Lesson 4: An Assortment of Input Widgets

1 Introduction

So far, we have not encountered any widgets that let the user provide input to the application, with the exception of the `GtkButton` widgets and, in a broad sense, the `GtkEventBox`. Applications with graphical user interfaces provide many different means for the user to provide input: various types of text entry widgets, buttons that bring up dialog boxes that let the user choose text style (fonts, size, etc.), colors, or files to open, controls such as checkboxes, radio buttons, combo boxes, slide controls, and spin buttons that let the user choose from among a set of finite alternatives.

This lesson covers several different types of input widgets, including several buttons, the text entry widget, and scales. Because some of these widgets require ancillary background knowledge and some of the demonstration programs have a few new features, the lesson also introduces concepts related to GLib timers, fonts, color representation, image file formats, and file filters.

2 Buttons

There are a variety of buttons in the GTK+ library. The following classes are derived from `GtkButton`:

- `GtkToggleButton`
- `GtkColorButton`
- `GtkFontButton`
- `GtkLinkButton`
- `GtkOptionMenu` (deprecated, use `GtkComboBox` instead)
- `GtkScaleButton`

Toggle buttons are used for creating radio buttons and checkboxes. Color buttons are used for selecting colors, and font buttons, for selecting fonts. Link buttons are used for connecting to a URL. Scale buttons pop up a scale widget. This kind of widget is commonly used for volume controls in multimedia applications; the `GtkVolumeButton` is derived from the `GtkScaleButton` specifically for this purpose. It may not seem as if menus can be called buttons, but that is what they are. A menu is a type of control that lets a user choose from a set of alternative actions. A `GtkOptionMenu` is a widget that lets a user choose from among several options. It has been deprecated since Gtk+ 2.4 and replaced by the `GtkComboBox`.

A `GtkComboBox` derives directly from `GtkBin`. A `GtkComboBox` is a widget that allows the user to choose from a list of valid choices. The widget consists of a text entry box and a pull down menu from which the user can select one of a set of predefined entries. It also allows the user to type a different option into the text box. The style in which the selected value is displayed, and the style of the pop-up list is determined by the current theme. We will cover combo boxes later.

We will begin by going over a few things about basic buttons that have not been covered yet.

2.1 Stock Items

Stock items represent commonly-used menu or toolbar items such as "`Open`" or "`About`". Each stock item is identified by a stock ID, which is a string, such as "`gtk-about`". GTK+ defines macros to represent these strings. For example

```
#define GTK_STOCK_ABOUT "gtk-about"
```

Applications should never use the actual strings; they should use the macros, which are pretty easy to guess. Applications can register their own stock items in addition to those built into GTK+. A stock ID is associated with a `GtkStockItem`, which contains the user-visible label, keyboard accelerator, among other things. When you use a stock item, you have access to these accelerators and labels, which include icons.

A good reason to use stock items when possible is to make your buttons, menu items, and toolbars conform to users' expectations. They will also change their appearance depending upon the themes in the user's environment.

Buttons can be created with stock items as their labels. The method is

```
GtkWidget *    gtk_button_new_from_stock      (const gchar *stock_id);
```

The macro name can be used instead of the string.

2.1.1 Example

The following program illustrates the use of buttons with stock items. It also introduces the `GtkButtonBox` class, which is a convenient container for groups of independent buttons (as opposed to radio buttons) that derives from `GtkBox`.

```
int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *button_box;
    GtkWidget *frame;


    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Stock Buttons");
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    // Control window width but let the height grow as needed
    gtk_widget_set_size_request (window, 250, -1);

    // Create a labeled frame and a vertical button box;
    frame = gtk_frame_new("Some Stock Buttons");
    button_box = gtk_vbutton_box_new ();

    // Set the attributes of the button box: buttons spread
    // evenly and have a spacing of 15 pixels between them,
    // with a 10 pixel border
    gtk_button_box_set_layout (GTK_BUTTON_BOX (button_box),
                               GTK_BUTTONBOX_SPREAD);
    gtk_box_set_spacing (GTK_BOX (button_box), 15);
    gtk_container_set_border_width (GTK_CONTAINER (button_box), 10);

    // Add the button box to frame and the frame to window
    gtk_container_add (GTK_CONTAINER (frame), button_box);
    gtk_container_add (GTK_CONTAINER (window), frame);


    // Create a bunch of buttons from stock;
    button = gtk_button_new_from_stock (GTK_STOCK_ABOUT);
    gtk_container_add (GTK_CONTAINER (button_box), button);

    button = gtk_button_new_from_stock (GTK_STOCK_OK);
    gtk_container_add (GTK_CONTAINER (button_box), button);

    button = gtk_button_new_from_stock (GTK_STOCK_CANCEL);
    gtk_container_add (GTK_CONTAINER (button_box), button);

    button = gtk_button_new_from_stock (GTK_STOCK_HELP);
    gtk_container_add (GTK_CONTAINER (button_box), button);

    button = gtk_button_new_from_stock (GTK_STOCK_PRINT);
    gtk_container_add (GTK_CONTAINER (button_box), button);

    // Don't bother with giving the buttons any actions;
    // just the close box
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_main_quit), NULL);

    // Show the widgets
    gtk_widget_show_all (window);

    /*
       In earlier versions of GTK, the button icons displayed
       automatically. They changed things on us. Now the Gnome
       settings by default will not display button icons.
       Each app has to change the setting to display them. To
       do this, you have to retrieve GTK's default settings into a
       GtkSettings object, and set the "gtk-button-images" property
       to TRUE.
     */
    GtkSettings *default_settings = gtk_settings_get_default();
    g_object_set(default_settings, "gtk-button-images", TRUE, NULL);

    gtk_main ();
    return 0;
}
```

Notice the last few lines of the program. To guarantee that the icons will display, your program has to explicitly set the `gtk-button-images` property to `TRUE` in GTK's default settings. This is done by getting a pointer to a `GtkSettings` object and setting the "`gtk-button-images`" value to `TRUE` with the `g_object_set()` call.

A GtkButtonBox can be horizontal or vertical, and it has two subclasses for this purpose. This program creates a vertical button box using

```
gtk_GtkWidget *    gtk_vbutton_box_new           (void);
```

Once it is created, the methods of the parent class are used, analogous to how horizontal and vertical boxes are used. The first step is to set how the box will be laid out, using

```
void        gtk_button_box_set_layout          (GtkButtonBox *widget,
                                                 GtkButtonBoxStyle layout_style);
```

The `GtkButtonBoxStyle` is an enumeration that defines various ways to distribute the buttons within the box

```
typedef enum
{
  GTK_BUTTONBOX_DEFAULT_STYLE,   // whatever the default is
  GTK_BUTTONBOX_SPREAD,          // evenly spread
  GTK_BUTTONBOX_EDGE,            // placed at the edges
  GTK_BUTTONBOX_START,           // grouped towards the start
  GTK_BUTTONBOX_END,             // grouped towards the end
  GTK_BUTTONBOX_CENTER           // centered
} GtkButtonBoxStyle;
```

The program above spreads the buttons evenly in the space allotted to the box. The `gtk_box_set_spacing()` method is used to add a 15 pixel spacing between the buttons in the above program. The buttons are added to the box using the `gtk_container_add()` method.

2.2 Toggle Buttons

Toggle buttons are not used very much, but they are a parent class of other, more useful buttons, namely check buttons and radio buttons, and they are therefore worthy of discussion. A toggle button is always in one of two states; a click switches it from one state to the other. Visually, they will appear depressed in one state and "popped up" in the other.

To create a new toggle button, use one of three methods:

```
GtkWidget *gtk_toggle_button_new( void );
GtkWidget *gtk_toggle_button_new_with_label( const gchar *label );
GtkWidget *gtk_toggle_button_new_with_mnemonic( const gchar *label );
```

These are self-explanatory at this point, assuming you have read the previous chapters.

(to be filled in more here)

2.3 Check Buttons

Check buttons are derived from toggle buttons, but look a little different. They are what I used to call check boxes, a small square within which you can click, with a label next to them. These are used for toggling the state of a two-state property in an application. They are really just a different way to present the toggle button.

(to be continued ...)

2.4 Radio Buttons

Radio buttons are similar to check buttons except they are grouped so that only one may be selected/depressed at a time. They are suitable for making a choice among a small number of alternatives.

(to be continued ...)

3 The Text Entry Widget

A `GtkEntry` widget is a very powerful, single-line text entry widget, endowed with a large number of methods. It displays a box into which the user can enter and edit text. If the entered text is longer than the width of the widget, the widget will scroll so that the cursor position is visible.

An entry is created with

```
GtkWidget *    gtk_entry_new                    ( void);
```

The next function alters the text which is currently within the Entry widget.

```
void           gtk_entry_set_text               ( GtkEntry *entry,
                                                  const gchar *text );
```

The function `gtk_entry_set_text()` sets the contents of the Entry widget, replacing the current contents. Note that the class Entry implements the Editable interface (yes, GObject supports Java-like interfaces) which contains some more functions for manipulating the contents.

The text entry widget has the ability to hide the entered text with an *invisibility character*. As the user enters characters, this character is displayed in the widget instead of what the user types. This feature makes it possible to put the widget into "password mode". To set the visibility property, use

```
void           gtk_entry_set_visibility         ( GtkEntry *entry,
                                                  gboolean visible);
```

By default, GTK+ picks the best invisible character that is available in the current font, but it can be changed with

```
void           gtk_entry_set_invisible_char     ( GtkEntry *entry,
                                                  gunichar ch);
```

The gunichar type is GLib's Unicode character representation; to be precise, it is type which can hold any UTF-32 or UCS-4 character code.

(to be continued ...)

4 The GtkAdjustment Object

A `GtkAdjustment` is one of those few GTK objects that is not a widget. It represents a discrete value which has an associated lower and upper bound, together with step and page increments, and a page size. It is important because it is the object that makes spin buttons, scales, sliders, progress bars, and scroll bars possible.

Step and page increments define how the value changes in small "steps" and in large "page" jumps. Think about the different parts of a scrollbar to get an intuition for this. Steps are the changes that take place when the arrows at either end of the bar are clicked. Page increments are the changes resulting from clicking in the "trough" of the scrollbar with the left mouse button, or using the Page Up or Page Down key on the keyboard. The page size is not used by all widgets; it is used in different ways by different widgets. For example, for an adjustment object for a scrollbar, it is the size of the visible area that is being scrolled in the same direction as the scroll. A `GtkAdjustment` object does not update its value itself. Instead it is left up to the owner of the `GtkAdjustment` to control the value.

A `GtkAdjustment` allows widgets to coordinate their actions through the use of the two signals that they emit – "value_changed" and "changed". When the value of an adjustment is changed, the "value-changed" signal is emitted. When the adjustment is reconfigured, by changing a bound or the page size, for instance, the "changed" signal is emitted. A reconfiguration can occur if the size of a document changes in a widget controlled by a scrollbar; the upper bound increases. If the window is resized, the page_size parameter is changed. Multiple widgets can react to the signals from a single adjustment, so that while a scrollbar moves, so does the content area of the widget that it controls.

You as a programmer do not change the value of the adjustment directly. An adjustment is always part of some widget called its *owner*. The owner is what typically calls `gtk_adjustment_value_changed()` and `gtk_adjustment_changed()`. For example, when a user slides the thumb of a scale, the scale widget sets the adjustment's new value using `gtk_adjustment_value_changed()` after which the "value-changed" signal is emitted on the adjustment.

Let us examine the functions for working with adjustments.

To create a new adjustment, use

```
GtkObject *     gtk_adjustment_new              (gdouble value,
                                                 gdouble lower,
                                                 gdouble upper,
                                                 gdouble step_increment,
                                                 gdouble page_increment,
                                                 gdouble page_size);
```

but cast the result to a `GtkAdjustment*` because a `GtkObject` is a resource hog and this type is deprecated in GTK+3 anyway. The parameters have the following meanings:

`value`	the initial value.
`lower`	the minimum value.
`upper`	the maximum value.
`step_increment`	the step increment.
`page_increment`	the page increment.
`page_size`	the page size.

The interpretation of these values depends upon the widget that uses them. For example:

```
GtkAdjustment adjustment;
adjustment = (GtkAdjustment *) gtk_adjustment_new (1.0, 1.0, 100.0, 1.0, 10.0, 0);
```

This creates an adjustment with range [1.0, 100.0] and initial value 1.0, with steps of 1.0 and 10.0.

You can set and retrieve the adjustment's values with

```
void            gtk_adjustment_set_value        (GtkAdjustment *adjustment,
                                                 gdouble value);
```

and

```
gdouble         gtk_adjustment_get_value        (GtkAdjustment *adjustment);
```

There are accessor and mutator methods for each of the individual properties other than the `value` of the `GtkAdjustment` object, i.e., lower, upper, and so on. The accessors are of the form

```
gdouble         gtk_adjustment_get_xxx          (GtkAdjustment *adjustment);
```

and the mutators, of the form

```
void            gtk_adjustment_set_xxx          (GtkAdjustment *adjustment,
                                                 gdouble xxx);
```

where the `xxx` is replaced by the property name. Because each time a property is set, a "changed" signal results, if you plan on changing several at once, it is better to use

```
GtkObject *     gtk_adjustment_configure        (GtkAdjustment *adjustment,
                                                 gdouble value,
                                                 gdouble lower,
                                                 gdouble upper,
                                                 gdouble step_increment,
                                                 gdouble page_increment,
                                                 gdouble page_size);
```

which causes emission of a single "changed" signal. There are no uses of adjustments outside of owning widgets, so we will move on to spin buttons to illustrate how they are used.

5 The Spin Button

A `GtkSpinButton` is actually a subclass of `GtkEntry`. It is a more secure way to allow users to enter numeric values than a `GtkEntry`, because if it is set up properly, it handles all input validation. It allows the user to click on one of two arrows to increment or decrement the displayed value within a fixed range and with fixed steps. The user can also type a value, but the spin button can be set up so that the value can be checked to ensure it is in a given range.

Spin buttons are based upon adjustments. In fact they are created by specifying an adjustment, so the first step is usually to create the adjustment and then create the spin button. A spin button is created with

```
GtkWidget *     gtk_spin_button_new             (GtkAdjustment *adjustment,
                                                 gdouble climb_rate,
                                                 guint digits);
```

where climb_rate controls the acceleration of the button and digits is the number of decimal digits to display. The climb_rate must be between 0.0 and 1.0. The larger the number, the faster the acceleration.

There is a convenience function for creating spin buttons that uses a default adjustment. Consult the API documentation for information about it.

A spin button's parameters can be changed at any time with `gtk_spin_button_configure()`:

```
void            gtk_spin_button_configure        (GtkSpinButton *spin_button,
                                                   GtkAdjustment *adjustment,
                                                   gdouble climb_rate,
                                                   guint digits);
```

There are various functions setting the button's properties, including

```
void            gtk_spin_button_set_digits       (GtkSpinButton *spin_button,
                                                   guint digits);
void            gtk_spin_button_set_increments   (GtkSpinButton *spin_button,
                                                   gdouble step,
                                                   gdouble page);
void            gtk_spin_button_set_range        (GtkSpinButton *spin_button,
                                                   gdouble min,
                                                   gdouble max);
```

and corresponding functions for accessing their values. To retrieve a pointer to the button's current adjustment object, use

```
GtkAdjustment* gtk_spin_button_get_adjustment   (GtkSpinButton *spin_button);
```

and to replace the adjustment being used by a spinner:

```
void            gtk_spin_button_set_adjustment   (GtkSpinButton *spin_button,
                                                   GtkAdjustment *adjustment);
```

There are two ways to retrieve the spin button's current value. One returns a `gdouble` and the other returns it as an integer, rounding using the usual rules:

```
gdouble         gtk_spin_button_get_value        (GtkSpinButton *spin_button);
gint            gtk_spin_button_get_value_as_int (GtkSpinButton *spin_button);
```

A spin button emits a few signals but the most important one is the "value-changed" signal, which is emitted whenever the value is changed. It would be in the callback for this signal that you would use the current value to update whatever data the spin button controlled.

We will start with a simple program to illustrate the basic functionality.

```
Listing spinbutton_demo1.c
// Include directives omitted to save space
void show_value (GtkSpinButton *spinner,
                 gpointer user_data)
{
    gint value = gtk_spin_button_get_value_as_int (spinner);
    g_printf("Current value is %d\n", value);
}

int  main (int argc, char *argv[])
{
    GtkWidget      *window;
    GtkWidget      *spinner;
    GtkAdjustment  *adjustment;
    gtk_init (&argc, &argv);

    window = gtk_window_new           (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title              (GTK_WINDOW (window),
                                           basename(argv[0]) );
    gtk_widget_set_size_request       (window, 250, 100);
    gtk_container_set_border_width    (GTK_CONTAINER (window), 5);

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

    // Create an adjustment that ranges from 0 to 100 with initial
    // value  50,  a step of 1 and a page increment of 5.
    adjustment = (GtkAdjustment *) gtk_adjustment_new (50.0, 0.0,
                                              100.0, 1.0, 5.0, 0.0);

    // Create a spin button with no decimal places
    spinner = gtk_spin_button_new (adjustment, 0.5, 2);
    g_signal_connect (G_OBJECT (spinner), "value-changed",
                      G_CALLBACK (show_value), NULL);

    gtk_container_add (GTK_CONTAINER (window), spinner);
    gtk_widget_show_all (window);
    gtk_main();
    return 0;
}
```

The spin button has a climb_rate of 0.5 and displays 2 decimal digits. When the value is changed, the `show_value()` callback is executed. This gets the value of the spinner as an integer and displays it on the terminal. When you run this program you will observe a few things.

1. If you hold either arrow key down to accelerate, not all values will be printed on the terminal window. This is because the signals arrive faster than they can be handled and some are thrown away.
2. If you type a number outside of the range of the adjustment, it will replace it with the nearest valid value.
3. If you type a number with decimal digits, it will accept it, even though it is not a value that can be generated using only the arrow buttons, and if you then use the arrow buttons, the sequence will be off by the decimal amount you entered.
4. You can enter non-numeric data. If you do, the current value will be set to the smallest value in its range.
5. The width of the spinner is much larger than it needs to be. You can put it into an hbox to control its size, and add the hbox into the window instead.

You can prevent the spinner from accepting numbers that are not valid, i.e., that cannot be generated by the arrows alone, by setting the boolean "snap-to-ticks" property to `TRUE`. By default it is `FALSE`. There is a function to do this, so that you do not have to use the `g_object_set()` function:

```
void           gtk_spin_button_set_snap_to_ticks(GtkSpinButton *spin_button,
                                                 gboolean snap_to_ticks);
```

When this property is set to `TRUE`, the spinner will round invalid, but in-range, values to the nearest "tick" value.

You may not want the spin button to replace the current value if the user types a number that is out of range. You may want it to just ignore the out-of-range value and remain at the current value. You can modify its update policy to do this. The update policy has two values:

`GTK_UPDATE_ALWAYS` When the button refreshes itself after a value is typed, it always displays the new value, coercing it to a bound if necessary.

`GTK_UPDATE_IF_VALID` When the button refreshes itself after a value is typed, it only replaces the current value if the new value is within the range of the underlying adjustment.

By default it is set to `GTK_UPDATE_ALWAYS`.

You can set it to `GTK_UPDATE_IF_VALID` with

```
void          gtk_spin_button_set_update_policy (GtkSpinButton *spin_button,
                                                  GtkSpinButtonUpdatePolicy policy);
```

passing `GTK_UPDATE_IF_VALID` as the second parameter.

You can prevent the user from entering non-numeric data by setting the button's "numeric" property to `FALSE`. The method that does this is

```
void          gtk_spin_button_set_numeric       (GtkSpinButton *spin_button,
                                                  gboolean numeric);
```

When this property is set, no non-numeric characters will appear in the spinner's textbox.

6 Scales

Scale widgets, defined by the `GtkScale` class, are widgets that let the user choose a numeric value by sliding a *slider* along a track called a *trough*, similar to brightness or volume controls or the playback slider in video or audio media players. Dragging the slider with the pointer moves it back and forth within the trough. Clicking in the trough advances the slider towards the location of the click, either completely, or by a designated amount, depending on which mouse button is used. There are both horizontal and vertical scales. Figure 1 shows an example of a horizontal scale. Horizontal scales belong to `GtkHScale` and vertical scales, to `GtkVScale`, both derived from `GtkScale`.

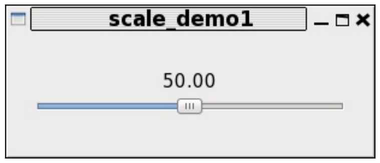

Figure 1: Horizontal scale

6.1 Creating and Adjusting Scales

Vertical and horizontal scales can be created with the following methods:

```
GtkWidget *    gtk_vscale_new                  ( GtkAdjustment *adjustment );
GtkWidget *    gtk_vscale_new_with_range       ( gdouble min,
                                                 gdouble max,
                                                 gdouble step );
GtkWidget *    gtk_hscale_new                  ( GtkAdjustment *adjustment );
GtkWidget *    gtk_hscale_new_with_range       ( gdouble min,
                                                 gdouble max,
                                                 gdouble step );
```

Because horizontal and vertical scales are essentially the same except for orientation, we will limit the discussion to horizontal scales. As with spin buttons, the different between the two constructors is that the `gtk_hscale_new_with_range()` creates an adjustment implicitly with the values of `min`, `max`, and `step` provided in the call, whereas `gtk_hscale_new()` lets the programmer supply the adjustment explicitly.

You have control of several properties of the scale widget. By default, a scale widget displays the current value of the adjustment as the slider position changes. You can turn this on or off with

```
void           gtk_scale_set_draw_value        ( GtkScale *scale,
                                                 gboolean draw_value );
```

The default number of decimal digits for the displayed number is 1. This can be adjusted with

```
void           gtk_scale_set_digits            ( GtkScale *scale,
                                                 gint digits);
```

This function affects not just the widget, but the adjustment itself. If current value of the adjustment is retrieved, it will match the value that is displayed. A consequence of this is that, if the same adjustment is embedded in multiple scales, changing the number of displayed digits in one scale changes them in the others that share the adjustment.

One can also control where the current value of the scale is drawn. The choices are above, below, to the left, and to the right of the trough, using

```
void           gtk_scale_set_value_pos         ( GtkScale *scale,
                                                 GtkPositionType pos);
```

where `pos` can be one of `GTK_POS_LEFT`, `GTK_POS_RIGHT`, `GTK_POS_TOP`, and `GTK_POS_BOTTOM`. If the current value is positioned on the side of the trough, as opposed to at either end, then as the slider moves, the number will move with it.

Another property that scales can have are labeled tick marks along the trough. You can use

```
void           gtk_scale_add_mark              ( GtkScale *scale,
                                                 gdouble value,
                                                 GtkPositionType position,
                                                 const gchar *markup);
```

to put a tick mark above or below a horizontal scale (or to the left or right of a vertical scale.) The `value` parameter determines where along the scale to position the mark. It should be a valid value within the range of the adjustment object. The `markup` parameter is a string that may include Pango markup. The `position` parameter is one of the values mentioned above. For example, to put the tick marks below a horizontal scale, use `GTK_POS_BOTTOM`. If you supply a position type that is not along the trough, it will default to the top for horizontal scales, and to the left for vertical scales. You can clear the marks with `gtk_scale_clear_marks()`.

The following listing shows the basic steps in creating a scale. This scale does not react to changes in the slider. That will come afterwards.

```
Listing scale_demo1.c
int main (int argc,  char *argv[])
{
    GtkWidget *window;
    GtkWidget *scale;
    GtkObject *adjustment;

    gtk_init (&argc, &argv);

    window = gtk_window_new  (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title  (GTK_WINDOW (window), basename(argv[0])  );
    gtk_widget_set_size_request  (window, WINWIDTH, WINHEIGHT);
    gtk_container_set_border_width  (GTK_CONTAINER (window), 20);

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

    adjustment = gtk_adjustment_new  (50.0, 0.0, 100.0, 0.1, 10.0, 0.0);
    scale      = gtk_hscale_new  (GTK_ADJUSTMENT (adjustment));

    // Set the number of decimal places to display
    gtk_scale_set_digits  (GTK_SCALE (scale), 2);

    // Set the position of the value with respect to the trough
    gtk_scale_set_value_pos  (GTK_SCALE (scale), GTK_POS_TOP);
    gtk_scale_set_draw_value  (GTK_SCALE (scale), TRUE);
    gtk_container_add  (GTK_CONTAINER(window), scale);

    gtk_widget_show_all(window);
    gtk_main  ();

    return 0;
}
```

6.2 Range Widgets

To react to the changes in value as the slider is moved, you need to connect a callback to a signal from the parent class of `GtkScale`, which is `GtkRange`. `GtkRange` is the base class for the classes of widgets that visualize adjustment objects, which also includes scrollbars. Scales and scrollbars are similar in function and implementation. Scrollbars, which are defined by the `GtkScrollbar` class, are also divided into horizontal and vertical subclasses, `GtkHScrollbar` and `GtkVScrollbar`. All range widgets, including scrollbars, contain a trough and a slider.

`GtkRange` contains the signals for monitoring the parameters of the adjustment and also provides properties and methods for influencing the sensitivity of the "steppers". Range widgets use the embedded adjustment object to calculate the length of the slider and its position within the trough. As the user manipulates the slider, the range widget will change the value of the adjustment.

The only signal that you need to monitor for basic applications is the “value-changed” signal, which is emitted on the widget whenever the slider moves. The following listing shows a simple example of handling this signal.

```
Listing scale_demo2.c
```

```
void on_value_changed          ( GtkRange  *scale,
                                 GtkLabel  *label )
{
    gchar     *buf;
    buf = g_strdup_printf ("Current volume: %d",
                           (gint) gtk_range_get_value(scale));
    gtk_label_set_text (label, buf);
    g_free (buf);

}

int main (int argc,  char *argv[])
{
    GtkWidget *window;
    GtkWidget *scale;
    GtkObject *adjustment;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *label1;
    GtkWidget *label2;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), basename(argv[0]) );
    gtk_widget_set_size_request (window, WINWIDTH, WINHEIGHT);
    gtk_container_set_border_width (GTK_CONTAINER (window), 20);

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

    vbox = gtk_vbox_new(FALSE, 0 );
    gtk_container_add (GTK_CONTAINER(window), vbox);

    hbox = gtk_hbox_new(FALSE, 0 );

    label1       = gtk_label_new("Volume Level:  ");
    gtk_misc_set_alignment(GTK_MISC(label1), 0, 1);
    gtk_box_pack_start (GTK_BOX (hbox), label1, FALSE, TRUE, 0);

    label2 = gtk_label_new("Current volume: 5");
    gtk_misc_set_alignment(GTK_MISC(label2), 1, 1);
    gtk_box_pack_end (GTK_BOX (hbox), label2, FALSE, TRUE, 0);
    gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 0);

    adjustment = gtk_adjustment_new (5.0, 0.0, 10.0,
                                     1.0, 1.0, 0.0);
    scale      = gtk_hscale_new (GTK_ADJUSTMENT (adjustment));
    gtk_scale_set_digits (GTK_SCALE (scale), 0);
    gtk_scale_set_value_pos (GTK_SCALE (scale), GTK_POS_TOP);
    gtk_scale_set_draw_value (GTK_SCALE (scale), TRUE);
    gtk_box_pack_start (GTK_BOX (vbox), scale, FALSE, TRUE, 0);

    // Connect the value-changed signal to a callback that
    // updates the second label
    g_signal_connect (G_OBJECT (scale), "value_changed",
                      G_CALLBACK (on_value_changed),
                      (gpointer) label2);

    gtk_widget_show_all(window);
    gtk_main ();

    return 0;
}
```

The `GtkRange` class provides some fairly sophisticated functionality in addition to the basics that you need for simple applications. One property is the "fill-level". If you have downloaded streaming media online, you have probably seen scales in the media players that contain a slider as well as an indicator that lets you know how much has been downloaded so far. In GTK+, you can display how much of a file has downloaded by setting the "fill-level" of the scale. You can turn on or off the widget's display of the "fill-level", and you can even prevent the user from positioning the slider beyond the current fill level. The three functions that perform these actions respectively, are:

```
void    gtk_range_set_fill_level             ( GtkRange *range,
                                               gdouble fill_level);
void    gtk_range_set_show_fill_level        ( GtkRange *range,
                                               gboolean show_fill_level);
void    gtk_range_set_restrict_to_fill_level ( GtkRange *range,
                                               gboolean restrict_to_fill_level);
```

You set the fill level to zero initially, and add a timeout to a main event loop. The timeout can be used to call a function to check on the progress of a download and update the fill level as it progresses. A timeout is added to the main event loop with the GLib function

```
guint       g_timeout_add                     ( guint interval,
                                                GSourceFunc function,
                                                gpointer data);
```

The interval is expressed in milliseconds. A `GSourceFunc` is a function whose prototype is

```
gboolean    (*GSourceFunc)                    ( gpointer user_data);
```

The function must return `TRUE` or `FALSE`. If it returns `TRUE`, the timeouts continue to interrupt the main event loop and the function is called again. If it returns `FALSE`, the timeout is removed. The data passed to `g_timeout_add()` in the third argument is passed to the `GSourceFunc` when it is called. Thus, to have a function `advance_fill`() called every half-second, and update the fill value in a `GtkHScale` named `scale`, we would add the instruction

```
g_timeout_add( 500, (GSourceFunc) advance_fill, scale);
```

to the main program, and define the `GSourceFunc`

```
gboolean advance_fill ( GtkRange *scale )
{
    gdouble   fill , max;
    gchar     *buf;
    GtkLabel *label;
    GtkAdjustment  *scale_adj;

    scale_adj = gtk_range_get_adjustment(scale);
    max       = gtk_adjustment_get_upper(scale_adj);
    fill      = gtk_range_get_fill_level(scale);
    label     = g_object_get_data(G_OBJECT(scale), "label");

    if ( fill < max ) {
        fill = fill + INTERVAL;
        gtk_range_set_fill_level( scale , fill );
        buf = g_strdup_printf ("Remaining: %4.2f\n",max - fill );
        gtk_label_set_text (label, buf);
        g_free (buf);
        return TRUE;
    }
    else
        return FALSE;
}
```

See `scale_demo3.c` for the complete program.

7 Selection Buttons

There are a few convenient buttons built into GTK+. These include the `GtkColorButton` and `GtkFontButton`, which are true buttons, and the `GtkFileChooserButton`, which is not derived from the button class but is a direct child of the `GtkHBox` class, as are `GtkInfoBar` and `GtkStatusBar`. All of these widgets fall into the category of *selection widgets* because they let the user select something or other from the entire domain of selectable things.

The color button can be used to select a color, which then becomes the currently selected color, out of the set of all possible representable colors. The font button lets the user pick a font, which similarly becomes the currently selected font, out of all possible fonts available to the user. The file chooser button lets the user choose a file or a folder, depending on how it is configured, either from all possible files or from a filtered subset of files.

Each of these buttons, when clicked, opens a dialog box that implements the selection capability. The dialog boxes that are opened in response to the color and font buttons are specialized to their particular tasks, and are not customizable or accessible to the programmer. The file chooser dialog box is a very powerful widget that is essentially a file browser. In addition, it implements the `GtkFileChooser` interface, which means that all of the functionality of this interface is available to the programmer by casting this button into a `GtkFileChooser` object.

These buttons are convenience buttons in the sense that their functionality exists in more general forms in other widgets.They exist mostly for very simple, or "quick and dirty" applications. Usually you will create a proxy such as a menu item or toolbar button to display a dialog box, and handle that dialog box's responses with greater control. We will describe the buttons briefly here, and at the same time introduce some of the more advanced topics that will be covered later.

7.1 The Font Button

The GtkFontButton lets the user choose a font. It displays the currently selected font as its label, and when clicked, opens a font selection dialog box to change the font. It has methods that control whether the font label is displayed using the font and/or the font size. It also has methods to set or get the title of the dialog box that is displayed when the button is clicked.

To create a font button use either of the two constructors:

```
GtkWidget *    gtk_font_button_new              ( void);
GtkWidget *    gtk_font_button_new_with_font    ( const gchar *fontname);
```

the difference being that the latter is initialized to contain the given fontname. Methods to set whether or not to use the font or size in the button label are:

```
void           gtk_font_button_set_use_font     ( GtkFontButton *font_button,
                                                  gboolean use_font);
void           gtk_font_button_set_use_size     ( GtkFontButton *font_button,
                                                  gboolean use_size);
```

The font button emits just one signal: "font-set", which is emitted when the user clicks the OK button in the font selection dialog box. When the user has selected a font, its name can be retrieved using

```
const gchar *  gtk_font_button_get_font_name    ( GtkFontButton *font_button);
```

Therefore, this function should be called in the signal handler for the "font-set" signal.

This signal is emitted only when the user selects the font; it is not emitted as the user changes the font within the dialog box. The font within the dialog box is a property of the dialog box itself, not the font button. If you wanted the program to react to changes in the value of that property as the user changed the font, you would have to connect a signal handler to the notify::font-name signal on the dialog. But you do not have access to that dialog when using the GtkFontButton. If your program needs to react to changes outside of the dialog, it is best to use the GtkFontSelectionDialog box directly.

The simplest way to show how to use this is with an example that changes the font of a label in a window. Labels have many methods, but what they do not have is a method to change their font, unless you want to use Pango markup to do so. Instead of using the label's own methods, we can use the more general widget method to change the font of the label:

```
void           gtk_widget_modify_font           ( GtkWidget *widget,
                                                  PangoFontDescription *font_desc);
```

This function takes a pointer to the widget and a PangoFontDescription pointer. Pango is a library for the layout and rendering of internationalized text. A PangoFontDescription object is derived from a GBoxed object, which descends directly from GObject. It is opaque to the programmer, but it provides many methods to manipulate fonts. One of the most useful, and the simplest, is a function that returns a PangoFontDescription pointer when given a valid fontname that is available on the host system:

```
PangoFontDescription * pango_font_description_from_string ( const char *str);
```

This creates a new font description from a string representation in a specific form. The API documentation describes this form in detail, but for our purposes now, is is enough to know that the string returned by the gtk_font_button_get_font_name() function is in the correct form to supply to this function. Therefore, a signal handler to change the font of a label passed as user data to the handler could be:

```
void on_font_changed            ( GtkFontButton *button,
                                  GtkWidget     *label)
{
    const gchar *fontname;
    PangoFontDescription *font_desc;

    fontname  = gtk_font_button_get_font_name (button);
    font_desc = pango_font_description_from_string(fontname);
    gtk_widget_modify_font(label, font_desc);
}
```

This retrieves the fontname as a string, gets a Pango description for it, and modifies the label. When creating the widgets, the label would need an initial display font, which should match the font of the font button. If Pango cannot render the font for one reason or another, it will generate an error message on the standard error stream. The following listing shows the main program that does this.

```
#include <glib.h>
#include <gdk/gdk.h>
#include <gtk/gtk.h>
#include <stdlib.h>
#include <libgen.h>

#define WINWIDTH     400
#define WINHEIGHT    300
#define START_FONT   "Sans 12"
#define LABEL_TEXT   "Click the button to change the font."

int main (int argc,  char *argv[])
{
    GtkWidget     *window;
    GtkWidget     *fontbutton;
    GtkWidget     *label;
    GtkWidget     *vbox;
    GtkWidget     *hbox;
    GtkWidget     *hseparator;
    PangoFontDescription *initial_font;

    gtk_init (&argc, &argv);

    window = gtk_window_new          (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title             (GTK_WINDOW (window), basename(argv[0]));
    gtk_container_set_border_width (GTK_CONTAINER (window), 20);

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

    vbox = gtk_vbox_new(FALSE, 0);
    gtk_container_add(GTK_CONTAINER(window), vbox);

    hbox = gtk_hbox_new( FALSE, 0);
    gtk_box_pack_start( GTK_BOX (vbox), hbox, FALSE, FALSE, 10);

    hseparator = gtk_hseparator_new();
    gtk_box_pack_start( GTK_BOX (vbox), hseparator, FALSE, FALSE, 10);

    label = gtk_label_new (LABEL_TEXT );
    initial_font = pango_font_description_from_string (START_FONT );
    gtk_widget_modify_font (label, initial_font);
    gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);
    gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, FALSE, 5);

    fontbutton = gtk_font_button_new_with_font (START_FONT );
    gtk_font_button_set_title (GTK_FONT_BUTTON (fontbutton),
                               "Select a Font");
    gtk_font_button_set_use_font(GTK_FONT_BUTTON (fontbutton), TRUE);

    g_signal_connect (G_OBJECT (fontbutton), "font_set",
                      G_CALLBACK (on_font_changed),
                      (gpointer) label);

    gtk_box_pack_end( GTK_BOX (hbox), fontbutton,  FALSE, FALSE, 0);
    g_signal_connect(G_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);
    gtk_main();
    return 0;
}
```

7.2 The Color Button

The GtkColorButton lets the user choose a color by launching a GtkColorSelectionDialog. The button's label is a small rectangular swatch of the currently selected color; it has no text in its label. It has a method to set or get the title of the dialog box that is displayed when the button is clicked, and a few methods to set or get the current color and alpha channel value. It is created with either of

```
GtkWidget *    gtk_color_button_new              ( void);
GtkWidget *    gtk_color_button_new_with_color   ( const GdkColor *color);
```

The former creates a button with a black swatch, whereas the second creates a button with the current color determined by the GdkColor* argument. The function to retrieve the currently selected color from a color button is

```
void           gtk_color_button_get_color        ( GtkColorButton *color_button,
                                                    GdkColor *color);
```

7.2.1 About Color Representations

To use the GtkColorButton you need to know a little about the representation of colors in GDK, which has an extensive API for dealing with color. A color is stored in a GdkColor structure, which uses the *RGB model* to represent a color as independent components of red, green, and blue:

```
typedef struct {
  guint32 pixel;
  guint16 red;
  guint16 green;
  guint16 blue;
} GdkColor;
```

The RGB model is a standard method of representing color, and with 16 bits allocated for each of red, green, and blue, there are in theory 2^{48}or roughly 281 trillion possible colors. Programmatically, a color is defined by an assignment of values to each of the `red`, `green`, and `blue` members of the `GdkColor` structure. Larger values are closer to white. Black is the absence of color, and that is what you get when you zero the members of this struct. (That is why the swatch on a new button is black when you do not use the second of the two constructors above.)

As a convenience, GDK provides a method that lets you specify the color value as a string called the color's spec, using

```
gboolean       gdk_color_parse                    ( const gchar *spec,
                                                     GdkColor *color);
```

The string can either one of a large set of standard names from the X11 rgb.txt file, or it can be a hex value in the form '`#rgb`' '`#rrggbb`' '`#rrrgggbbb`' or '`#rrrrggggbbbb`' where '`r`', '`g`' and '`b`' are hex digits of the red, green, and blue components of the color, respectively. Standard color names are a very colorful collection in fact, with names such as "baker's chocolate" and "Free Speech Green".

Although you can visualize a color by its RGB values, that is not what GDK uses. Remember that GDK ultimately has to deliver a color to the underlying *X11* server, which has to make that color appear on the monitor. Because there are many different types of monitors, there are different hardware representations of color. That is where the `pixel` value in the `GdkColor` structure comes into play.

The X11 protocol defines a *visual* as the method of representing a color in hardware. A *true-color visual* represents colors by the actual RGB values, which are usually 24 bits or more. If 24 bits are used, 8-bits are used for each of R, G, and B in the pixel. The remaining 8 bits can be used for the *alpha channel*, which will be explained below.

Some systems use *colormaps* instead, and these are called *pseudo-color visuals*, because they do not necessarily contain the true colors. A colormap is an array whose entries consist of three values each – d bits for red, d bits for green, and d bits for blue. The number N of entries in the array determines how many different colors can be displayed on the screen at a single time. Colormaps are called *palettes* in some systems. If $d = 8$ then the palette contains 24-bit color values, which means that there are about 16 million (2^{24}) different possible colors in the palette, even though the palette can display only N of these at a time in a pseudo-color visual. When $N = 256$ for example, the palette can store 256 colors, each of which has 24-bit depth.

(need a picture here)

If a system uses a colormap, then the `pixel` member of the `GdkColor` structure contains the index in a colormap of the color defined by the `red`, `green`, and `blue` members of the structure[1]. There is usually a standard system colormap that is loaded by default. Your application can use the system colormap by calling `gdk_colormap_get_system()`.

A *true-color visual* also uses a colormap, but it is used differently. In a true-color visual, the three values in each entry of the array are accessed independently. To be precise, suppose that the colormap has 256 entries, each of which consists of 3 8-bit values, called red, green, and blue, respectively. Rather than thinking of this as a single array with 3-member structures as its entries, think of it instead as three parallel arrays named red, green, and blue, each of size 256. Since $256 = 2^8$, 8 bits can be used to index any row of these arrays. The pixel member of the `GdkColor` structure is decomposed into 8-bit fields. The high-order 8 bits are the index into the red array; the next 8 bits are the index into the green array, and the next 8 bits, the index into the blue array.

Since the entries in each array are 8 bits each, each of the pixel indices specifies an 8-bit color value. In other words, there are 256 different values for red, 256 for green, and 256 for blue. Since these are independent, there are $256 \times 256 \times 256 = 2^8 \cdot 2^8 \cdot 2^8 = 2^{24}$ possible colors. Thus, using a table of the same size as the pseudo-color visual, we can represent 16 million colors using true-color, versus the 256 colors that the table can represent using pseudo-color.

[1]The actual color in the colormap may not match those RGB values exactly.

(need a picture here)

When an application needs to render a color on the screen, it can store it in the colormap, so that it can use that same color repeatedly, perhaps at a later time. This is called *allocating* the color. Your application can fill in the `red`, `green`, and `blue` members of a `GdkColor` structure, and then ask for it to be allocated. After allocation, the `pixel` member contains a hardware-dependent value that is used to access the colormap.

Most systems allow colormaps to be shared among applications. If a color is shared, then other applications can use the colors that are stored in it. If the color could be changed by the application that first allocated it, this would corrupt the color in the other application. Therefore, GDK allows you to mark a color as writeable or read-only. If it is read-only, your application cannot modify it once it is allocated, and it can be shared. If it is writeable, your application can modify it, and so GDK will not let other applications share it. It is better, of course, to use read-only colors, since the colormap will take up less memory. And of course, when you finish with it, you can tell GDK to release that color's slot.

7.2.2 Using Color in GDK

We describe a very simple GTK+ application that uses a `GtkColorButton`. The main window contains a `GtkDrawingArea` widget, a label, and a color button. To organize things a bit within the window, we use a couple of boxes and an alignment.

A `GtkDrawingArea` is essentially a blank canvas on which your application can draw. It has its own GDK window, which is where your drawing is done. If you draw on this widget though, when the widget is covered by another window and then exposed again, the drawing will be lost, because the windowing system does not redraw this for you. How to use this widget for drawing is the subject of a different lesson. Here we avoid this problem because we are using this widget only to display the currently selected color as its background color. The background color of a widget is drawn by the windowing system.

We create the drawing area widget with

```
GtkWidget *    gtk_drawing_area_new              ( void);
```

which creates an empty canvas for us. We create an initial color using `gdk_color_parse()`, giving it an RGB text string as the choice. The three steps needed are

```
#define STARTCOLOR  "baker's chocolate"

// missing stuff here of course
// in main():

GdkColor  color;
gdk_color_parse (STARTCOLOR, &color);
```

Notice that `color` is on the program stack, not in dynamic memory. We can then use this `GdkColor` as the background of the drawing area and the starting color of the `GtkColorButton` that we create. To modify the background color of a widget, we need the method of the widget base class:

```
void           gtk_widget_modify_bg              ( GtkWidget *widget,
                                                   GtkStateType state,
                                                   const GdkColor *color);
```

All widgets that have windows can be assigned a background color, which is part of the widget's `GtkStyle` structure. Every widget can be in one of five different states, and the style associated to each state can be different. Therefore, one can assign a different background color to the widget for each independent state. The `GtkStateType` argument is an enumerated type that refers to one of the states in which a widget can be. The `gtk_widget_modify_bg()` method needs the `GtkStateType` argument so that it knows to which state's background to apply the color. The values of the enumeration are

GTK_STATE_NORMAL State during normal operation.

GTK_STATE_ACTIVE State of a currently active widget, such as a depressed button.

GTK_STATE_PRELIGHT State indicating that the mouse pointer is over the widget and the widget will respond to mouse clicks.

GTK_STATE_SELECTED State of a selected item, such the selected row in a list.

GTK_STATE_INSENSITIVE State indicating that the widget is unresponsive to user actions.

We will set the background color of the drawing area in its normal state with the call

```
gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &color);
```

and since `color` also contains our `STARTCOLOR` value, we can set the color of the button with the call

```
colorbutton = gtk_color_button_new_with_color (&color);
```

Then the only remaining important tasks are to define the callback function and to connect it to the "color-set" signal of the color button. The callback is:

```
void on_color_set                ( GtkColorButton *button,
                                   GtkWidget      *widget)
{
    GdkColor color;

    /* get the selected color from the dialog -
       be sure to pass by address! */
    gtk_color_button_get_color (button, &color);

    /* change the background color of the widget */
    gtk_widget_modify_bg (widget, GTK_STATE_NORMAL, &color);
}
```

Notice that the variable `color` is on the stack and that we have to pass it by address to the `gtk_color_button_get_color()` method. Notice also that we change the background color of the drawing area widget in two places: first in the main program before we start the `gtk_main()` loop, and second, in the callback.

Finally, we connect the callback to the "color-set" signal on the color button and pass a pointer to the drawing area widget as user data:

```
g_signal_connect (G_OBJECT (colorbutton), "color_set",
                  G_CALLBACK (on_color_set),
                  (gpointer) drawingarea);
```

The complete program is called `colorbutton_demo1.c` and can be found in the `buttons` demo directory on the server.

7.3 The File Chooser Button

The `GtkFileChooserButton` is a widget that lets the user select a file. It is a file name with a button to bring up a `GtkFileChooserDialog`. The user can then use that dialog to change the file associated with that button. When this button is used though, multiple files cannot be selected.

There are two methods to create a new

```
GtkWidget *    gtk_file_chooser_button_new        ( const gchar *title,
                                                    GtkFileChooserAction action);
GtkWidget *    gtk_file_chooser_button_new_with_dialog
                                                  ( GtkWidget *dialog);
```

The first function creates a file choosing button that will open a built-in dialog; the second requires that the programmer supply a dialog box. As this is our introduction to these buttons and we have not yet covered dialog boxes, we limit discussion to the first of the methods.

To create a new file chooser button, the programmer has to supply a string that will become the title of the dialog box that is opened when the user clicks on the button, and the action that the dialog must perform. The action is whether to open or save a file, or to open or create a directory. It is specified as a value of the `GtkFileChooserAction` enumerated type :

`GTK_FILE_CHOOSER_ACTION_OPEN` Indicates open mode. The file chooser will only let the user pick an existing file.

`GTK_FILE_CHOOSER_ACTION_SAVE` Indicates save mode. The file chooser will let the user pick an existing file, or type in a new filename.

`GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER` Indicates an Open mode for selecting folders. The file chooser will let the user pick an existing folder.

`GTK_FILE_CHOOSER_ACTION_CREATE_FOLDER` Indicates a mode for creating a new folder. The file chooser will let the user name an existing or new folder.

When the `GTK_FILE_CHOOSER_ACTION_OPEN` mode is specified, the `GtkFileChooserDialog` that is opened displays the files in a specific directory. The user can control which directory it starts in with

```
void      gtk_file_chooser_set_current_folder  ( GtkFileChooser   *chooser,
                                                 gchar            *directory );
```

In general this sets the current directory of the dialog box. If we want it to start in the user's home directory, a logical choice, we would use the convenient GLib function

```
const gchar * g_get_home_dir                   (void);
```

which returns a string that can be passed to the former method. We could also use the POSIX `getenv()` function, as in `getenv(“HOME”)`, which would also give as the path to the home directory.

The button itself has very few methods of its own; its utility is primarily from its being an implementation of the `GtkFileChooser` interface. Basically, besides the dialog box that it owns, it has three properties:

`"focus-on-click"`	whether or not a click allows it to grab focus.
`"title"`	the title of the dialog box, as a string.
`"width-chars"`	the width of the label inside the button, measured in characters.

Each of these can be set or retrieved by a method of the form

```
void      gtk_file_chooser_button_set_xxx   (GtkFileChooserButton *button,
                                             gint n_chars);
gint      gtk_file_chooser_button_get_xxx   (GtkFileChooserButton *button);
```

where the "xxx" is replaced by the property name, replacing "-" by "_" and where the return type of the get method and the type of the second parameter of the set method are matching and one of gboolean, const gchar*, and gint respectively.

Like the font button and the color button, the file chooser button emits just a single signal, "file-set", which is emitted when the user clicks the OK button in the file selection dialog box. Because the GtkFileChooserButton implements the GtkFileChooser interface, it also emits the signals of that class. One can therefore monitor the "selection-changed" signal instead; the result will be the same, but only if the button is set to select files. If the button is created with the GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER action, it will not emit the "file-set" signal; instead your program must connect to the "selection-changed" signal.

We will demonstrate how to use this button with a simple application that displays two buttons, one to select a folder, and the other to select a file. We will call these the folder chooser and the file chooser respectively. When a folder is selected, the folder chooser button will change the current directory of the file chooser button. When a file is selected, the file chooser button will change a label widget in the main program that displays the pathname of the currently selected file. To make this all the more interesting, we will add file filters to the file chooser button.

7.3.1 The GtkFileFilter

A GtkFileFilter can be used to restrict the files being shown in a GtkFileChooser. Files can be filtered based on their

- *name* using gtk_file_filter_add_pattern(),
- *mime type* using gtk_file_filter_add_mime_type(), or
- with a *custom filter function* using gtk_file_filter_add_custom().

When filtering by name using gtk_file_filter_add_pattern(), shell file globs can be used. For example, the pattern "*.c" will filter out all files that do not end in a ".c" extension, and "*.[ch]" will match header files and C source files.

Filtering with mime types handles aliasing and sub-classing of mime types. For example, a filter for text/plain also matches a file with mime type application/rtf, since application/rtf is a subclass of text/plain. Also, GtkFileFilter allows wildcards for the subtype of a mime type, so you can filter for all text files with "text/*" for example.

The functions to add by name (pattern) or by mime type are:

```
void        gtk_file_filter_add_pattern       ( GtkFileFilter *filter,
                                                const gchar *pattern);
void        gtk_file_filter_add_mime_type     ( GtkFileFilter *filter,
                                                const gchar *mime_type);
```

The procedure for adding a filter to a GtkFileChooser has four steps:

1. Create the filter using

   ```
   GtkFileFilter * gtk_file_filter_new          ( void);
   ```

2. Give the filter a name, using

```
void            gtk_file_filter_set_name      ( GtkFileFilter *filter,
                                                const gchar *name);
```

3. Add one or more patterns or mime/types, one at a time, using one of the above functions, and
4. Add the filters to the `GtkFileChooser` with

```
void            gtk_file_chooser_add_filter    ( GtkFileChooser *chooser,
                                                 GtkFileFilter *filter);
```

The order in which the filters are added to the file chooser will be the order in which they appear in the combo box in the `GtkFileChooserDialog` box. The following listing shows a function that adds a number of filters to a `GtkFileChooserButton`'s dialog box.

```
void set_up_filters ( GtkFileChooser  *chooser)
{
    GtkFileFilter *filter;

    filter = gtk_file_filter_new ();
    gtk_file_filter_set_name    (filter, "All Files");
    gtk_file_filter_add_pattern (filter, "*");
    gtk_file_chooser_add_filter (chooser, filter);

    filter = gtk_file_filter_new ();
    gtk_file_filter_set_name    (filter, "Text Files");
    gtk_file_filter_add_mime_type (filter, "text/*");
    gtk_file_chooser_add_filter (chooser, filter);

    filter = gtk_file_filter_new ();
    gtk_file_filter_set_name (filter, "Image Files");
    gtk_file_filter_add_pixbuf_formats(filter);
    gtk_file_chooser_add_filter (chooser, filter);

    filter = gtk_file_filter_new ();
    gtk_file_filter_set_name (filter, "C/C++ Files");
    gtk_file_filter_add_pattern (filter, "*.[chC]");
    gtk_file_filter_add_pattern (filter, "*.cc");
    gtk_file_filter_add_pattern (filter, "*.cpp");
    gtk_file_filter_add_pattern (filter, "*.CC");
    gtk_file_chooser_add_filter (chooser, filter);
}
```

7.3.2 Supported Image File Formats

Notice that `GtkFileFilter` also has a method to add an image file filter:

```
void            gtk_file_filter_add_pixbuf_formats  (GtkFileFilter *filter);
```

This function will add to the given filter all image formats supported by `GdkPixbuf` on the given host. If you are curious as to which image file types are actually supported, you can use the following function, defined in the Gdk-PixBuf Reference Manual:

```
GSList *        gdk_pixbuf_get_formats               (void);
```

The following listing is of a command-line program that will print the list of image formats supported by GdkPixbuf on your host machine.

```
Listing .
#include <gtk/gtk.h>
// Applies gdk_pixbuf_format_is_writable(data) to the GdkPixBufFormat
// struct pointed to by data, and if it returns true, it prepends that
// pointer to the list pointed to by *list. Because the list itself is
// modified in the function, the pointer to the front of the list changes.
// Therefore, it is passed by reference as a pointer, i.e., as **list, and
// so in this function, *list is the pointer itself. When it is finished
// *list still points to the front of the list.
void add_if_writable (GdkPixbufFormat *data, GSList **list)
{
  if (gdk_pixbuf_format_is_writable (data))
    *list = g_slist_prepend (*list, data);
}

// This is also a GFunc function, which is why it is given the second
// parameter, list, even though it does not use it.
// It retrieves the name of the image file format stored in the current
// GdkPixbufFormat struct and prints it on standard output.
void format_print (GdkPixbufFormat *data, GSList **list)
{
    g_print ( "%s\n", gdk_pixbuf_format_get_name(data) );
}

int  main (int argc, char *argv[])
{
    // Get the list of all formats
    // This function returns a Glib singly-linked list of pointers
    // to GdkPixbufFormat structs. The structs are owned by Gdk but the
    // list of pointers must be freed when you are finished with the
    // list, with g_slist_free().
    GSList *formats          = gdk_pixbuf_get_formats ();

    // Initialize an empty GSList
    GSList *writable_formats = NULL;
    /*
       Produce the list of writable formats using the foreach algorithm
       applying a function of the form

             void gfunc (gpointer data, gpointer user_data)

       to each element of the list, passing in user_data as the second
       argument to it, and the current list element as the first argument
       to it. For example, in the code below, the GFunc is add_if_writable()
       and the list is formats. The data passed as the second argument is
       a pointer to the list writable formats. add_if_writable() will be
       applied to the current list item in format for each item in format,
       with &writable_formats as its second argument.
    */
    g_slist_foreach (formats, ( GFunc) add_if_writable, &writable_formats);

    // Print the list of writable formats using the foreach algorithm
    g_slist_foreach (formats, ( GFunc) format_print,    &writable_formats);
    g_slist_free (formats);
    return 0;
}
```

The main program and callbacks of our demonstration program follow.

```
void on_folder_changed (GtkFileChooser *folder_chooser,
                        GtkFileChooser *file_chooser)
{
    gchar *folder = gtk_file_chooser_get_filename (folder_chooser);
    gtk_file_chooser_set_current_folder (file_chooser, folder);
}

void on_file_changed   (GtkFileChooser *file_chooser,
                        GtkLabel *label)
{
    gchar *file = gtk_file_chooser_get_filename (file_chooser);
    if ( NULL == file )
        gtk_label_set_text (label, "no file selected");
    else
        gtk_label_set_text (label, file);
}

int main (int argc,  char *argv[])
{
    GtkWidget      *window;
    GtkWidget      *folder_chooser;
    GtkWidget      *file_chooser;
    GtkWidget      *filename_label;
    GtkWidget      *frame;
    GtkWidget      *vbox;
    GtkWidget      *alignment;

    gtk_init (&argc, &argv);

    window = gtk_window_new          (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title             (GTK_WINDOW (window), basename(argv[0]));
    gtk_widget_set_size_request      (window, WINWIDTH, WINHEIGHT);
    gtk_container_set_border_width   (GTK_CONTAINER (window), 20);

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

    // Put  a vbox in the main window and pack everything into it
    vbox = gtk_vbox_new (FALSE, 5);
    gtk_container_add (GTK_CONTAINER (window), vbox);

    // First in is a frame that will contain  a label. The label will be
    // changed by the on_file_changed callback to contain the absolute
    // pathname of the selected file.
    frame = gtk_frame_new ("Currently selected file: ");
    gtk_box_pack_start (GTK_BOX (vbox), frame, TRUE, TRUE, 10);

    // The filename_label will contain the path. Turn on line wrap and
    // wrap on characters since there are no spaces in the path and
    // it might get pretty long. Add soem padding too.
    filename_label  = gtk_label_new ("");
    gtk_label_set_justify( GTK_LABEL(filename_label), GTK_JUSTIFY_LEFT);
    gtk_label_set_line_wrap( GTK_LABEL(filename_label), TRUE);
    gtk_label_set_line_wrap_mode( GTK_LABEL(filename_label),
                                  PANGO_WRAP_CHAR);
    gtk_misc_set_alignment( GTK_MISC(filename_label), 0, 0);
    gtk_container_add (GTK_CONTAINER (frame), filename_label);


    // Create the two filechooser buttons, one to select a folder,
    // the other to select a file. Make them a fixed minimum size so
    // they don't look strange.
    folder_chooser = gtk_file_chooser_button_new ("Choose a Folder",
                                  GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER);
    gtk_widget_set_size_request    (folder_chooser, BUTTONWIDTH, -1);

    file_chooser   = gtk_file_chooser_button_new ("Choose a File",
                                  GTK_FILE_CHOOSER_ACTION_OPEN);
    gtk_widget_set_size_request    (file_chooser, BUTTONWIDTH, -1);

    // Let the starting directory be the home directory
    gtk_file_chooser_set_current_folder (GTK_FILE_CHOOSER (folder_chooser),
                                         g_get_home_dir());
    gtk_file_chooser_set_current_folder (GTK_FILE_CHOOSER (file_chooser),
                                         g_get_home_dir());

    // Connect the selection_changed signal to two different callbacks
    g_signal_connect (G_OBJECT (folder_chooser), "selection_changed",
                      G_CALLBACK (on_folder_changed),
                      (gpointer) file_chooser);

    g_signal_connect (G_OBJECT (file_chooser), "file_set",
                      G_CALLBACK (on_file_changed),
                      (gpointer) filename_label);

    set_up_filters(GTK_FILE_CHOOSER(file_chooser));

    alignment = gtk_alignment_new ( 1, 0, 0, 0 );
    gtk_container_add (GTK_CONTAINER (alignment), file_chooser);
    gtk_box_pack_end (GTK_BOX (vbox), alignment, FALSE, FALSE, 10);

    alignment = gtk_alignment_new ( 1, 0, 0, 0 );
    gtk_container_add (GTK_CONTAINER (alignment), folder_chooser);
    gtk_box_pack_end (GTK_BOX (vbox), alignment, FALSE, FALSE, 10);

    gtk_widget_show_all (window);

    gtk_main ();
    return 0;
```