become non-printing characters so it will be necessary to delete them and retype the minus signs.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );
gboolean draw_callback( GtkWidget*, cairo_t*, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name;
      GtkWidget   *drawing_area_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
      drawing_area_name = gtk_drawing_area_new();
      gtk_widget_set_size_request( drawing_area_name, 300, 200 );
      g_signal_connect( G_OBJECT( drawing_area_name ), "draw",
            G_CALLBACK( draw_callback ), NULL );
      gtk_container_add( GTK_CONTAINER( window_name ), drawing_area_name );

      g_signal_connect( window_name, "delete_event",
            G_CALLBACK( delete_event ),NULL );

      gtk_widget_show_all( window_name );

      gtk_main(); //Basic program

      return 0; //Basic program
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
      gtk_main_quit();

      return FALSE;
}

gboolean draw_callback( GtkWidget *widget, cairo_t *cr, gpointer data )
{
      guint width, height;
      cairo_text_extents_t te;
      char  *text;

      width = gtk_widget_get_allocated_width( widget );
      height = gtk_widget_get_allocated_height( widget );

      cairo_set_source_rgb( cr, 0, 0, 0 );
      cairo_select_font_face( cr, "Nimbus Sans L", CAIRO_FONT_SLANT_NORMAL,
                        CAIRO_FONT_WEIGHT_BOLD );
      cairo_set_font_size( cr, 18 );

      text = "Drawing Area";
      cairo_text_extents( cr, text, &te );
      cairo_move_to( cr, ( width - te.width ) / 2, 50 );
      cairo_show_text( cr, text );
```

```
	cairo_move_to( cr, ( width - te.width ) / 2, 75 );
	cairo_line_to( cr, ( width + te.width ) / 2, 75 );
	cairo_stroke( cr );

	cairo_rectangle( cr, (width - te.width ) / 2, 100, te.width, 50 );
	cairo_stroke( cr );

	return FALSE;
}
```

13 Print Operation

In order to carry out any printing, the first requirement is to set up a print operation using :-

```
operation_name = gtk_print_operation_new();
```

This operation must now be connected to two callbacks, namely begin_print and draw_page.

Begin_print is connected using :-

```
g_signal_connect( G_OBJECT( operation_name ), "begin_print",
      G_CALLBACK( begin_print ), NULL );
```

and the callback has the form :-

```
static void begin_print( GtkPrintOperation *operation, GtkPrintContext *context,
                  gpointer data );
```

The begin_print callback can be used to set up the various printer settings, however the draw_page callback can be used to initiate the normal print dialog in which all of the usual settings can be made. In spite of this, the begin_print callback must have an instruction to set the number of pages to be printed using :-

```
gtk_print_operation_set_n_pages( operation, 1 );
```

The rest of the printing is done by the draw_page callback. This callback is connected to the print operation using :-

```
g_signal_connect( G_OBJECT( operation_name ), "draw_page",
      G_CALLBACK( draw_page ), NULL );
```

and the callback is of the form :-

```
static void draw_page( GtkPrintOperation *operation, GtkPrintContext *context,
      gint page_nr, gpointer data );
```

The print operation then must be run using :-

```
res = gtk_print_operation_run( operation_name ),
      GTK_PRINT_OPERATION_ACTION_PRINT_DIALOG,
      GtkWindow( window_name ), NULL );
```

The page to be printed is created using cairo commands in exactly the same way as for drawing on a drawing area except for the context, in this case, being a printer context. Therefore the context is created using :-

```
cr = gtk_print_context_get_cairo_context( context );
```

and the width and height are obtained from :-

```
width = gtk_print_context_get_width( context );
```
and
```
height = gtk_print_context_get_height( context );
```

Cairo Commands

Before any lines are drawn, the colour and thickness of the lines should be set using :-

```
cairo_set_source_rgb( cr, r, g, b );
cairo_set_line width( cr, width );
```

Then, to draw a line use :-

```
cairo_move_to( cr, x, y );
cairo_line_to( cr, x, y );
cairo_stroke( cr );
```

where move_to defines the start point of the line, line_to defines the end point, and stroke causes the line to be drawn.

For a rectangle use :-

```
cairo_rectangle( cr, x, y, width, height );
```

to define the rectangle and then either :-

```
cairo_stroke( cr ); or cairo_fill( cr );
```

where stroke will draw a rectangle using the defined colour while fill will fill the rectangle with that colour.

For text it is first necessary to select a font using :-

```
cairo_select_font_face( cr, "Font_name", CAIRO_FONT_SLANT_NORMAL,
                  CAIRO_FONT_WIEGHT_BOLD );
cairo_set_font_size( cr, size );
```

then to write the text :-

```
cairo_move_to( cr, x, y );
cairo_show_text( cr, text_name );
```

More details of various aspects of cairo are available in the cairo API documentation.

Note that, in the following example, #include< cairo.h > has been added to include the

cairo headers. The example also shows how the width and height of the text can be used to position text in the drawing area. These are known as cairo text extents. Also, if the example is cut and pasted into an editor, the minus signs may become non-printing characters so it will be necessary to delete them and retype the minus signs.

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

static gboolean delete_event( GtkWidget*, GdkEvent*, gpointer );
static void print_callback( GtkWidget*, GtkWidget* );
static void begin_print( GtkPrintOperation*, GtkPrintContext*, gpointer );
static void draw_callback( GtkPrintOperation*, GtkPrintContext*, gint, gpointer );

int main( int argc, char *argv[] )
{
      GtkWidget   *window_name;
      GtkWidget   *print_button_name;

      gtk_init( &argc, &argv );

      window_name = gtk_window_new( GTK_WINDOW_TOPLEVEL );
      gtk_window_set_title( GTK_WINDOW( window_name ), "Main" );
      gtk_window_set_default_size( GTK_WINDOW( window_name ), 300, 200 );
      gtk_window_move( GTK_WINDOW( window_name ), 650, 400 );
      print_button_name = gtk_button_new_with_label( "Print" );
      g_signal_connect( G_OBJECT( print_button_name ), "clicked",
                  G_CALLBACK( print_callback ), ( gpointer ) window_name );
      gtk_container_add( GTK_CONTAINER( window_name ), print_button_name );

      g_signal_connect( window_name, "delete_event", G_CALLBACK( delete_event ),
                        NULL );

      gtk_widget_show_all( window_name );

      gtk_main();

      return 0;
}

static gboolean delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
      gtk_main_quit();
      return FALSE;
}

static void print_callback( GtkWidget *widget, GtkWidget *window )
{
      gint  res;
      GtkPrintOperation *print_operation_name;

      print_operation_name = gtk_print_operation_new();
      g_signal_connect( G_OBJECT( print_operation_name ), "begin_print",
                        G_CALLBACK( begin_print ), NULL );
      g_signal_connect( G_OBJECT( print_operation_name ), "draw_page",
                        G_CALLBACK( draw_callback ), NULL );
      res = gtk_print_operation_run( ( print_operation_name ),
```

```
                GTK_PRINT_OPERATION_ACTION_PRINT_DIALOG, GTK_WINDOW( window ), NULL );
}

static void begin_print( GtkPrintOperation *operation, GtkPrintContext *context,
                            gpointer data )
{
      gtk_print_operation_set_n_pages( operation, 1 );
}

static void draw_callback( GtkPrintOperation *operation, GtkPrintContext *context,
                            gint page_number, gpointer data )
{
      cairo_t     *cr;
      guint width, height;
      cairo_text_extents_t te;
      char  *text;

      cr = gtk_print_context_get_cairo_context( context );
      width = gtk_print_context_get_width( context );
      height = gtk_print_context_get_height( context );

      cairo_set_source_rgb( cr, 0, 0, 0 );
      cairo_select_font_face( cr, "Numbus Sans L", CAIRO_FONT_SLANT_NORMAL,
                            CAIRO_FONT_WEIGHT_BOLD );
      cairo_set_font_size( cr, 18 );

      text = "Drawing Area";
      cairo_text_extents( cr, text, &te );
      cairo_move_to( cr, ( width - te.width ) / 2, 50 );
      cairo_show_text( cr, text );

      cairo_move_to( cr, ( width - te.width ) / 2, 75 );
      cairo_line_to( cr, ( width + te.width ) / 2, 75 );
      cairo_stroke( cr );

      cairo_rectangle( cr, (width - te.width ) / 2, 100, te.width, 50 );
      cairo_stroke( cr );

}
```