GTK+ 3 Tutorial for Lisp

Version 0.0

Short Contents

Table of Contents

1 Introduction

The `cl-cffi-gtk` library is a Lisp binding to GTK+ (GIMP Toolkit) which is a library for creating graphical user interfaces. Gtk+ is licensed using the LGPL which has been adopted for the `cl-cffi-gtk` library with a preamble that clarifies the terms for use with Lisp programs and is referred as the LLGPL.

This work is based on the `cl-gtk2` library which has been developed by Kalyanov Dmitry and already is a fairly complete Lisp binding to GTK+. The focus of the `cl-cffi-gtk` library is to document the Lisp library much more complete and to do the implementation as consistent as possible. Most informations about GTK+ can be gained by reading the C documentation. Therefore, the C documentation from `www.gtk.org` is included into the Lisp files to document the Lisp binding to the GTK+ library. This way the calling conventions are easier to determine and missing functionality is easier to detect.

The GTK+ library is called the GIMP toolkit because GTK+ was originally written for developing the GNU Image Manipulation Program (GIMP), but GTK+ has now been used in a large number of software projects, including the GNU Network Object Model Environment (GNOME) project. GTK+ is built on top of GDK (GIMP Drawing Kit) which is basically a wrapper around the low-level functions for accessing the underlying windowing functions (Xlib in the case of the X windows system), and gdk-pixbuf, a library for client-side image manipulation.

GTK+ is essentially an object oriented application programmers interface (API). Although written completely in C, GTK+ is implemented using the idea of classes and callback functions (pointers to functions).

A third component is called GLib which contains replacements for standard calls, as well as additional functions for handling linked lists, etc. The replacement functions are used to increase the portability of GTK+, as some of the functions implemented here are not available or are non standard on other Unixes such as g_strerror(). Some also contain enhancements to the libc versions, such as g_malloc() that has enhanced debugging utilities.

In version 2.0, GLib has picked up the type system which forms the foundation for the class hierarchy of GTK+, the signal system which is used throughout GTK+, a thread API which abstracts the different native thread APIs of the various platforms and a facility for loading modules.

As the last component, GTK+ uses the Pango library for internationalized text output.

This tutorial describes the Lisp interface to GTK+. It is based on the official GTK+ 2.0 Tutorial of the C implementation.

2 Getting Started

2.1 Installation

The first thing to do is to download the `cl-cffi-gtk` source and to install it. The latest version is available from the repository at `github.com/crategus/cl-cffi-gtk`. The `cl-cffi-gtk` library can be loaded with the command `(asdf:operate 'asdf:load-op :cl-cffi-gtk)` from the Lisp prompt. The library is developed with the Lisp SBCL 1.0.56 on a Linux system and GTK+ 3.4.

At this time (April 2012) there is no GTK+ 3 library for Windows available at `www.gtk.org`. The current maintained version is GTK+ 2.24. The repository at `github.com/crategus/cl-cffi-gtk` has a branch `gtk-2-24` which is tested with SBCL 1.0.54 and GTK+ 2.24 on Windows.

With SBCL an alternative command to load the `cl-cffi-gtk` library is `(asdf:load-system 'cl-cffi-gtk)`. The libraries GLib, GObject, GDK, GIO, Pango, and Cairo are part of GTK+ and loaded in addition with the above command. These libraries can be loaded individually with the commands:

```
(asdf:load-system 'cl-cffi-gtk-glib)
(asdf:load-system 'cl-cffi-gtk-gobject)
(asdf:load-system 'cl-cffi-gtk-gdk)
(asdf:load-system 'cl-cffi-gtk-gio)
(asdf:load-system 'cl-cffi-gtk-pango)
(asdf:load-system 'cl-cffi-gtk-cairo)
```

It is assumed, that the corresponding system definition files `cl-cffi-gtk.asd` for GTK+, `cl-cffi-gtk-glib.asd` for GLib, `cl-cffi-gtk-gobject.asd` for GObject, `cl-cffi-gtk-gdk.asd` for GDK, `cl-cffi-gtk-gio.asd` for GIO, `cl-cffi-gtk-pango.asd`, for Pango and `cl-cffi-gtk-cairo.asd` for Cairo are correctly registered to `asdf:*central-registry*` or the corresponding symlinks are created.

For example for SBCL on a Linux system the symlinks are added to the directory `~/.sbcl/systems` with the commands:

```
cd ~/.sbcl/systems
ln -s /path-to-cl-cffi-gtk/gtk/cl-cffi-gtk.asd
ln -s /path-to-cl-cffi-gtk/gdk/cl-cffi-gtk-gdk.asd
ln -s /path-to-cl-cffi-gtk/gobject/cl-cffi-gtk-gobject.asd
ln -s /path-to-cl-cffi-gtk/glib/cl-cffi-gtk-glib.asd
ln -s /path-to-cl-cffi-gtk/gio/cl-cffi-gtk-gio.asd
ln -s /path-to-cl-cffi-gtk/pango/cl-cffi-gtk-pango.asd
ln -s /path-to-cl-cffi-gtk/cairo/cl-cffi-gtk-cairo.asd
```

The `cl-cffi-gtk` library depends further on the following libraries, which must be installed:

CFFI the Common Foreign Function Interface, purports to be a portable foreign function interface for Common Lisp. See `http://common-lisp.net/project/cffi/`.

Trivial-Garbage
: provides a portable API to finalizers, weak hash-tables and weak pointers on all major CL implementations. See http://www.cliki.net/trivial-garbage.

Iterate
: is a lispy and extensible replacement for the LOOP macro. See http://common-lisp.net/project/iterate/.

Bordeaux-Threads
: lets you write multi-threaded applications in a portable way. See http://common-lisp.net/project/bordeaux-threads/.

Closer-MOP
: Closer to MOP is a compatibility layer that rectifies many of the absent or incorrect MOP features as detected by MOP Feature Tests. See http://common-lisp.net/project/closer/closer-mop.html.

Information about the installation can be obtained with the function `cl-cffi-gtk-build-info`. This is an example for the output, when calling the function from the Lisp prompt after loading the library:

```
* (cl-cffi-gtk-build-info)

cl-cffi-gtk version: 0.0.0
cl-cffi-gtk build date: 22:35 5/2/2012
GTK+ version: 3.4.1
GLIB version: 2.32.1
Pango version: 1.30.0
Cairo version: 1.10.2
Machine type: X86
Machine version: Intel(R) Pentium(R) M processor 1.73GHz
Software type: Linux
Software version: 3.2.0-24-generic
Lisp implementation type: SBCL
Lisp implementation version: 1.0.56
```

2.2 A Simple Window

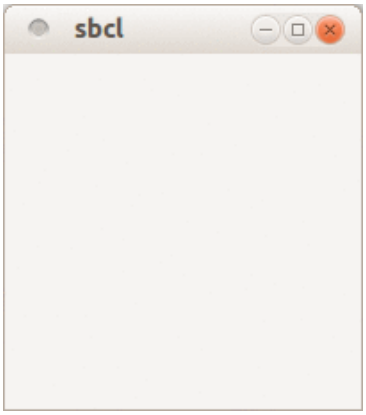

Figure 2.1: A Simple Window

The `cl-cffi-gtk` source distribution contains the complete source to all of the examples used in this tutorial. To begin the introduction to GTK+, the output of the simplest program possible is shown in Figure 2.1 and the Lisp code in Example 2.2.

The program creates a 200 x 200 pixel window. In this case the window has the default title "sbcl". The window can be sized and moved. Because no special action is implemented to close the window, depending on the operating system the program might hang. First in Example 2.1 the C program of the GTK+ 2.0 Tutorial is presented to show the close connection between the C library and the implementation of the Lisp binding. The code of the Lisp program is shown in Example 2.2.

Example 2.1: A simple window in the programing language C

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

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

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_show  (window);

    gtk_main ();

    return 0;
}
```

Example 2.2: A Simple Window in the programing language Lisp

```
(asdf:operate 'asdf:load-op :cl-cffi-gtk)

(defpackage :gtk-tutorial
  (:use :gtk :gdk :gobject :glib :pango :cairo :common-lisp))

(in-package :gtk-tutorial)

(defun example-simple-window ()
  (within-main-loop
    (let (;; Create a toplevel window.
          (window (gtk-window-new :toplevel)))
      ;; Show the window.
      (gtk-widget-show-all window))))
```

The first four lines of code load the `cl-cffi-gtk` library and define an own package `:gtk-tutorial` for the example program. The package `:gtk-tutorial` includes the symbols from the packages `:gtk` for GTK, `:gdk` for GDK, `:glib` for GLib, `:pango` for Pango, and `:cairo` for Cairo. In further examples of this tutorial these first lines of code are omitted.

The macro `within-main-loop` is a wrapper about a GTK+ program. The functionality of the macro corresponds to the C functions `gtk_init()` and `gtk_main()` which initialize and start a GTK+ program. Both functions have corresponding Lisp functions with the names `gtk-init` and `gtk-main`, but these functions are not used in this tutorial. `gtk-init` is automatically called when loading the Lisp library `cl-cffi-gtk` and the function `gtk-main` is called from the macro `within-main-loop`.

Only two further functions are needed in this simple example. The window is created with the function `gtk-window-new`. The keyword `:toplevel` tells GTK+ to create a toplevel window. The second call `gtk-widget-show-all` displays the new window.

In addition to the function `gtk-widget-show-all` the function `gtk-widget-show` is available. The function `gtk-widget-show` only displays the widget, which is the argument to the function. The function `gtk-widget-show-all` displays the window and all including child widgets. For the first simple window this makes no difference, because the window has not child widgets.

2.3 More about the Lisp binding to GTK+

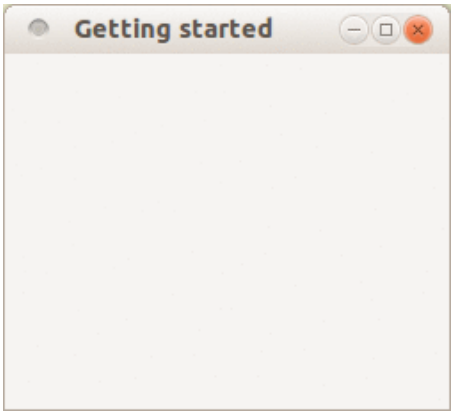

Figure 2.2: Getting started

Figure 2.2 and Example 2.3 show a second implementation of the simple program discussed in Section 2.2 [A Simple Window], page 4. The second implementation uses the fact, that all GTK+ widgets are internally represented in the Lisp binding through a Lisp class. The Lisp class `gtk-window` represents the required window, which corresponds to the C class `GtkWindow`. An instance of the Lisp class `gtk-window` can be created with the function `make-instance`. Furthermore, the slots of the window class can be given new values to overwrite the default values. These slots represent the properties of the C classes. [GtkWindow], page 153 shows a list of properties, which are available for an instance of the class `GtkWindow`. In addition an instance has all properties of the inherited classes. The object hierarchy shows, that the class `GtkWindow` inherits all properties of the classes `GtkWidget`, `GtkContainer`, and `GtkBin`.

In Example 2.3 the property `type` with the keyword `:toplevel` creates again a toplevel window. In addition a title is set assigning the string `"Getting started"` to the property `title` and the width of the window is a little enlarged assigning the value `250` to the property `default-width`. `title`, `type`, and `default-width` are properties of the class

GtkWindow as shown in [GtkWindow], page 153. The result of the example program is shown in Figure 2.2.

The keyword :toplevel is one of the values of the enumeration type GtkWindowType in C. In the Lisp binding this enumeration is implemented as gtk-window-type with the two possible keywords :toplevel for GTK_WINDOW_TOPLEVEL and :popup for GTK_WINDOW_POPUP. Most windows are of the type :toplevel. Windows with this type are managed by the window manager and have a frame by default. Windows with type :popup are ignored by the window manager and are used to implement widgets such as menus or tooltips.

Example 2.3: Getting Started

```
(defun example-getting-started ()
  (within-main-loop
    (let (;; Create a toplevel window with a title and a default width.
          (window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Getting started"
                                 :default-width 250)))
      ;; Show the window.
      (gtk-widget-show-all window))))
```

The Example 2.3 shows, that the Lisp function gtk-window-new is not really needed. The function gtk-window-new is internally implemented in the Lisp binding simply as:

```
(defun gtk-window-new (type)
  (make-instance 'gtk-window :type type))
```

To set the title of the window or to change the default width of a window the C library knows accessor functions to set the corresponding values. In C the title of the window is set with the function gtk_window_set_title(). The corresponding Lisp function is gtk-window-set-title. Accordingly, the default width of the window can be set in C with the function gtk_window_set_default_size(), which sets both the default width and the default height. In Lisp this function is named gtk-window-set-default-size. As we have seen, these Lisp accessor functions are not really needed when creating a window, but the functions are provided to allow the user to translate a C program more easy to Lisp.

At last, in Lisp it is possible to use the accessors of the slots to get or set the value of a widget property. The properties default-width and default-height of the Lisp class gtk-window have the Lisp accessor functions gtk-window-default-width and gtk-window-default-height. With these accessor functions the C function gtk_window_set_default_size() is implemented the following way in the Lisp library:

```
(defun gtk-window-set-default-size (window width height)
  (setf (gtk-window-default-width window) width
        (gtk-window-default-height window) height))
```

As a second example the Lisp implementation of the C function gtk_window_get_default_size() is shown:

```
(defun gtk-window-get-default-size (window)
  (values (gtk-window-default-width window)
          (gtk-window-default-height window)))
```

In distinction to the C function `gtk_window_get_default_size()`, which is implemented as

```
void gtk_window_get_default_size (GtkWindow *window,
                                  gint      *width,
                                  gint      *height)
```

the Lisp implementation does not modify the arguments `width` and `height`, but returns the values.

2.4 Hello World in GTK+

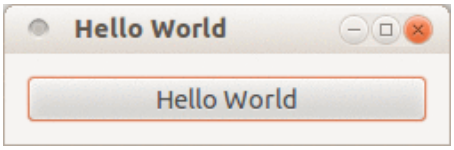

Figure 2.3: Hello World

Now a program with a button is presented. The output is shown in Figure 2.3. Again the C program from the GTK+ 2.0 Tutorial is shown first in Example 2.4 to learn more about the differences between a C and a Lisp implementation.

Example 2.4: Hello World in the programing language C

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

/* This is a callback function. The data arguments are ignored
 * in this example. More on callbacks below. */
static void hello( GtkWidget *widget, gpointer data )
{
    g_print ("Hello World\n");
}

static gboolean delete_event( GtkWidget *widget,
                              GdkEvent  *event,
                              gpointer   data )
{
    /* If you return FALSE in the "delete-event" signal handler,
     * GTK will emit the "destroy" signal. Returning TRUE means
     * you don't want the window to be destroyed.
     * This is useful for popping up 'are you sure you want to quit?'
     * type dialogs. */

    g_print ("delete event occurred\n");

    /* Change TRUE to FALSE and the main window will be destroyed with
     * a "delete-event". */

    return TRUE;
```

```
}

/* Another callback */
static void destroy( GtkWidget *widget, gpointer data )
{
    gtk_main_quit ();
}

int main( int argc, char *argv[] )
{
    /* GtkWidget is the storage type for widgets */
    GtkWidget *window;
    GtkWidget *button;

    /* This is called in all GTK applications. Arguments are parsed
     * from the command line and are returned to the application. */
    gtk_init (&argc, &argv);

    /* create a new window */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* When the window is given the "delete-event" signal (this is given
     * by the window manager, usually by the "close" option, or on the
     * titlebar), we ask it to call the delete_event () function
     * as defined above. The data passed to the callback
     * function is NULL and is ignored in the callback function. */
    g_signal_connect (window, "delete-event",
                      G_CALLBACK (delete_event), NULL);

    /* Here we connect the "destroy" event to a signal handler.
     * This event occurs when we call gtk_widget_destroy() on the window,
     * or if we return FALSE in the "delete-event" callback. */
    g_signal_connect (window, "destroy",
                      G_CALLBACK (destroy), NULL);

    /* Sets the border width of the window. */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* Creates a new button with the label "Hello World". */
    button = gtk_button_new_with_label ("Hello World");

    /* When the button receives the "clicked" signal, it will call the
     * function hello() passing it NULL as its argument.  The hello()
     * function is defined above. */
    g_signal_connect (button, "clicked",
                      G_CALLBACK (hello), NULL);
```

```
    /* This will cause the window to be destroyed by calling
     * gtk_widget_destroy(window) when "clicked".  Again, the destroy
     * signal could come from here, or the window manager. */
    g_signal_connect_swapped (button, "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              window);

    /* This packs the button into the window (a gtk container). */
    gtk_container_add (GTK_CONTAINER (window), button);

    /* The final step is to display this newly created widget. */
    gtk_widget_show (button);

    /* and the window */
    gtk_widget_show (window);

    /* All GTK applications must have a gtk_main(). Control ends here
     * and waits for an event to occur (like a key press or
     * mouse event). */
    gtk_main ();

    return 0;
}
```

Now, the Lisp implementation is presented in Example 2.5. One difference is, that the function `make-instance` is used to create the window and the button. Another point is, that the definition of separate callback functions is avoided. The callback functions are short, implemented through Lisp `lambda` functions and are passed as the third argument to the function `g-signal-connect`. More about signals and callback functions follows in Section 2.5 [Introduction to Signals and Callbacks], page 12.

In Example 2.5 a border with a width of `12` is added to the window setting the property `border-width` when creating the window with the function `make-instance`. The C implementation uses the function `gtk_container_set_border_width()` which is available in Lisp as `gtk-container-set-border-width`. The property `border-width` is inherited from the C class `GtkContainer`, which in the Lisp library is represented through the Lisp class `gtk-container`. Therefore, the accessor function has the prefix `gtk_container` in C and `gtk-container` in Lisp. In addition Lisp knows the accessor function `gtk-container-border-width` to set or to get the property `border-width`. A full list of properties of `GtkContainer` is available in [GtkContainer], page 143.

Example 2.5: Hello World in the programing language Lisp

```
(defun example-hello-world ()
  (within-main-loop
    (let (;; Create a toplevel window, set a border width.
          (window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Hello World"
```

```
                              :default-width 250
                              :border-width 12))
      ;; Create a button with a label.
      (button (make-instance 'gtk-button :label "Hello World")))
  ;; Signal handler for the button to handle the signal "clicked".
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (format t "Hello world.~%")
                      (gtk-widget-destroy window)))
  ;; Signal handler for the window to handle the signal "destroy".
  (g-signal-connect window "destroy"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-main-quit)))
  ;; Signal handler for the window to handle the signal "delete-event".
  (g-signal-connect window "delete-event"
                    (lambda (widget event)
                      (declare (ignore widget event))
                      (format t "Delete Event Occured.~%")
                      t))
  ;; Put the button into the window.
  (gtk-container-add window button)
  ;; Show the window and the button.
  (gtk-widget-show-all window))))
```

An attentive reader notes that in distinction to the C implementation the function `gtk-widget-show` is not called for every single widget, which are in Example 2.5 the window and the button. Instead the function `gtk-widget-show-all` is used to display the window with all including widgets.

Three more functions are introduced in Example 2.5. The function `gtk-widget-destroy` takes as an argument any widget and destroys it. In the above example this function is called by the signal handler of the button. When the button is clicked by the user, the signal `"clicked"` is catched by the signal handler, which causes a call of the function `gtk-widget-destroy` for the toplevel window. Now the toplevel window receives the signal `"destroy"`, which is handled by a signal handler of the toplevel window. This signal handler calls the function `gtk-main-quit`, which stops the event loop and finishes the application.

A second signal handler is connected to the toplevel window to catch the signal `"delete-event"`. The signal `"delete-event"` occurs, when the user or the window manager tries to close the window. In this case, the signal handler prints a message on the console. Because the value `T` is returned from the signal handler the handling of the signal is stopped and the window is not closed, but the execution of the application is continued. To close the window, the user has to press the button in this example.

At last, the function `gtk-container-add` is used to put the button into the toplevel window. Chapter 3 [Packing Widgets], page 21 shows how it is possible to put more than one widget into a window.

2.5 Introduction to Signals and Callbacks

GTK+ is an event driven toolkit, which means GTK+ will sleep until an event occurs and control is passed to the appropriate function. This passing of control is done using the idea of "signals". (Note that these signals are not the same as the Unix system signals, and are not implemented using them, although the terminology is almost identical.) When an event occurs, such as the press of a mouse button, the appropriate signal will be "emitted" by the widget that was pressed. This is how GTK+ does most of its useful work. There are signals that all widgets inherit, such as "destroy", and there are signals that are widget specific, such as "toggled" on a toggle button.

To make a button perform an action, a signal handler is set up to catch these signals and to call the appropriate function. This is done in the C GTK+ library by using a function such as

```
gulong g_signal_connect( gpointer      *object,
                         const gchar   *name,
                         GCallback     func,
                         gpointer      func_data );
```

where the first argument is the widget which will be emitting the signal, and the second the name of the signal to catch. The third is the function to be called when the signal is caught, and the fourth, the data to have passed to this function.

The function specified in the third argument is called a "callback function", and is for a C program of the form

```
void callback_func( GtkWidget *widget,
                    ... /* other signal arguments */
                    gpointer   callback_data );
```

where the first argument will be a pointer to the widget that emitted the signal, and the last a pointer to the data given as the last argument to the C function `g_signal_connect()` as shown above. Note that the above form for a signal callback function declaration is only a general guide, as some widget specific signals generate different calling parameters.

This mechanism is realized in Lisp with a similar function `g-signal-connect` which has the arguments `widget`, `name`, and `func`. In distinction from C the Lisp function `g-signal-connect` has not the argument `func_data`. The functionality of passing data to a callback function can be realized with the help of a `lambda` function in Lisp.

As an example the following code shows a typical C implementation which is used in the Hello World program.

```
g_signal_connect (window, "destroy", G_CALLBACK (destroy), NULL);
```

This is the corresponding callback function which is called when the event "destroy" occurs.

```
static void destroy (GtkWidget *widget, gpointer data)
{
    gtk_main_quit ();
}
```

In the corresponding Lisp implementation we simply declare a `lambda` function as a callback function which is passed as the third argument.

```
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
```

If it is necessary to have a separate function which needs user data, the following implementation is possible

```
(defun separate-event-handler (widget arg1 arg2 arg3)
  [ here is the code of the event handler ] )
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (separate-event-handler widget arg1 arg2 arg3)))
```

If no extra data is needed, but the callback function should be separated out than it is also possible to implement something like

```
(g-signal-connect window "destroy" #'separate-event-handler)
```

Furthermore, the C function

```
gulong g_signal_connect_swapped (gpointer    *object,
                                 const gchar *name,
                                 GCallback   func,
                                 gpointer    *callback_data);
```

is not implemented in Lisp. Again this functionality is already present with the help of `lambda` functions in Lisp.

2.6 An Upgraded Hello World

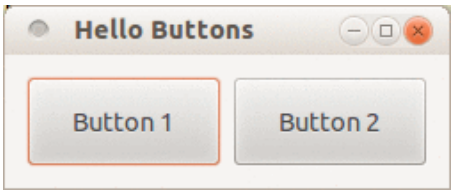

Figure 2.4: Upgraded Hello World

Figure 2.4 and Example 2.7 show a slightly improved Hello World with better examples of callbacks. This will also introduce the next topic, packing widgets. First, the C program is shown in Example 2.6.

Example 2.6: An upgraded Hello World in the programing language C

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

/* Our new improved callback.  The data passed to this function
 * is printed to stdout. */
static void callback( GtkWidget *widget, gpointer data )
{
    g_print ("Hello again - %s was pressed\n", (gchar *) data);
}
```

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

int main( int argc, char *argv[] )
{
    /* GtkWidget is the storage type for widgets */
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;

    /* This is called in all GTK applications. Arguments are parsed
     * from the command line and are returned to the application. */
    gtk_init (&argc, &argv);

    /* Create a new window */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* This is a new call, which just sets the title of our
     * new window to "Hello Buttons!" */
    gtk_window_set_title (GTK_WINDOW (window), "Hello Buttons!");

    /* Here we just set a handler for delete_event that immediately
     * exits GTK. */
    g_signal_connect (window, "delete-event",
                      G_CALLBACK (delete_event), NULL);

    /* Sets the border width of the window. */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* We create a box to pack widgets into.  This is described in detail
     * in the "packing" section. The box is not really visible, it
     * is just used as a tool to arrange widgets. */
    box1 = gtk_hbox_new (FALSE, 0);

    /* Put the box into the main window. */
    gtk_container_add (GTK_CONTAINER (window), box1);

    /* Creates a new button with the label "Button 1". */
    button = gtk_button_new_with_label ("Button 1");
```

```
    /* Now when the button is clicked, we call the "callback" function
     * with a pointer to "button 1" as its argument */
    g_signal_connect (button, "clicked",
                      G_CALLBACK (callback), (gpointer) "button 1");

    /* Instead of gtk_container_add, we pack this button into the invisible
     * box, which has been packed into the window. */
    gtk_box_pack_start (GTK_BOX(box1), button, TRUE, TRUE, 0);

    /* Always remember this step, this tells GTK that our preparation for
     * this button is complete, and it can now be displayed. */
    gtk_widget_show (button);

    /* Do these same steps again to create a second button */
    button = gtk_button_new_with_label ("Button 2");

    /* Call the same callback function with a different argument,
     * passing a pointer to "button 2" instead. */
    g_signal_connect (button, "clicked",
                      G_CALLBACK (callback), (gpointer) "button 2");

    gtk_box_pack_start(GTK_BOX (box1), button, TRUE, TRUE, 0);

    /* The order in which we show the buttons is not really important, but I
     * recommend showing the window last, so it all pops up at once. */
    gtk_widget_show (button);

    gtk_widget_show (box1);

    gtk_widget_show (window);

    /* Rest in gtk_main and wait for the fun to begin! */
    gtk_main ();

    return 0;
}
```

The Lisp implementation in Example 2.7 tries to be close to the C program. Therefore, the window and the box are created with the functions `gtk-window-new` and `gtk-box-new`. Various properties like the title of the window, the default size or the border width are set with the functions `gtk-window-set-title`, `gtk-window-set-default-size` and `gtk-container-set-border-width`. As described for Example 2.5 the function `gtk-widget-show-all` is used to display the window including all child widgets.

One main difference of the Lisp implementation is the use of the function `gtk-box-new` with an argument `:horizontal` to create a horizontal box. The `GtkHBox` widget which is used in the GTK+ 2.0 Tutorial is deprecated and is replaced by `GtkBox` with the property

`orientation`. More about boxes and their usages follows in Chapter 3 [Packing Widgets], page 21.

Example 2.7: Upgraded Hello world

```
(defun example-upgraded-hello-world ()
  (within-main-loop
    (let ((window (gtk-window-new :toplevel))
          (box (gtk-box-new :horizontal 6))
          (button  nil))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-window-set-title window "Hello Buttons")
      (gtk-window-set-default-size window 250 75)
      (gtk-container-set-border-width window 12)
      (setq button (gtk-button-new-with-label "Button 1"))
      (g-signal-connect button "clicked"
                        (lambda (widget)
                          (declare (ignore widget))
                          (format t "Button 1 was pressed.~%")))
      (gtk-box-pack-start box button :expand t :fill t :padding 0)
      (setq button (gtk-button-new-with-label "Button 2"))
      (g-signal-connect button "clicked"
                        (lambda (widget)
                          (declare (ignore widget))
                          (format t "Button 2 was pressed.~%")))
      (gtk-box-pack-start box button :expand t :fill t :padding 0)
      (gtk-container-add window box)
      (gtk-widget-show-all window))))
```

The second implementation in Example 2.8 makes more use of a Lisp style. The window is created with the Lisp function `make-instance`. All desired properties of the window are initialized by assigning values to the slots of the classes `gtk-window` and `gtk-box`. The Lisp implementation uses a lot keywords arguments with default values for long lists of arguments. In Example 2.8 the keyword arguments `expand`, `fill`, and `padding` of the function `gtk-box-pack-start` take their default values. In future examples of this tutorial the style shown in Example 2.8 is preferred. Furthermore, the C code is no longer presented for comparison.

Example 2.8: Second implementation of an Upgraded Hello World

```
(defun example-upgraded-hello-world-2 ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Hello Buttons"
                                 :default-width 250
```

```
                          :default-height 75
                          :border-width 12))
    (box (make-instance 'gtk-box
                        :orientation :horizontal
                        :spacing 6)))
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(let ((button (gtk-button-new-with-label "Button 1")))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (format t "Button 1 was pressed.~%")))
  (gtk-box-pack-start box button))
(let ((button (gtk-button-new-with-label "Button 2")))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (format t "Button 2 was pressed.~%")))
  (gtk-box-pack-start box button))
(gtk-container-add window box)
(gtk-widget-show-all window))))
```

2.7 Drawing

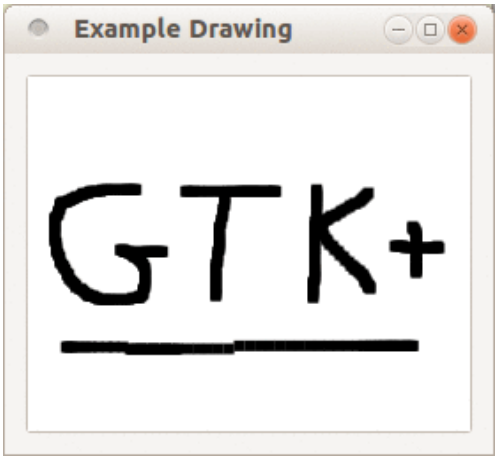

Figure 2.5: Drawing in response to input

Many widgets, like buttons, do all their drawing themselves. You just tell them the label you want to see, and they figure out what font to use, draw the button outline and focus rectangle, etc. Sometimes, it is necessary to do some custom drawing. In that case, a `GtkDrawingArea` might be the right widget to use. It offers a canvas on which you can draw by connecting to the "draw" signal.

The contents of a widget often need to be partially or fully redrawn, e.g. when another window is moved and uncovers part of the widget, or when tie window containing it is resized. It is also possible to explicitly cause part or all of the widget to be redrawn, by calling `gtk-widget-queue-draw` or its variants. GTK+ takes care of most of the details by providing a ready-to-use cairo context to the ::draw signal handler.

The following example shows a ::draw signal handler. It is a bit more complicated than the previous examples, since it also demonstrates input event handling by means of ::button-press and ::motion-notify handlers.

Example 2.9: Drawing in response to input

```
(let ((surface nil))
  (defun example-drawing ()
    (within-main-loop
      (let ((window (make-instance 'gtk-window
                                   :type :toplevel
                                   :title "Example Drawing"
                                   :border-width 12))
            (frame (make-instance 'gtk-frame
                                  :shadow-type :in))
            (area (make-instance 'gtk-drawing-area
                                 :width-request 250
                                 :height-request 200)))
        (g-signal-connect window "destroy"
                          (lambda (widget)
                            (declare (ignore widget))
                            (gtk-main-quit)))
        ;; Signals used to handle the backing surface
        (g-signal-connect area "draw"
           (lambda (widget cr)
             (declare (ignore widget))
             (cairo-set-source-surface (pointer cr) surface 0.0d0 0.0d0)
             (cairo-paint (pointer cr))
             nil))
        (g-signal-connect area "configure-event"
           (lambda (widget event)
             (declare (ignore event))
             (when surface
               (cairo-surface-destroy surface))
             (setq surface
                   (gdk-window-create-similar-surface
                                   (gtk-widget-get-window widget)
                                   :color
                                   (gtk-widget-get-allocated-width widget)
                                   (gtk-widget-get-allocated-height widget)))
             ;; Clear surface
             (let ((cr (cairo-create surface)))
```

```
        (cairo-set-source-rgb cr 1.0d0 1.0d0 1.0d0)
        (cairo-paint cr)
        (cairo-destroy cr))
      (format t "leave event 'configure-event'~%")
      t))
;; Event signals
(g-signal-connect area "motion-notify-event"
   (lambda (widget event)
     (format t "MOTION-NOTIFY-EVENT ~A~%" event)
     (when (member :button1-mask (event-motion-state event))
       (let ((cr (cairo-create surface))
             (x (event-motion-x event))
             (y (event-motion-y event)))
         (cairo-rectangle cr (- x 3.0d0) (- y 3.0d0) 6.0d0 6.0d0)
         (cairo-fill cr)
         (cairo-destroy cr)
         (gtk-widget-queue-draw-area widget
                                     (truncate (- x 3.0d0))
                                     (truncate (- y 3.0d0))
                                     6
                                     6)))
     ;; We have handled the event, stop processing
     t))
(g-signal-connect area "button-press-event"
   (lambda (widget event)
     (format t "BUTTON-PRESS-EVENT ~A~%" event)
     (if (eql 1 (event-button-button event))
         (let ((cr (cairo-create surface))
               (x (event-button-x event))
               (y (event-button-y event)))
           (cairo-rectangle cr (- x 3.0d0) (- y 3.0d0) 6.0d0 6.0d0)
           (cairo-fill cr)
           (cairo-destroy cr)
           (gtk-widget-queue-draw-area widget
                                       (truncate (- x 3.0d0))
                                       (truncate (- y 3.0d0))
                                       6
                                       6))
         ;; Clear surface
         (let ((cr (cairo-create surface)))
           (cairo-set-source-rgb cr 1.0d0 1.0d0 1.0d0)
           (cairo-paint cr)
           (cairo-destroy cr)
           (gtk-widget-queue-draw widget)))))
(gtk-widget-set-events area
                       (append (gtk-widget-get-events area)
                               '(:button-press-mask
```

```
                                    :pointer-motion-mask)))
    (gtk-container-add frame area)
    (gtk-container-add window frame)
    (gtk-widget-show-all window)))))
```

3 Packing Widgets

3.1 Packing Boxes

When creating an application, it is necessary to put more than one widget inside a window. The first Hello world example only used one button so it could simply use the function `gtk-container-add` to "pack" the button into the window. But when you want to put more than one widget into a window, how do you control where that widget is positioned? This is where packing comes in.

Most packing is done by creating boxes. These are invisible widget containers that can pack widgets into, which come in two forms, a horizontal box, and a vertical box. When packing widgets into a horizontal box, the objects are inserted horizontally from left to right or right to left depending on the call used. In a vertical box, widgets are packed from top to bottom or vice versa. You may use any combination of boxes inside or beside other boxes to create the desired effect.

To create a new box of the type `GtkBox`, the function `gtk-box-new` or the call `(make-instance 'gtk-box)` is used. The first argument of the function `gtk-box-new` takes a keyword of the enumeration type `GtkOrientation`, which is in the Lisp binding implemented as `gtk-orientation` with the values `:horizontal` or `:vertical` to determine a horizontal or a vertical box. Because `GtkBox` implements the interface `GtkOrientable` an instance of `GtkBox` has the property `orientiation` of type `GtkOrientation`, which can be set by a call of `make-instance`.

The following examples show two equivalent ways to create an instance of a horizontal box. The second argument of the function `gtk-box-new` is the value of the property `spacing`, which is described in Section 3.2 [Details of Boxes], page 22.

```
(let ((box (gtk-box-new :horizontal 3)))
  [...] )
```

or

```
(let ((box (make-instance 'gtk-box
                          :orientation :horizontal
                          :spacing 3)))
  [...] )
```

The `gtk-box-pack-start` and `gtk-box-pack-end` functions are used to place widgets inside of boxes. The `gtk-box-pack-start` function starts at the top and works its way down in a vertical box, and packs left to right in a horizontal box. The function `gtk-box-pack-end` does the opposite, packing from bottom to top in a vertical box, and right to left in a horizontal box. The widgets, which are packed into a box, can be containers, which are composed of other widgets. Using the functions for packing widgets in boxes allows to right justify or left justify the widgets. The functions can be mixed in any way to achieve the desired effect. Most of the examples in this tutorial use the function `gtk-box-pack-start`. In the following example a vertical box is created. Then two label widgets are packed into the box with the function `gtk-box-pack-start`.

```
(let ((box (gtk-box-new :vertical 3)))
  (gtk-box-pack-start box (gtk-label-new "LABEL 1"))
  (gtk-box-pack-start box (gtk-label-new "LABEL 2"))
```

```
  [...] )
```

By using boxes, GTK+ knows where to place the widgets so GTK+ can do automatic resizing and other nifty things. A number of options control as to how the widgets should be packed into boxes. This method of packing boxes gives the user quite a bit of flexibility when placing widgets.

Note:

The classes `GtkHBox` for horizontal and `GtkVBox` for vertical boxes are deprecated, but still present in GTK+ 3.4, which is the version used in this tutorial. In this tutorial these classes are not used. In addition a single-row or single-column `GtkGrid` provides exactly the same functionality as `GtkBox`. See Section 3.5 [Packing Using Grids], page 30 for examples to replace `GtkBox` with `GtkGrid`.

3.2 Details of Boxes

Because of the flexibility, packing boxes in GTK+ can be confusing at first. A lot of options control the packing of boxes, and it is not immediately obvious how the options all fit together. In the end, however, basically five different styles are available.

Boxes have the properties `homogeneous` and `spacing`. The property `homogeneous` controls whether each widget in the box has the same width in a horizontal box or the same height in a vertical box. The property `spacing` controls the amount of space between children in the box. A complete example for creating a box is therefore

```
(let ((box (make-instance 'gtk-box
                          :orientation :vertical
                          :spacing 3
                          :homogeneous t)))
  [...] )
```

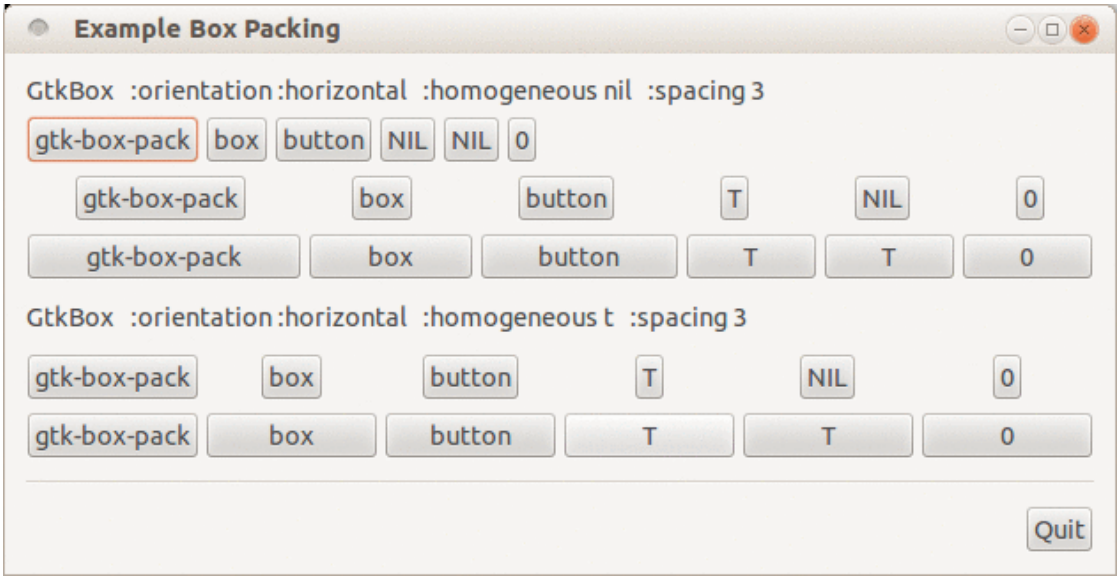

Figure 3.1: Example Box Packing with a spacing of 3

Figure 3.1 shows an example of packing buttons into horizontal boxes. The buttons have a small spacing of 3. Each line of the example contains one horizontal box with several buttons. The first button represents the call of the function `gtk-box-pack-start` and the following buttons represent the arguments of the function. The first two arguments are `box` for the box and `child` for the child widgets to put into the box, which are in our example buttons. The further arguments of `gtk-box-pack-start` are in the C implementation `expand`, `fill` and `padding`. In the Lisp binding to GTK+ these arguments are defined as the keyword arguments `:expand` and `:fill`, which both have a default value of `T`, and `:padding` with a default value of `0`. The keyword arguments can be omitted, when the default values should not be changed.

The keyword argument `:expand` with a value `T` to the functions `gtk-box-pack-start` and `gtk-box-pack-end` controls whether the widgets are laid out in the box to fill in all the extra space in the box so the box is expanded to fill the area allotted to it; or with a value `NIL` the box is shrunk to just fit the widgets. Setting expand to `NIL` allows to do right and left justification of the widgets. Otherwise, the widgets expand to fit into the box. The same effect can be achieved by using only one of the functions `gtk-box-pack-start` or `gtk_box_pack_end`.

The keyword argument `:fill` with a value `T` to the `gtk-box-pack` functions control whether the extra space is allocated to the objects themselves, or with a value `NIL` as extra padding in the box around these objects. It only has an effect if the keyword argument `expand` is also `T`.

The difference between spacing (set when the box is created) and padding (set when elements are packed) is, that spacing is added between objects, and padding is added on either side of a child widget.

The code for Figure 3.1 is shown in Example 3.1. The function `example-box-packing` takes an optional argument `spacing`, which has the default value `0` and controls the spacing of the buttons in the boxes.

The example uses two widgets which are not introduced up to now. The first one is the `GtkLabel` widget, which is described in Section 5.1 [Labels], page 47 and is used to display text. The second one is the `GtkSeparator` widget, which draws a horizontal or vertical line. The orientation of the line depends on the only argument to the function `gtk-separator-new`, which is of the enumeration type `GtkOrientation` with the values `:horizontal` for a horizontal line and `:vertical` for a vertical line.

Example 3.1: Example Packing Boxes

```
(defun make-box (homogeneous spacing expand fill padding)
  (let ((box (make-instance 'gtk-box
                            :orientation :horizontal
                            :homogeneous homogeneous
                            :spacing spacing)))
    (gtk-box-pack-start box
                        (gtk-button-new-with-label "gtk-box-pack")
                        :expand expand
                        :fill fill
                        :padding padding)
```

```
    (gtk-box-pack-start box
                        (gtk-button-new-with-label "box")
                        :expand expand
                        :fill fill
                        :padding padding)
    (gtk-box-pack-start box
                        (gtk-button-new-with-label "button")
                        :expand expand
                        :fill fill
                        :padding padding)
    (gtk-box-pack-start box
                        (if expand
                            (gtk-button-new-with-label "T")
                            (gtk-button-new-with-label "NIL"))
                        :expand expand
                        :fill fill
                        :padding padding)
    (gtk-box-pack-start box
                        (if fill
                            (gtk-button-new-with-label "T")
                            (gtk-button-new-with-label "NIL"))
                        :expand expand
                        :fill fill
                        :padding padding)
    (gtk-box-pack-start box
                        (gtk-button-new-with-label (format nil "~A" padding))
                        :expand expand
                        :fill fill
                        :padding padding)
    box))

(defun example-box-packing (&optional (spacing 0))
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Box Packing"
                                 :type :toplevel
                                 :border-width 12))
          (vbox (make-instance 'gtk-box
                               :orientation :vertical
                               :spacing 6))
          (button (make-instance 'gtk-button
                                 :label "Quit"))
          (quitbox (make-instance 'gtk-box
                                  :orientation :horizontal)))
      (g-signal-connect button "clicked"
                        (lambda (widget)
                          (declare (ignore widget))
```

```
                    (gtk-widget-destroy window)))
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(gtk-box-pack-start vbox
                    (make-instance 'gtk-label
                                   :label
                                   (format nil
                                    "GtkBox   ~
                                     :orientation :horizontal   ~
                                     :homogeneous nil   ~
                                     :spacing ~A"
                                    spacing)
                                   :xalign 0)
                    :expand nil)
(gtk-box-pack-start vbox
                    (make-box nil spacing nil nil 0)
                    :expand nil)
(gtk-box-pack-start vbox
                    (make-box nil spacing t nil 0)
                    :expand nil)
(gtk-box-pack-start vbox
                    (make-box nil spacing t t 0)
                    :expand nil)
(gtk-box-pack-start vbox
                    (make-instance 'gtk-label
                                   :label
                                   (format nil
                                      "GtkBox   ~
                                       :orientation :horizontal   ~
                                       :homogeneous t   ~
                                       :spacing ~A"
                                      spacing)
                                   :xalign 0)
                    :expand nil
                    :padding 6)
(gtk-box-pack-start vbox
                    (make-box t spacing t nil 0)
                    :expand nil)
(gtk-box-pack-start vbox
                    (make-box t spacing t t 0)
                    :expand nil)
(gtk-box-pack-start vbox
                    (gtk-separator-new :horizontal)
                    :expand nil
                    :padding 6)
```

```
;; Align the quit-button on the right side
(gtk-box-pack-end quitbox button :expand nil)
(gtk-box-pack-start vbox quitbox :expand nil)
(gtk-container-add window vbox)
(gtk-widget-show-all window))))
```

3.3 Packing Using Tables

Tables are another way of packing widgets and can be extremely useful in certain situations. Using tables a grid is created that widgets can placed in. The widgets may take up as many spaces as specified. Tables can be created with the function `gtk-table-new`. The function takes three arguments which set the properties of a table. Alternatively, the table is created with the function `make-instance`.

The first argument of `gtk-table-new` is the number of rows to make in the table, while the second is the number of columns. The last argument `homogeneous` has to do with how the boxes of the table are sized. If `homogeneous` is `T`, the table boxes are resized to the size of the largest widget in the table. If `homogeneous` is `NIL`, the size of a table boxes is dictated by the tallest widget in its same row, and the widest widget in its column. The rows and columns are laid out from `0` to `n`, where `n` is the number specified in the call to `gtk-table-new`. For `rows = 2` and `columns = 2`, the layout is shown in Figure 3.2. Note that the coordinate system starts in the upper left hand corner.

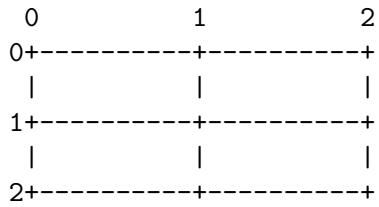

Figure 3.2: Layout of a 2 x 2 table

To place a widget into a table, the function `gtk-table-attach` can be used. The arguments are listed in Table 3.1. The first argument `table` is the table you have created and the second `child` the widget you wish to place into the table. The left and right attach arguments specify where to place the widget, and how many boxes to use. If you want a button in the lower right table entry of a 2 x 2 table, and want it to fill that entry only, `left-attach` is `= 1`, `right-attach = 2`, `top-attach = 1`, `bottom-attach = 2`. Now, if you wanted a widget to take up the whole top row of a 2 x 2 table, you would use `left-attach = 0`, `right-attach = 2`, `top-attach = 0`, `bottom-attach = 1`.

Table 3.1: Arguments of the function `gtk-table-attach`

`table` The GtkTable to add a new widget to.

`child` The widget to add.

`left-attach`
The column number to attach the left side of a child widget to.

`right-attach`
The column number to attach the right side of a child widget to.

`top-attach`
: The row number to attach the top of a child widget to.

`bottom-attach`
: The row number to attach the bottom of a child widget to.

`:xoptions`
: Used to specify the properties of the child widget when the table is resized. The default value is `'(:expand :fill)`.

`:yoptions`
: The same as `xoptions`, except this field determines behavior of vertical resizing. The default value is `'(:expand :fill)`.

`:xpadding`
: An integer value specifying the padding on the left and right of the widget being added to the table. The default value is `0`.

`:ypadding`
: The amount of padding above and below the child widget. The default value is `0`.

The arguments `:xoptions` and `:yoptions` are of the enumeration type `GtkAttachOptions` and used to specify packing options. The packing options can be OR'ed together to allow multiple options. In the Lisp binding a list of options is used to combine multiple options. Possible values of the enumeration type `GtkAttachOptions` are listed in Table 3.2.

Padding is just like in boxes, creating a clear area around the widget specified in pixels and is controlled with the arguments `:xpadding` and `:ypadding`.

Table 3.2: Values of the type `GtkAttachOptions`

`:fill`	If the table box is larger than the widget, and `:fill` is specified, the widget will expand to use all the room available.
`:shrink`	If the table widget was allocated less space then was requested (usually by the user resizing the window), then the widgets would normally just be pushed off the bottom of the window and disappear. If `:shrink` is specified, the widgets will shrink with the table.
`:expand`	This will cause the table to expand to use up any remaining space in the window.

In the Lisp binding the arguments `:xoptions`, `:yoptions`, `:xpadding`, and `:ypadding` of the function `gtk-table-attach` are defined as keyword arguments with default values as shown in Table 3.1. In the C library this is realized with a second function `gtk_table_attach_defaults()`. In the Lisp binding the function `gtk-table-attach-defaults` is a second equivalent implementation of `gtk-table-attach`, when using the default values of the keyword arguments.

The functions `gtk-table-set-row-spacing` and `gtk-table-set-col-spacing` places spacing between the rows at the specified row or column. The first argument of the functions is a `GtkTable`, the second argument a row or a column and the third argument the spacing.

Note that for columns, the space goes to the right of the column, and for rows, the space goes below the row.

You can also set a consistent spacing of all rows and columns with the functions `gtk-table-set-row-spacings` and `gtk-table-set-col-spacings`. Both functions take a `GtkTable` as the first argument and the desired spacing `spacing` as the second argument. Note that with these calls, the last row and last column do not get any spacing.

Note:

`GtkTable` has been deprecated since GTK+ 3.4. It is recommended to use `GtkGrid` instead. `GtkGrid` provides the same capabilities as `GtkTable` for arranging widgets in a rectangular grid, but does support height-for-width geometry management, which is newly introduced for widgets in GTK+ 3. This chapter will vanish in the near future.

3.4 Table Packing Example

Figure 3.3 is a window with three buttons in a 2 x 2 table. The first two buttons are placed in the upper row. A third, quit button, is placed in the lower row, spanning both columns. The code of this example is shown in Example 3.2.

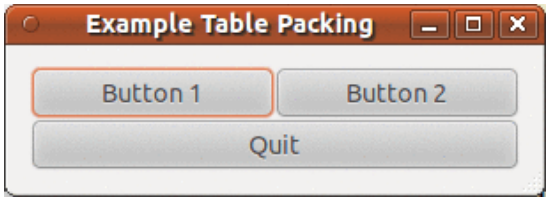

Figure 3.3: Table packing

Example 3.2: Table Packing

```
(defun example-table-packing ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Table Packing"
                                 :border-width 12
                                 :default-width 300))
          (table (make-instance 'gtk-table
                                :n-columns 2
                                :n-rows 2
                                :homogeneous t))
          (button1 (make-instance 'gtk-button
                                  :label "Button 1"))
          (button2 (make-instance 'gtk-button
                                  :label "Button 2"))
          (quit (make-instance 'gtk-button
                               :label "Quit")))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
```

```
                    (gtk-main-quit)))
    (g-signal-connect quit "clicked"
                      (lambda (widget)
                        (declare (ignore widget))
                        (gtk-widget-destroy window)))
    (gtk-table-attach table button1 0 1 0 1)
    (gtk-table-attach table button2 1 2 0 1)
    (gtk-table-attach table quit    0 2 1 2)
    (gtk-container-add window table)
    (gtk-widget-show window))))
```

Figure 3.4 is an extended example to show the possibility to increase the spacing of the rows and columns. This is implemented through two toggle buttons which increase and decrease the spacings. Toggle buttons are described in Section 4.2 [Toggle Buttons], page 39 later in this tutorial. The code of Figure 3.4 is shown in Example 3.3.

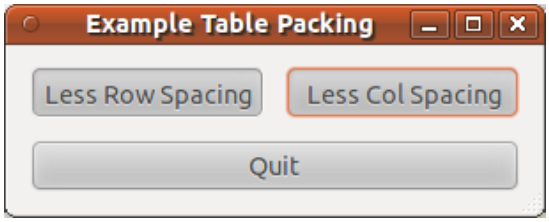

Figure 3.4: Table packing with more spacing

Example 3.3: Table Packing with more spacing

```
(defun example-table-packing-2 ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Table Packing"
                                 :border-width 12
                                 :default-width 300))
          (table (make-instance 'gtk-table
                                :n-columns 2
                                :n-rows 2
                                :homogeneous t))
          (button1 (make-instance 'gtk-toggle-button
                                  :label "More Row Spacing"))
          (button2 (make-instance 'gtk-toggle-button
                                  :label "More Col Spacing"))
          (quit (make-instance 'gtk-button
                               :label "Quit")))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (g-signal-connect button1 "toggled"
```

```
   (lambda (widget)
     (if (gtk-toggle-button-get-active widget)
         (progn
           (gtk-table-set-row-spacings table 12)
           (gtk-button-set-label widget "Less Row Spacing"))
         (progn
           (gtk-table-set-row-spacings table 0)
           (gtk-button-set-label widget "More Row Spacing")))))
(g-signal-connect button2 "toggled"
   (lambda (widget)
     (if (gtk-toggle-button-get-active widget)
         (progn
           (gtk-table-set-col-spacings table 12)
           (gtk-button-set-label widget "Less Col Spacing"))
         (progn
           (gtk-table-set-col-spacings table 0)
           (gtk-button-set-label widget "More Col Spacing")))))
(g-signal-connect quit "clicked"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-widget-destroy window)))
(gtk-table-attach table button1 0 1 0 1)
(gtk-table-attach table button2 1 2 0 1)
(gtk-table-attach table quit    0 2 1 2)
(gtk-container-add window table)
(gtk-widget-show window))))
```

3.5 Packing Using Grids

`GtkGrid` is an attempt to write a comprehensive, legacy-free, box-layout container that is flexible enough to replace `GtkBox`, `GtkTable` and the like.

The layout model of `GtkGrid` is to arrange its children in rows and columns. This is done by assigning positions on a two-dimensions grid that stretches arbitrarily far in all directions. Children can span multiple rows or columns.

3.5.1 GtkBox versus GtkGrid - packing

`GtkBox` works by arranging child widgets in a single line, either horizontally or vertically. It allows packing children from the beginning or end, using `gtk-box-pack-start` and `gtk-box-pack-end`.

The following code creates a simple box with two labels:

```
(let ((box (gtk-box-new :horizontal 0)))
  (gtk-box-pack-start box (gtk-label-new "ONE") nil nil 0)
  (gtk-box-pack-start box (gtk-label-new "TWO") nil nil 0)
  [...] )
```

This can be done with `GtkGrid` as follows:

```
(let ((grid (gtk-grid-new))
```

```
      (child1 (gtk-label-new "ONE"))
      (child2 (gtk-label-new "TWO")))
  (gtk-grid-attach grid child1 0 0 1 1)
  (gtk-grid-attach-next-to grid child2 child1 :right 1 1)
  [...]  )
```

And similarly for `gtk-box-pack-end`. In that case, you would use `:left` to place the grid children from left to right.

If you only need to pack children from the start, using `gtk-container-add` is an even simpler alternative. `GtkGrid` places children added with `gtk-container-add` in a single row or column according to its "orientation".

One difference to keep in mind is that the `gtk-box-pack-start` and `gtk-box-pack-end` functions allow you to place an arbitrary number of children from either end without ever 'colliding in the middle'. With `GtkGrid`, you have to leave enough space between the two ends, if you want to combine packing from both ends towards the middle. In practice, this should be easy to avoid; and `GtkGrid` simply ignores entirely empty rows or columns for layout and spacing.

On the other hand, `GtkGrid` is more flexible in that its grid extends indefinitively in both directions there is no problem with using negative numbers for the grid positions. So, if you discover that you need to place a widget before your existing arrangement, you always can.

3.5.2 GtkBox versus GtkGrid: sizing

When adding a child to a `GtkBox`, there are two hard-to-remember parameters (child properties, more exactly) named `expand` and `fill` that determine how the child size behaves in the main direction of the box. If expand is set, the box allows the position occupied by the child to grow when extra space is available. If fill is also set, the extra space is allocated to the child widget itself. Otherwise it is left 'free'. There is no control about the 'minor' direction; children are always given the full size in the minor direction.

`GtkGrid` does not have any custom child properties for controlling size allocation to children. Instead, it fully supports the newly introduced `hexpand`, `vexpand`, `halign` and `valign` properties for widgets.

The `hexpand` and `vexpand` properties operate in a similar way to the expand child properties of `GtkBox`. As soon as a column contains a hexpanding child, `GtkGrid` allows the column to grow when extra space is available (similar for rows and vexpand). In contrast to `GtkBox`, all the extra space is always allocated to the child widget, there are no 'free' areas.

To replace the functionality of the fill child properties, you can set the `halign` and `valign` properties. An align value of `:fill` has the same effect as setting fill to `T`, a value of `:center` has the same effect as setting fill to FALSE.

Expansion and alignment with `GtkBox`:

```
(let ((box (gtk-box-new :horizontal 0)))
  (gtk-box-pack-start box (gtk-label-new "ONE") t nil 0)
  (gtk-box-pack-start box (gtk-label-new "TWO") t t 0)
  [...] )
```

This can be done with `GtkGrid` as follows:

```
(let ((grid (gtk-grid-new))
      (child1 (make-instance 'gtk-label
                             :label "ONE"
                             :hexpand t
                             :halign :center))
      (child2 (make-instance 'gtk-label
                             :label "TWO"
                             :hexpand t
                             :halign :fill))
  (gtk-grid-attach grid child1 0 0 1 1)
  (gtk-grid-attach-next-to grid child2 child1 :right 1 1)
  [...] )
```

One difference between the new `GtkWidget` expand properties and the `GtkBox` child property of the same name is that widget expandability is 'inherited' from children. What this means is that a container will become itself expanding as soon as it has an expanding child. This is typically what you want, it lets you e.g. mark the content pane of your application window as expanding, and all the intermediate containers between the content pane and the toplevel window will automatically do the right thing. This automatism can be overridden at any point by setting the expand flags on a container explicitly.

Another difference between `GtkBox` and `GtkGrid` with respect to expandability is when there are no expanding children at all. In this case, `GtkBox` will forcibly expand all children whereas `GtkGrid` will not. In practice, the effect of this is typically that a grid will 'stick to the corner' when the toplevel containing it is grown, instead of spreading out its children over the entire area. The problem can be fixed by setting some or all of the children to expand.

When you set the `homogeneous` property on a `GtkBox`, it reserves the same space for all its children. `GtkGrid` does this in a very similar way, with `row-homogeneous` and `column-homogeneous` properties which control whether all rows have the same height and whether all columns have the same width.

3.5.3 GtkBox versus GtkGrid - spacing

With `GtkBox`, you have to specify the "spacing" when you construct it. This property specifies the space that separates the children from each other. Additionally, you can specify extra space to put around each child individually, using the `padding` child property.

`GtkGrid` is very similar when it comes to spacing between the children, except that it has two separate properties, `row-spacing` and `column-spacing`, for the space to leave between rows and columns. Note that row-spacing is the space between rows, not inside a row. So, if you doing a horizontal layout, you need to set `column-spacing`.

`GtkGrid` does not have any custom child properties to specify per-child padding; instead you can use the `margin` property. You can also set different padding on each side with the `margin-left`, `margin-right`, `margin-top` and `margin-bottom` properties.

Example with spacing in boxes:

```
(let ((box (gtk-box-new :vertical 6))
      (child (gtk-label-new "Child")))
  (gtk-box-pack-start box child nil nil 12)
```

```
  [...] )
```

This can be done with `GtkGrid` as follows:

```
(let ((grid (gtk-grid-new))
      (child (make-instance 'gtk-label
                            :label "Child"
                            :margin 12)))
  (gtk-grid-attach box child 0 0 1 1)
  [...] )
```

3.6 Grid Packing Examples

We repeat the implementation of Example 3.1 and Example 3.2 using `GtkGrid`. The first Example 3.4 shows how to replace `GtkBox` wit `GtkGrid` to create vertical and horizontal boxes. In the second Example 3.5 `GtkTable` is replaced with `GtkGrid`. See Figure 3.1 and Figure 3.3 for the output of the example programs.

Example 3.4: Example 3.1 using `GtkGrid`

```
(defun make-grid (homogeneous spacing expand align margin)
  (let ((box (make-instance 'gtk-grid
                            :orientation :horizontal
                            :column-homogeneous homogeneous
                            :column-spacing spacing)))
    (gtk-container-add box
                       (make-instance 'gtk-button
                                      :label "gtk-container-add"
                                      :hexpand expand
                                      :halgin align
                                      :margin margin))
    (gtk-container-add box
                       (make-instance 'gtk-button
                                      :label "box"
                                      :hexpand expand
                                      :halign align
                                      :margin margin))
    (gtk-container-add box
                       (make-instance 'gtk-button
                                      :label "button"
                                      :hexpand expand
                                      :halign align
                                      :margin margin))
    (gtk-container-add box
                       (make-instance 'gtk-button
                                      :label (if expand "T" "NIL")
                                      :hexpand expand
                                      :halign align
                                      :margin margin))
```

```
    (gtk-container-add box
                       (make-instance 'gtk-button
                                      :label (format nil "~A" align)
                                      :hexpand expand
                                      :halign align
                                      :margin margin))
    (gtk-container-add box
                       (make-instance 'gtk-button
                                      :label (format nil "~A" margin)
                                      :hexpand expand
                                      :halign align
                                      :margin margin))
    box))

(defun example-grid-packing (&optional (spacing 0))
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Grid Packing"
                                 :type :toplevel
                                 :border-width 12
                                 :default-height 200
                                 :default-width 300))
          (vbox (make-instance 'gtk-grid
                               :orientation :vertical
                               :row-spacing 6))
          (button (make-instance 'gtk-button
                                 :label "Quit"))
          (quitbox (make-instance 'gtk-box
                                  :orientation :horizontal)))
      (g-signal-connect button "clicked"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-widget-destroy window)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-container-add vbox
                         (make-instance 'gtk-label
                                        :label
                                        (format nil
                                         "GtkGrid homogeneous nil spacing ~A"
                                         spacing)
                                        :xalign 0
                                        :yalign 0
                                        :vexpand nil
                                        :valign :start))
```

```
      (gtk-container-add vbox (gtk-separator-new :horizontal))
      (gtk-container-add vbox (make-grid nil spacing nil :center 0))
      (gtk-container-add vbox (make-grid nil spacing t :center 0))
      (gtk-container-add vbox (make-grid nil spacing t :fill 0))
      (gtk-container-add vbox (gtk-separator-new :horizontal))
      (gtk-container-add vbox
                         (make-instance 'gtk-label
                                        :label
                                        (format nil
                                           "GtkGrid homogeneous t spacing ~A"
                                           spacing)
                                         :xalign 0
                                         :yalign 0
                                         :vexpand nil
                                         :valign :start
                                         :margin 6))
      (gtk-container-add vbox (gtk-separator-new :horizontal))
      (gtk-container-add vbox (make-grid t spacing t :center 0))
      (gtk-container-add vbox (make-grid t spacing t :fill 0))
      (gtk-container-add vbox (gtk-separator-new :horizontal))
      (gtk-container-add quitbox button)
      (gtk-container-add vbox quitbox)
      (gtk-container-add window vbox)
      (gtk-widget-show-all window))))
```

Example 3.5: Example 3.2 using `GtkGrid`

```
(defun example-grid-packing-2 ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Grid Packing"
                                 :border-width 12
                                 :default-width 300))
          (grid (make-instance 'gtk-grid
                               :column-homogeneous t
                               :row-homogeneous t))
          (button1 (make-instance 'gtk-button
                                  :label "Button 1"))
          (button2 (make-instance 'gtk-button
                                  :label "Button 2"))
          (quit (make-instance 'gtk-button
                               :label "Quit")))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
```

```
(g-signal-connect quit "clicked"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-widget-destroy window)))
(gtk-grid-attach grid button1 0 1 1 1)
(gtk-grid-attach grid button2 1 1 1 1)
(gtk-grid-attach grid quit    0 2 2 1)
(gtk-container-add window grid)
(gtk-widget-show-all window))))
```

4 Button Widgets

4.1 Normal Buttons

We have almost seen all there is to see of the button widget. The button widget is pretty simple. There is however more than one way to create a button. You can use the the function `gtk-button-new-with-label` or the function `gtk-button-new-with-mnemonic` to create a button with a label, use `gtk-button-new-from-stock` to create a button containing the image and text from a stock item or use `gtk-button-new` to create a blank button. It is then up to you to pack a label or pixmap into this new button. To do this, create a new box, and then pack your objects into this box using the function `gtk-box-pack-start`, and then use the function `gtk-container-add` to pack the box into the button.

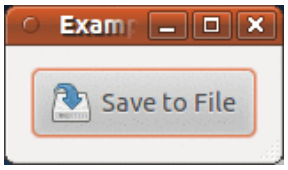

Figure 4.1: Button with an image from a file

Figure 4.1 is an example of using `gtk-button-new` to create a button with an image and a label in it. The code to create a box is shown in Example 4.2 and breaken up from the rest so you can use it in your programs. The main program which uses this subroutine is shown in Example 4.1.

The `image-label-box` function could be used to pack images and labels into any widget that can be a container.

Figure 4.2 shows more buttons, which are created with standard functions and with the function `make-instance`. To get buttons which show both a label and an image the global setting of the property `gtk-button-images` has to be set to the value `T`. The code of Figure 4.2 is shown in Example 4.3.

Example 4.1: A button with an image and a label

```
(defun example-button ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Cool Button"
                                 :type :toplevel
                                 :border-width 12))
          (button (make-instance 'gtk-button))
          (box (image-label-box "save.png" "Save to File")))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-container-add button box)
      (gtk-container-add window button)
```

```
      (gtk-widget-show-all window))))
```

Example 4.2: Code to create a button with an image and a label

```
(defun image-label-box (filename text)
  (let ((box (make-instance 'gtk-box
                            :orientation :horizontal
                            :border-width 3))
        (label (make-instance 'gtk-label
                              :label text))
        (image (gtk-image-new-from-file filename)))
    (gtk-box-pack-start box image :expand nil :fill nil :padding 3)
    (gtk-box-pack-start box label :expand nil :fill nil :padding 3)
    box))
```

Figure 4.2: More Examples to create buttons

Example 4.3: More buttons

```
(defun example-buttons ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Buttons"
                                 :type :toplevel
                                 :default-width 250
                                 :border-width 12))
          (vbox1 (make-instance 'gtk-box
                                :orientation :vertical
                                :spacing 6))
          (vbox2 (make-instance 'gtk-box
                                :orientation :vertical
                                :spacing 6))
          (hbox  (make-instance 'gtk-box
                                :orientation :horizontal
                                :spacing 6)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
```

```
                    (gtk-main-quit)))
;; Set gtk-button-images to T. This allows buttons with text and image.
(setf (gtk-settings-gtk-button-images (gtk-settings-get-default)) t)
;; These are the standard functions to create a button.
(gtk-box-pack-start vbox1
                    (gtk-button-new-with-label "Label"))
(gtk-box-pack-start vbox1
                    (gtk-button-new-with-mnemonic "_Mnemonic"))
(gtk-box-pack-start vbox1
                    (gtk-button-new-from-stock "gtk-apply"))
;; Create some buttons with make-instance.
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-button
                                   :image-position :right
                                   :image
                                   (gtk-image-new-from-stock "gtk-edit"
                                                             :button)
                                   :label "gtk-edit"
                                   :use-stock t))
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-button
                                   :image-position :top
                                   :image
                                   (gtk-image-new-from-stock "gtk-cut"
                                                             :button)
                                   :label "gtk-cut"
                                   :use-stock t))
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-button
                                   :image-position :bottom
                                   :image
                                   (gtk-image-new-from-stock
                                                          "gtk-cancel"
                                                          :button)
                                   :label "gtk-cancel"
                                   :use-stock t))
(gtk-box-pack-start hbox vbox1)
(gtk-box-pack-start hbox vbox2)
(gtk-container-add window hbox)
(gtk-widget-show-all window))))
```

4.2 Toggle Buttons

Toggle buttons are derived from normal buttons and are very similar, except toggle buttons always are in one of two states, alternated by a click. Toggle buttons can be depressed, and when clicked again, the toggle button will pop back up. Toggle buttons are the basis

for check buttons and radio buttons, as such, many of the calls used for toggle buttons are inherited by radio and check buttons.

Toggle buttons can be created with the functions `gtk-toggle-button-new`, `gtk-toggle-button-new-with-label`, and `gtk-toggle-button-new-with-mnemonic`. The first function creates a blank toggle button, and the last two functions, a toggle button with a label widget already packed into it. The `gtk-toggle-button-new-with-mnemonic` variant additionally parses the label for '_'-prefixed mnemonic characters.

To retrieve the state of the toggle widget, including radio and check buttons, a construct as shown in the example below is used. This tests the state of the toggle button, by accessing the active field of the toggle widget's structure. The signal of interest to us emitted by toggle buttons (the toggle button check button, and radio button widgets) is the `"toggled"` signal. To check the state of these buttons, set up a signal handler to catch the toggled signal, and access the property `active` to determine the state of the button. A signal handler will look something like:

```
(g-signal-connect button "toggled"
   (lambda (widget)
     (if (gtk-toggle-button-get-active widget)
         (progn
           ;; If control reaches here, the toggle button is down
         )
        (progn
           ;; If control reaches here, the toggle button is up
         ))))
```

To force the state of a toggle button, and its children, the radio and check buttons, use this function `gtk-toggle-button-set-active`. This function can be used to set the state of the toggle button, and its children the radio and check buttons. Passing in your created button as the first argument, and a `T` or `NIL` for the second state argument to specify whether it should be down (depressed) or up (released). Default is up, or `NIL`.

Note that when you use the `gtk-toggle-button-set-active` function, and the state is actually changed, it causes the `"clicked"` and `"toggled"` signals to be emitted from the button. The current state of the toggle button as a boolean `T` or `NIL` value is returned from the function `gtk-toggle-button-get-active`.

In Example 3.3 the usage of toggle buttons is shown.

4.3 Check Buttons

Check buttons inherit many properties and functions from the toggle buttons above, but look a little different. Rather than being buttons with text inside them, they are small squares with the text to the right of them. These are often used for toggling options on and off in applications.

The creation functions are similar to those of the normal button: `gtk-check-button-new`, `gtk-check-button-new-with-label`, and `gtk-check-button-new-with-mnemonic`. The `gtk-check-button-new-with-label` function creates a check button with a label beside it.

Checking the state of the check button is identical to that of the toggle button. Figure 4.3 shows toggle buttons and Example 4.4 the code to create toggle buttons.

4.4 Radio Buttons

Radio buttons are similar to check buttons except they are grouped so that only one may be selected or depressed at a time. This is good for places in your application where you need to select from a short list of options.

Creating a new radio button is done with one of these calls: `gtk-radio-button-new`, `gtk-radio-button-new-with-label`, and `gtk-radio-button-new-with-mnemonic`. These functions take a list of radio buttons as the first argument or `NIL`. When `NIL` a new list of radio buttons is created. The newly created list for the radio buttons can be get with the function `gtk-radio-button-get-group`. More radio buttons can then be added to this list. The important thing to remember is that `gtk-radio-button-get-group` must be called for each new button added to the group, with the previous button passed in as an argument. The result is then passed into the next call to `gtk-radio-button-new` or the other two functions for creating a radio button. This allows a chain of buttons to be established. Example 4.4 creates a radio button group with three buttons.

You can shorten this slightly by using the following syntax, which removes the need for a variable to hold the list of buttons:

```
(setq button
      (gtk-radio-button-new-with-label (gtk-radio-button-get-group button)
                                       "Button"))
```

Each of these functions has a variant, which take a radio button as the first argument and allows to omit the `gtk-radio-button-get-group` call. In this case the new radio button is added to the list of radio buttons the argument is already a part of. These functions are: `gtk-radio-button-new-from-widget`, `gtk-radio-button-new-with-label-from-widget`, and `gtk-radio-button-new-with-mnemonic-from-widget`.

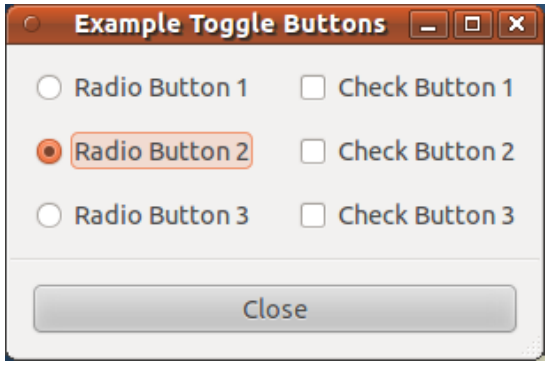

Figure 4.3: Toggle Buttons

It is also a good idea to explicitly set which button should be the default depressed button with the function `gtk-toggle-button-set-active`. This is described in the section on toggle buttons, and works in exactly the same way. Once the radio buttons are grouped together, only one of the group may be active at a time. If the user clicks on one radio button, and then on another, the first radio button will first emit a `"toggled"` signal (to report becoming inactive), and then the second will emit its `"toggled"` signal (to report becoming active).

Example 4.4: Radio and Toggle Buttons

```
(defun example-toggle-buttons ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Toggle Buttons"
                                 :type :toplevel))
          (vbox (make-instance 'gtk-box
                               :orientation :vertical))
          (hbox (make-instance 'gtk-box
                               :orientation :horizontal)))
      ;; Handler for the signal "destroy"
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      ;; Create three radio buttons and put the buttons in a vbox
      (let ((vbox (make-instance 'gtk-box
                                 :orientation :vertical
                                 :spacing 12
                                 :border-width 12))
            (button (gtk-radio-button-new-with-label nil "Radio Button 1")))
        (gtk-box-pack-start vbox button)
        (setq button
              (gtk-radio-button-new-with-label
                                        (gtk-radio-button-get-group button)
                                        "Radio Button 2"))
        (gtk-toggle-button-set-active button t)
        (gtk-box-pack-start vbox button)
        (setq button
              (gtk-radio-button-new-with-mnemonic
                                        (gtk-radio-button-get-group button)
                                        "_Radio Button 3"))
        (gtk-box-pack-start vbox button)
        ;; Put the vbox with the radio buttons in a hbox
        (gtk-box-pack-start hbox vbox :expand nil :fill nil))
      ;; Create three check buttons and put the buttons in a vbox
      (let ((vbox (make-instance 'gtk-box
                                 :orientation :vertical
                                 :homogenous nil
                                 :spacing 12
                                 :border-width 12)))
        (gtk-box-pack-start
                         vbox
                         (gtk-check-button-new-with-label "Check Button 1"))
        (gtk-box-pack-start
```

```
                    vbox
                    (gtk-check-button-new-with-label "Check Button 2"))
  (gtk-box-pack-start
                    vbox
                    (gtk-check-button-new-with-label "Check Button 3"))
  ;; Put the vbox with the buttons in a hbox
  (gtk-box-pack-start hbox vbox :expand nil :fill nil))
;; Put the hbox in a vbox
(gtk-box-pack-start vbox hbox :expand nil :fill nil)
;; Add a separator to the vbox
(gtk-box-pack-start vbox
                    (make-instance 'gtk-separator
                                   :orientation :horizontal)
                    :expand nil :fill nil)
;; Add a quit button to the vbox
(let ((vbox-quit (make-instance 'gtk-box
                                :orientation :vertical
                                :spacing 12
                                :border-width 12))
      (button (make-instance 'gtk-button :label "Close")))
  (gtk-box-pack-start vbox-quit button :expand nil :fill nil)
  (gtk-box-pack-start vbox vbox-quit :expand nil)
  (g-signal-connect button "clicked"
                    (lambda (button)
                      (declare (ignore button))
                      (gtk-widget-destroy window))))
;; Put the vbox in the window widget
(gtk-container-add window vbox)
(gtk-widget-show-all window))))
```

4.5 Link Buttons

Figure 4.4: Link Buttons

A `GtkLinkButton` is a `GtkButton` with a hyperlink, similar to the one used by web browsers, which triggers an action when clicked. It is useful to show quick links to resources.

A link button is created by calling either `gtk-link-button-new` or `gtk-link-button-new-with-label`. If using the former, the URI you pass to the constructor is used as a label for the widget.

The URI bound to a `GtkLinkButton` can be set specifically using `gtk-link-button-set-uri`, and retrieved using `gtk-link-button-get-uri`.

By default, `GtkLinkButton` calls `gtk-show-uri` when the button is clicked. This behaviour can be overridden by connecting to the `"activate-link"` signal and returning `T` from the signal handler.

Figure 4.4 shows two different styles of link buttons. The code is shown in Example 4.5.

Example 4.5: Link Buttons

```
(defun example-link-button ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Link Button"
                                 :default-width 270
                                 :border-width 12))
          (grid (make-instance 'gtk-grid
                               :orientation :vertical
                               :row-spacing 6
                               :column-homogeneous t)))
      (g-signal-connect window "destroy"
                               (lambda (widget)
                                 (declare (ignore widget))
                                 (gtk-main-quit)))
      (gtk-container-add grid
                         (make-instance 'gtk-label
                                        :use-markup t
                                        :label
                                        "<b>Link Button with url</b>"))
      (gtk-container-add grid
                         (gtk-link-button-new "http://www.gtk.org/"))
      (gtk-container-add grid
                         (make-instance 'gtk-label
                                        :use-markup t
                                        :label
                                        "<b>Link Button with Label</b>"))
      (gtk-container-add grid
                         (gtk-link-button-new-with-label
                                                 "http://www.gtk.org/"
                                                 "Project WebSite"))
      (gtk-container-add window grid)
      (gtk-widget-show-all window))))
```

4.6 Switches

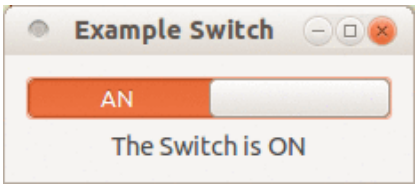

Figure 4.5: Switch

`GtkSwitch` is a widget that has two states: on or off. The user can control which state should be active by clicking the empty area, or by dragging the handle. The switch is created with the function `gtk-switch-new` or the call `(make-instance 'gtk-switch)`.

`GtkSwitch` has the property `active`, which can be set with the function `gtk-switch-set-active` or retrieved with the function `gtk-switch-get-active`.

An example of a switch is shown in Figure 4.5. The code is shown in Example 4.6. Note that in the example the signal "notify::active" is connected to the switch to display a label with the state of the switch.

Example 4.6: Switches

```
(defun example-switch ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Switch"
                                 :default-width 230
                                 :border-width 12))
          (switch (make-instance 'gtk-switch
                                 :active t))
          (label (make-instance 'gtk-label
                                :label "The Switch is ON"))
          (grid (make-instance 'gtk-grid
                               :orientation :vertical
                               :row-spacing 6
                               :column-homogeneous t)))
      (g-signal-connect window "destroy"
                               (lambda (widget)
                                 (declare (ignore widget))
                                 (gtk-main-quit)))
      (g-signal-connect switch "notify::active"
         (lambda (widget param)
           (declare (ignore param))
           (if (gtk-switch-get-active widget)
               (gtk-label-set-label label "The Switch is ON")
               (gtk-label-set-label label "The Switch is OFF"))))
      (gtk-container-add grid switch)
      (gtk-container-add grid label)
```

```
(gtk-container-add window grid)
(gtk-widget-show-all window))))
```

5 Display Widgets

5.1 Labels

Labels are used a lot in GTK+, and are relatively simple. The `GtkLabel` widget displays a small amount of text. As the name implies, most labels are used to label another widget such as a `GtkButton`, a `GtkMenuItem`, or a `GtkOptionMenu`. Labels emit no signals as they do not have an associated X window. If you need to catch signals, or do clipping, place it inside a `GtkEventBox` widget or a Button widget.

To create a new label, use `make-instance` with the class name `gtk-label` or the functions `gtk-label-new` or `gtk-label-new-with-mnemonic`. The sole argument of the functions is the string you wish the label to display. To change the text of the label after creation, use the function `gtk-label-set-text`. The first argument is the label you created previously, and the second is the new string. The space needed for the new string will be automatically adjusted if needed. You can produce multi-line labels by putting line breaks in the label string. To retrieve the current string, use `gtk-label-get-text`.

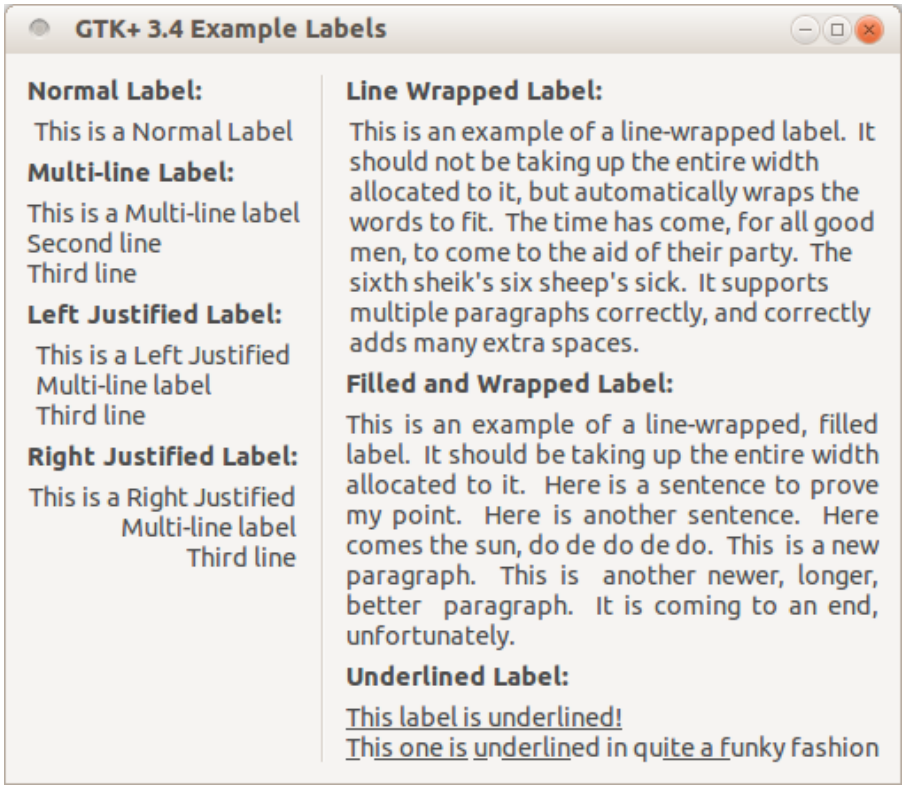

Figure 5.1: Labels

Label with Mnemonics

Labels may contain mnemonics. Mnemonics are underlined characters in the label, used for keyboard navigation. Mnemonics are created by providing a string with an underscore

before the mnemonic character, such as "_File", to the functions gtk-label-new-with-mnemonic or gtk-label-set-text-with-mnemonic.

Mnemonics automatically activate any activatable widget the label is inside, such as a GtkButton; if the label is not inside the mnemonic's target widget, you have to tell the label about the target using gtk-label-set-mnemonic-widget.

Here's a simple example where the label is inside a button:

```
;; Pressing Alt+H will activate this button
(let* ((button (gtk-button-new))
       (label (gtk-label-new-with-mnemonic "_Hello")))
  (gtk-container-add button label)
  [...] )
```

There's a convenience function to create buttons with a mnemonic label already inside:

```
;; Pressing Alt+H will activate this button
(let ((button (gtk-button-new-with-mnemonic "_Hello")))
  [...] )
```

To create a mnemonic for a widget alongside the label, such as a GtkEntry, you have to point the label at the entry with gtk-label-set-mnemonic-widget:

```
;; Pressing Alt+H will focus the entry
(let* ((entry (gtk-entry-new))
       (label (gtk-label-new-with-mnemonic "_Hello")))
  (gtk-label-set-mnemonic-widget label entry)
  [...] )
```

Markup (styled text)

To make it easy to format text in a label (changing colors, fonts, etc.), label text can be provided in a simple markup format. Here's how to create a label with a small font:

```
(let ((label (gtk-label-new)))
  (gtk-label-set-markup label
                        "<small>Small text</small>")
  [...] )
```

or

```
(let ((label (make-instance 'gtk-label
                            :use-markup t
                            :label "<small>Small text</small>")))
  [...] )
```

(See complete documentation of available tags in the Pango manual.)

The markup passed to gtk-label-set-markup must be valid; for example, literal <, > and & characters must be escaped as \<, \gt;, and \&. If you pass text obtained from the user, file, or a network to gtk-label-set-markup, you'll want to escape it with g-markup-escape-text or g-markup-printf-escaped. (Note: The functions g-markup-escape-text and g-markup-printf-escaped are not implemented in the Lisp binding.)

Markup strings are just a convenient way to set the PangoAttrList on a label; gtk-label-set-attributes may be a simpler way to set attributes in some cases. Be careful

though; PangoAttrList tends to cause internationalization problems, unless you're applying attributes to the entire string (i.e. unless you set the range of each attribute to [0, G_MAXINT)). The reason is that specifying the start_index and end_index for a PangoAttribute requires knowledge of the exact string being displayed, so translations will cause problems.

Selectable labels

Labels can be made selectable with gtk-label-set-selectable. Selectable labels allow the user to copy the label contents to the clipboard. Only labels that contain useful-to-copy information such as error messages should be made selectable.

Text layout

A label can contain any number of paragraphs, but will have performance problems if it contains more than a small number. Paragraphs are separated by newlines or other paragraph separators understood by Pango.

The label widget is capable of line wrapping the text automatically. This can be activated using the function gtk-label-set-line-wrap. The first argument is the label and the second argument take T or NIL to switch on or to switch off the line wrapping.

gtk-label-set-justify sets how the lines in a label align with one another. The first argument is the label and the second argument one of the following values of the enumeration type GtkJustification. The possible values are shown in Table 5.1. If you want to set how the label as a whole aligns in its available space, see gtk-misc-set-alignment.

Table 5.1: Values of the type GtkJustification

:left	The text is placed at the left edge of the label.
:right	The text is placed at the right edge of the label.
:center	The text is placed in the center of the label.
:fill	The text is placed is distributed across the label.

The width-chars and max-width-chars properties can be used to control the size allocation of ellipsized or wrapped labels. For ellipsizing labels, if either is specified (and less than the actual text size), it is used as the minimum width, and the actual text size is used as the natural width of the label. For wrapping labels, width-chars is used as the minimum width, if specified, and max-width-chars is used as the natural width. Even if max-width-chars specified, wrapping labels will be rewrapped to use all of the available width.

If you want your label underlined, then you can set a pattern on the label with the function gtk-label-set-pattern. The pattern argument indicates how the underlining should look. It consists of a string of underscore and space characters. An underscore indicates that the corresponding character in the label should be underlined. For example, the string "__ __" would underline the first two characters and eight and ninth characters.

Links

GTK+ supports markup for clickable hyperlinks in addition to regular Pango markup. The markup for links is borrowed from HTML, using the a with href and title attributes. GTK+

renders links similar to the way they appear in web browsers, with colored, underlined text. The title attribute is displayed as a tooltip on the link. An example looks like this:

```
(gtk-label-set-markup label
  "Go to the <a href=\"http://gtk.org/\"> GTK+ Website</a> for more ..."))
```

It is possible to implement custom handling for links and their tooltips with the `activate-link` signal and the `gtk-label-get-current-uri` function.

GtkLabel as GtkBuildable

The `GtkLabel` implementation of the `GtkBuildable` interface supports a custom `<attributes>` element, which supports any number of `<attribute>` elements. The `<attribute>` element has attributes named `name`, `value`, `start` and `end` and allows you to specify `PangoAttribute` values for this label.

Example 5.1: A UI definition fragment specifying Pango attributes

```
<object class="GtkLabel">
  <attributes>
   <attribute name="weight" value="PANGO_WEIGHT_BOLD"/>
   <attribute name="background" value="red" start="5" end="10"/>"
  </attributes>
</object>
```

The `start` and `end` attributes specify the range of characters to which the Pango attribute applies. If `start` and `end` are not specified, the attribute is applied to the whole text. Note that specifying ranges does not make much sense with translatable attributes. Use markup embedded in the translatable content instead.

Examples

Figure 5.1 and Figure 5.2 illustrate the functions for `GtkLabel`. The code for these examples is shown in Example 5.2 and Example 5.3.

Example 5.2: Labels

```
(defun make-heading (text)
  (make-instance 'gtk-label
                 :xalign 0
                 :use-markup t
                 :label (format nil "<b>~A</b>" text)))

(defun example-labels ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "GTK+ 3.4 Example Labels"
                                 :default-width 250
                                 :border-width 12))
          (vbox1 (make-instance 'gtk-box
                                :orientation :vertical
```

```
                          :spacing 6))
    (vbox2 (make-instance 'gtk-box
                          :orientation :vertical
                          :spacing 6))
    (hbox (make-instance 'gtk-box
                         :orientation :horizontal
                         :spacing 12)))
;; Connect a handler for the signal "destroy" to window.
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
;; Create a Normal Label
(gtk-box-pack-start vbox1
                    (make-heading "Normal Label:")
                    :expand nil)
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :label "This is a Normal Label")
                    :expand nil)
;; Create a Multi-line Label
(gtk-box-pack-start vbox1
                    (make-heading "Multi-line Label:")
                    :expand nil)
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :label
                                   (format nil
                                         "This is a Multi-line label~%~
                                          Second line~%~
                                          Third line"))
                    :expand nil)
;; Create a Left Justified Label
(gtk-box-pack-start vbox1
                    (make-heading "Left Justified Label:")
                    :expand nil)
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :justify :left
                                   :label
                                   (format nil
                                           "This is a Left Justified~%~
                                            Multi-line label~%~
                                            Third line"))
                    :expand nil)
;; Create a Right Justified Label
(gtk-box-pack-start vbox1
```

```
                    (make-heading "Right Justified Label:")
                    :expand nil)
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :justify :right
                                   :label
                                   (format nil
                                           "This is a Right Justified~%~
                                            Multi-line label~%~
                                            Third line"))
                    :expand nil)
;; Create a Line wrapped label
(gtk-box-pack-start vbox2
                    (make-heading "Line Wrapped Label:")
                    :expand nil)
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :wrap t
                                   :label
                                   (format nil
                                    "This is an example of a ~
                                     line-wrapped label.  It should ~
                                     not be taking up the entire ~
                                     width allocated to it, but ~
                                     automatically wraps the words to ~
                                     fit.  The time has come, for all ~
                                     good men, to come to the aid of ~
                                     their party.  The sixth sheik's ~
                                     six sheep's sick.  It supports ~
                                     multiple paragraphs correctly, ~
                                     and correctly adds many extra ~
                                     spaces."))
                    :expand nil)
;; Create a Filled and wrapped label
(gtk-box-pack-start vbox2
                    (make-heading "Filled and Wrapped Label:")
                    :expand nil)
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :wrap t
                                   :justify :fill
                                   :label
                                   (format nil
                                    "This is an example of a ~
                                     line-wrapped, filled label.  It ~
                                     should be taking up the entire ~
                                     width allocated to it.  Here is ~
```

```
                                    a sentence to prove my point.  ~
                                    Here is another sentence.  Here ~
                                    comes the sun, do de do de do.  ~
                                    This  is a new paragraph.  This ~
                                    is  another newer, longer, ~
                                    better  paragraph.  It is coming ~
                                    to an end, unfortunately."))
                    :expand nil)
;; Create an underlined label
(gtk-box-pack-start vbox2
                    (make-heading "Underlined Label:")
                    :expand nil)
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :justify :left
                                   :use-underline t
                                   :pattern
    "_____ - _____ - ______     __ _______ ___"
                                   :label
                                   (format nil
                                    "This label is underlined!~%~
                                     This one is underlined in quite ~
                                     a  funky fashion"))
                    :expand nil)
;; Put the boxes into the window and show the window
(gtk-box-pack-start hbox vbox1 :expand nil)
(gtk-box-pack-start hbox (gtk-separator-new :vertical))
(gtk-box-pack-start hbox vbox2 :expand nil)
(gtk-container-add window hbox)
(gtk-widget-show-all window))))
```

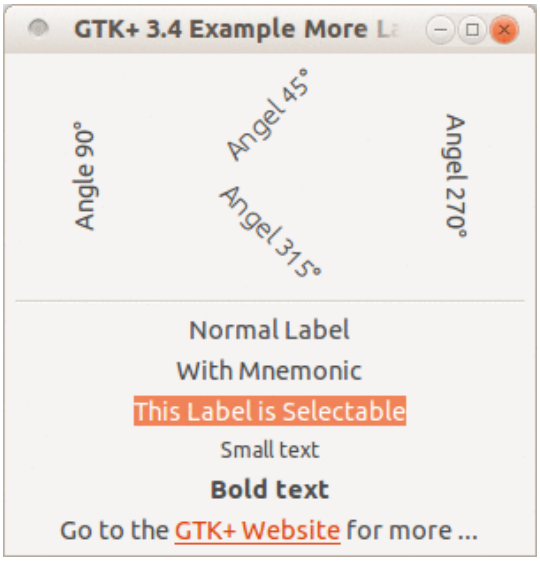

Figure 5.2: More Labels

Example 5.3: More Labels

```
(defun example-more-labels ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "GTK+ 3.4 Example More Labels"
                                 :default-width 300
                                 :border-width 6))
          (vbox1 (make-instance 'gtk-box
                                :orientation :vertical
                                :homogeneous nil
                                :spacing 6))
          (vbox2 (make-instance 'gtk-box
                                :orientation :vertical
                                :homogeneous nil
                                :spacing 6))
          (hbox (make-instance 'gtk-box
                               :orientation :horizontal
                               :homogeneous nil
                               :spacing 6)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-box-pack-start hbox
                          (make-instance 'gtk-label
                                         :label "Angle 90"
```

```
                                    :angle 90))
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :label "Angel 45"
                                   :angle 45))
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :label "Angel 315"
                                   :angle 315))
(gtk-box-pack-start hbox vbox1)
(gtk-box-pack-start hbox
                    (make-instance 'gtk-label
                                   :label "Angel 270"
                                   :angle 270))
(gtk-box-pack-start vbox2 hbox)
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-hseparator))
(gtk-box-pack-start vbox2
                    (gtk-label-new "Normal Label"))
(gtk-box-pack-start vbox2
                    (gtk-label-new-with-mnemonic "With _Mnemonic"))
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :label "This Label is Selectable"
                                   :selectable t))
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :label
                                   "<small>Small text</small>"
                                    :use-markup t))
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :label
                                   "<b>Bold text</b>"
                                    :use-markup t))
(gtk-box-pack-start vbox2
                    (make-instance 'gtk-label
                                   :use-markup t
                                   :label
                                   (format nil
                                   "Go to the ~
                                   <a href=\"http://gtk.org/\">~
                                   GTK+ Website</a> for more ...")))
(gtk-container-add window vbox2)
(gtk-widget-show-all window))))
```

5.2 Progress Bars

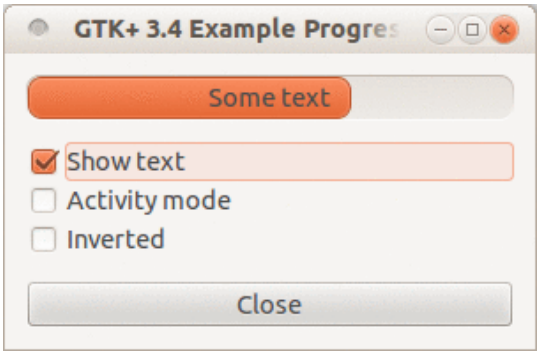

Figure 5.3: Progress Bar

Progress bars are used to show the status of an operation. They are pretty easy to use, as you will see with the code below. But first lets start out with the function `gtk-progress-bar-new` to create a new progress bar. Now that the progress bar has been created we can use it and set the fraction with the function `gtk-progress-bar-set-fraction`, which has two arguments. The first argument is the progress bar you wish to operate on, and the second argument is the amount "completed", meaning the amount the progress bar has been filled from 0-100%. This is passed to the function as a real number ranging from 0 to 1.

A progress bar may be set to one of a number of orientations using the function `gtk-progress-bar-set-orientation`. The second argument is the orientation and can take one of the values

The orientation argument may take one of the values of Table 5.2 to indicate the direction in which the progress bar moves.

Table 5.2: `GtkProgressBarOrientation` is an enumeration representing possible orientations and growth directions for the visible progress bar.

`:left-to-right`
: A horizontal progress bar growing from left to right.

`:right-to-left`
: A horizontal progress bar growing from right to left.

`:bottom-to-top`
: A vertical progress bar growing from bottom to top.

`:top-to-bottom`
: A vertical progress bar growing from top to bottom.

As well as indicating the amount of progress that has occurred, the progress bar may be set to just indicate that there is some activity. This can be useful in situations where progress cannot be measured against a value range. The function `gtk-progress-bar-pulse` indicates that some progress has been made. The step size of the activity indicator is set using the function `gtk-progress-bar-set-pulse-step`.

When not in activity mode, the progress bar can also display a configurable text string within its trough, using the function `gtk-progress-bar-set-text`.

You can turn off the display of the string by calling `gtk-progess-bar-set-text` again with `NIL` as second argument.

The current text setting of a progressbar can be retrieved with the function `gtk-progress-bar-get-text`.

Progress Bars are usually used with timeouts or other such functions (see section on Timeouts, I/O and Idle Functions) to give the illusion of multitasking. All will employ the `gtk-progress-bar-set-fraction` or `gtk-progress-bar-pulse` functions in the same manner.

Example 5.4 shows an example of the progress bar, updated using timeouts. This code also shows you how to reset the Progress Bar. The output of this example is in Figure 5.3.

Example 5.4: Progress Bar

```
(defstruct pbar-data
  pbar
  timer
  mode)

(defun progress-bar-timeout (pdata)
  (if (pbar-data-mode pdata)
      (gtk-progress-bar-pulse (pbar-data-pbar pdata))
      (let ((val (+ (gtk-progress-bar-get-fraction (pbar-data-pbar pdata))
                    0.01)))
        (when (> val 1.0) (setq val 0.0))
        (gtk-progress-bar-set-fraction (pbar-data-pbar pdata) val)))
  t)

(defun example-progress-bar ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "GTK+ 3.4 Example Progress Bar"
                                 :default-width 300))
          (pdata (make-pbar-data :pbar (make-instance 'gtk-progress-bar)))
          (vbox (make-instance 'gtk-box
                               :orientation :vertical
                               :border-width 12
                               :spacing 12))
          (align (gtk-alignment-new 0.1 0.9 1.0 0.0))
          (table (gtk-table-new 2 3 t)))
      (setf (pbar-data-timer pdata)
            (g-timeout-add 100
                           (lambda ()
                             (progress-bar-timeout pdata))))
      (g-signal-connect window "destroy"
```

```
                  (lambda (widget)
                    (declare (ignore widget))
                    (g-source-remove (pbar-data-timer pdata))
                    (setf (pbar-data-timer pdata) 0)
                    (gtk-main-quit)))
(gtk-box-pack-start vbox align)
(gtk-container-add align (pbar-data-pbar pdata))
(gtk-box-pack-start vbox table)
(let ((check (gtk-check-button-new-with-mnemonic "_Show text")))
  (g-signal-connect check "clicked"
     (lambda (widget)
       (declare (ignore widget))
       (let ((text (gtk-progress-bar-get-text (pbar-data-pbar pdata))))
         (if (or (null text) (zerop (length text)))
             (gtk-progress-bar-set-text (pbar-data-pbar pdata)
                                        "Some text")
             (gtk-progress-bar-set-text (pbar-data-pbar pdata)
                                        ""))
         (gtk-progress-bar-set-show-text
                              (pbar-data-pbar pdata)
                              (gtk-toggle-button-get-active check)))))
  (gtk-table-attach table check 0 1 0 1))
(let ((check (gtk-check-button-new-with-label "Activity mode")))
  (g-signal-connect check "clicked"
     (lambda (widget)
       (declare (ignore widget))
       (setf (pbar-data-mode pdata)
             (not (pbar-data-mode pdata)))
       (if (pbar-data-mode pdata)
           (gtk-progress-bar-pulse (pbar-data-pbar pdata))
           (gtk-progress-bar-set-fraction (pbar-data-pbar pdata)
                                          0.0))))
  (gtk-table-attach table check 0 1 1 2))
(let ((check (gtk-check-button-new-with-label "Inverted")))
  (g-signal-connect check "clicked"
     (lambda (widget)
       (declare (ignore widget))
       (gtk-progress-bar-set-inverted
                                (pbar-data-pbar pdata)
                                (gtk-toggle-button-get-active check))))
  (gtk-table-attach table check 0 1 2 3))
(let ((button (gtk-button-new-with-label "Close")))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-widget-destroy window)))
  (gtk-box-pack-start vbox button))
```

```
    (gtk-container-add window vbox)
    (gtk-widget-show-all window))))
```

5.3 Statusbars

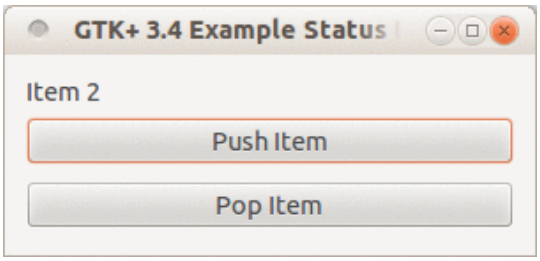

Figure 5.4: Statusbar

Statusbars are simple widgets used to display a text message. They keep a stack of the messages pushed onto them, so that popping the current message will re-display the previous text message.

In order to allow different parts of an application to use the same statusbar to display messages, the statusbar widget issues context Identifiers which are used to identify different "users". The message on top of the stack is the one displayed, no matter what context it is in. Messages are stacked in last-in-first-out order, not context identifier order.

A statusbar is created with a call to `gtk-statusbar-new`. A new context Identifier is requested using a call to the function `gtk-statusbar-get-context-id` with a short textual description of the context as the second argument.

There are three functions that can operate on statusbars: `gtk-statusbar-push`, `gtk-statusbar-pop`, and `gtk-statusbar-remove`. The first function, `gtk-statusbar-push`, is used to add a new message to the statusbar. It returns a message identifier, which can be passed later to the function `gtk-statusbar-remove` to remove the message with the given message and context identifiers from the stack of the statusbar. The function `gtk-statusbar-pop` removes the message highest in the stack with the given context identifier.

In addition to messages, statusbars may also display a resize grip, which can be dragged with the mouse to resize the toplevel window containing the statusbar, similar to dragging the window frame. The functions `gtk-statusbar-set-has-resize-grip` and `gtk-statusbar-get-has-resize-grip` control the display of the resize grip.

Example 5.5 creates a statusbar and two buttons, one for pushing items onto the statusbar, and one for popping the last item back off.

Example 5.5: Statusbar

```
(defun example-statusbar ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :title "Example Status Bar"
                                  :default-width 300
                                  :border-width 12))
```

```
      (vbox (make-instance 'gtk-vbox
                           :homogeneous nil
                           :spacing 3))
      (statusbar (make-instance 'gtk-statusbar))
      (id (gtk-statusbar-get-context-id statusbar "Example Status Bar"))
      (count 0))
  (g-signal-connect window "destroy"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-main-quit)))
  (gtk-box-pack-start vbox statusbar)
  (let ((button (gtk-button-new-with-label "Push Item")))
    (g-signal-connect button "clicked"
       (lambda (widget)
         (declare (ignore widget))
         (setq count (+ 1 count))
         (gtk-statusbar-push statusbar id (format nil "Item ~A" count))))
    (gtk-box-pack-start vbox button :expand t :fill t :padding 3))
  (let ((button (gtk-button-new-with-label "Pop Item")))
    (g-signal-connect button "clicked"
       (lambda (widget)
         (declare (ignore widget))
         (gtk-statusbar-pop statusbar id)))
    (gtk-box-pack-start vbox button :expand t :fill t :padding 3))
  (gtk-container-add window vbox)
  (gtk-widget-show window))))
```

5.4 Info Bars

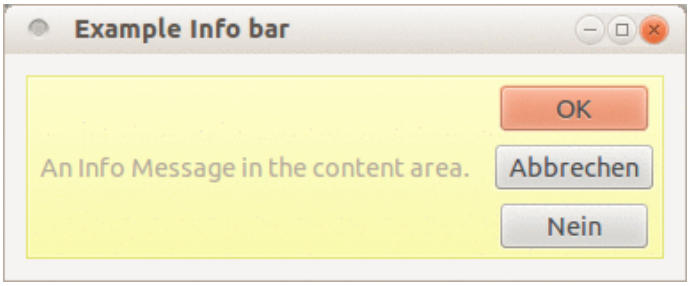

Figure 5.5: Info Bar

`GtkInfoBar` is a widget that can be used to show messages to the user without showing a dialog. It is often temporarily shown at the top or bottom of a document. In contrast to `GtkDialog`, which has a horizontal action area at the bottom, `GtkInfoBar` has a vertical action area at the side.

The API of `GtkInfoBar` is very similar to `GtkDialog`, allowing you to add buttons to the action area with `gtk-info-bar-add-button` or `gtk-info-bar-new-with-buttons`. The sensitivity of action widgets can be controlled with `gtk-info-bar-set-response-`

sensitive. To add widgets to the main content area of a GtkInfoBar, use gtk-info-bar-get-content-area and add your widgets to the container.

Similar to GtkMessageDialog, the contents of a GtkInfoBar can by classified as error message, warning, informational message, etc, by using gtk-info-bar-set-message-type. GTK+ uses the message type to determine the background color of the message area.

Example 5.6: Info Bar

```
(defun example-info-bar ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :title "Example Info bar"
                                  :border-width 12
                                  :default-width 250))
           (grid (make-instance 'gtk-grid))
           (info-bar (make-instance 'gtk-info-bar
                                    :no-show-all t))
           (message (make-instance 'gtk-label
                                   :label ""))
           (content (gtk-info-bar-get-content-area info-bar)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-widget-show message)
      ;; Add a label to the content area of the info bar
      (gtk-container-add content message)
      ;; Add a button OK to the action area
      (gtk-info-bar-add-button info-bar "gtk-ok" 1)
      ;; Add two more buttons to the action area
      (gtk-info-bar-add-buttons info-bar "gtk-cancel" 2
                                         "gtk-no" 3)
      ;; Connect a handler for the "response" signal of the info bar
      (g-signal-connect info-bar "response"
         (lambda (widget response-id)
           (declare (ignore widget))
           (format t "response-id is ~A~%" response-id)
           (gtk-widget-hide info-bar)))
      (gtk-grid-attach grid info-bar 0 2 1 1)
      ;; Show the info bar
      (gtk-label-set-text message "An Info Message in the content area.")
      (gtk-info-bar-set-message-type info-bar :info)
      (gtk-widget-show info-bar)
      ;; Add the container grid to the window and show all
      (gtk-container-add window grid)
      (gtk-widget-show-all window))))
```

6 Adjustments

6.1 Introduction

GTK+ has various widgets that can be visually adjusted by the user using the mouse or the keyboard, such as the range widgets, described in Chapter 7 [Range Widgets], page 67. There are also a few widgets that display some adjustable portion of a larger area of data, such as the viewport widget `GtkViewport`.

Obviously, an application needs to be able to react to changes the user makes in range widgets. One way to do this would be to have each widget emit its own type of signal when its adjustment changes, and either pass the new value to the signal handler, or require it to look inside the data structure of the widget in order to ascertain the value. But you may also want to connect the adjustments of several widgets together, so that adjusting one adjusts the others. The most obvious example of this is connecting a scrollbar to a panning viewport or a scrolling text area. If each widget has its own way of setting or getting the adjustment value, then the programmer may have to write their own signal handlers to translate between the "output" of one widget's signal and the "input" of another's adjustment setting function.

GTK+ solves this problem using the object `GtkAdjustment`, which is not a widget but a way for widgets to store and pass adjustment information in an abstract and flexible form. The most obvious use of `GtkAdjustment` is to store the configuration parameters and values of range widgets, such as scrollbars and scale controls. However, since `GtkAdjustment` is derived from `GObject`, adjustments have some special powers beyond those of normal data structures. Most importantly, they can emit signals, just like widgets, and these signals can be used not only to allow a program to react to user input on adjustable widgets, but also to propagate adjustment values transparently between adjustable widgets.

You will see how adjustments fit in when you see the other widgets that incorporate them: Progress Bars, Viewports, Scrolled Windows, and others.

6.2 Creating an Adjustment

Many of the widgets which use adjustment objects do so automatically, but some cases will be shown in later examples where you may need to create an adjustment yourself. An adjustment can be created with the function `gtk-adjustment-new` which has the arguments `value`, `lower`, `upper`, `step-increment`, `page-increment`, and `page-size`.

The argument `value` is the initial value you want to give to the adjustment, usually corresponding to the topmost or leftmost position of an adjustable widget. The argument `lower` specifies the lowest value which the adjustment can hold. The argument `step-increment` specifies the "smaller" of the two increments by which the user can change the value, while `page-increment` is the "larger" one. The argument `page-size` usually corresponds somehow to the visible area of a panning widget. The argument `upper` is used to represent the bottom most or right most coordinate in a panning widget's child. Therefore it is not always the largest number that `value` can take, since `page-size` of such widgets is usually non-zero.

6.3 Using Adjustments the Easy Way

The adjustable widgets can be roughly divided into those which use and require specific units for these values and those which treat them as arbitrary numbers. The group which treats the values as arbitrary numbers includes the range widgets (scrollbars and scales, the progress bar widget, and the spin button widget). These widgets are all the widgets which are typically "adjusted" directly by the user with the mouse or keyboard. They will treat the lower and upper values of an adjustment as a range within which the user can manipulate the value of the adjustment. By default, they will only modify the value of an adjustment.

The other group includes the text widget, the viewport widget, the compound list widget, and the scrolled window widget. All of these widgets use pixel values for their adjustments. These are also all widgets which are typically "adjusted" indirectly using scrollbars. While all widgets which use adjustments can either create their own adjustments or use ones you supply, you will generally want to let this particular category of widgets create its own adjustments. Usually, they will eventually override all the values except the value itself in whatever adjustments you give them, but the results are, in general, undefined (meaning, you'll have to read the source code to find out, and it may be different from widget to widget).

Now, you are probably thinking, since text widgets and viewports insist on setting everything except the value of their adjustments, while scrollbars will only touch the value of the adjustment, if you share an adjustment object between a scrollbar and a text widget, manipulating the scrollbar will automagically adjust the viewport widget? Of course it will! Just like this:

```
(let (;; A viewport creates its own adjustments
      (viewport (gtk-viewport-new))
      ;; use the adjustment from the viewport for the scrollbar
      (vscrollbar (make-instance 'gtk-scrollbar
                                 :orientation :vertical
                                 :vadjustment
                                 (gtk-scrollable-get-vadjustment viewport))))
  [...] )
```

6.4 Adjustment Internals

Ok, you say, that's nice, but what if I want to create my own handlers to respond when the user adjusts a range widget or a spin button, and how do I get at the value of the adjustment in these handlers? To answer these questions and more, let's start by taking a look at the Lisp class representing `GtkAdjustment` itself:

```
(define-g-object-class "GtkAdjustment" gtk-adjustment
  (:superclass gtk-object
   :export t
   :interfaces nil
   :type-initializer "gtk_adjustment_get_type")
  ((lower
    gtk-adjustment-lower
    "lower" "gdouble" t t)
```

```
(page-increment
 gtk-adjustment-page-increment
 "page-increment" "gdouble" t t)
(page-size
 gtk-adjustment-page-size
 "page-size" "gdouble" t t)
(step-increment
 gtk-adjustment-step-increment
 "step-increment" "gdouble" t t)
(upper
 gtk-adjustment-upper
 "upper" "gdouble" t t)
(value
 gtk-adjustment-value
 "value" "gdouble" t t)))
```

The slots of the class are `lower`, `page-increment`, `page-size`, `step-increment`, `upper`, and `value`. The slots represent the properties of the C class `GtkAdjustment`. The slots can be accessed with the corresponding Lisp accessor functions. Alternativly, the C accessor functions like `gtk_adjustment_get_value()` and `gtk_adjustment_set_value()` are availabe in the Lisp binding through e. g. `gtk-adjustment-get-value` and `gtk-adjustment-set-value` for the property `value`.

As mentioned earlier, an adjustment object is a subclass of `GObject` just like all the various widgets, and thus it is able to emit signals. This is, of course, why updates happen automagically when you share an adjustment object between a scrollbar and another adjustable widget; all adjustable widgets connect signal handlers to their adjustment's "value-changed" signal, as can your program.

The various widgets that use the adjustment object will emit the signal "value-changed" on an adjustment whenever they change its value. This happens both when user input causes the slider to move on a range widget, as well as when the program explicitly changes the value with `gtk-adjustment-set-value`. So, for example, if you have a scale widget, and you want to change the rotation of a picture whenever its value changes, you would create a callback like this:

```
(defun cb-rotate-picture (adj picture)
  (set-picture-rotation picture (gtk-adjustment-get-value adj))
  ... )
```

and connect it to the scale widget's adjustment like this:

```
(g-signal-connect adj "value-changed"
                  (lambda (widget)
                    (cb-rotate-picture widget picture)))
```

What about when a widget reconfigures the upper or lower fields of its adjustment, such as when a user adds more text to a text widget? In this case, it emits the signal "changed". Range widgets typically connect a handler to this signal, which changes their appearance to reflect the change - for example, the size of the slider in a scrollbar will grow or shrink in inverse proportion to the difference between the lower and upper values of its adjustment.

You probably won't ever need to attach a handler to the signal "changed", unless you are writing a new type of range widget. However, if you change any of the values in an adjustment directly, you should emit this signal on it to reconfigure whatever widgets are using it, like this `(g-signal-emit-by-name adj "changed")`.

7 Range Widgets

7.1 Introduction to Range Widgets

The category of range widgets includes the ubiquitous scrollbar widget and the less common scale widget. Though these two types of widgets are generally used for different purposes, they are quite similar in function and implementation. All range widgets share a set of common graphic elements, each of which has its own X window and receives events. They all contain a "trough" and a "slider" (what is sometimes called a "thumbwheel" in other GUI environments). Dragging the slider with the pointer moves it back and forth within the trough, while 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.

As mentioned in Chapter 6 [Adjustments], page 63, all range widgets are associated with an adjustment object, from which they calculate the length of the slider and its position within the trough. When the user manipulates the slider, the range widget will change the value of the adjustment.

7.2 Scrollbar Widgets

These are your standard, run-of-the-mill scrollbars. These should be used only for scrolling some other widget, such as a list, a text box, or a viewport (and it is generally easier to use the scrolled window widget in most cases). For other purposes, you should use scale widgets, as they are friendlier and more featureful.

The `GtkScrollbar` widget is a horizontal or vertical scrollbar, depending on the value of the `orientation` property. A scrollbar can be created with the function `gtk-scrollbar-new`, which takes two arguments. The first argument gives the direction `:horizontal` or `:vertical` and the second a `GtkAdjustment`. The second argument can be `NIL`. In this case a `GtkAdjustment` is created.

The position of the thumb in a scrollbar is controlled by the scroll adjustments. See `GtkAdjustment` for the fields in an adjustment - for `GtkScrollbar`, the property `value` of `GtkAdjustment` represents the position of the scrollbar, which must be between the `lower` and `(upper - page-size)` properties. The property `page-size` represents the size of the visible scrollable area. The properties `step-increment` and `page-increment` are used when the user asks to step down (using the small stepper arrows) or page down (using for example the PageDown key).

Note

The `GtkHScrollbar` widget for a horizontal scrollbar and the `GtkVScrollbar` widget for a vertical scrollbar are deprecated, but present in GTK+ 3.2. Furthermore, the deprecated functions `gtk-hscrollbar-new` and `gtk-vscrollbar-new` are availabe in GTK+ 3.2, but these functions return a `GtkScrollbar` widget and not a `GtkHScrollbar` or `GtkVScrollbar` widget.

7.3 Scale Widgets

Scale widgets are used to allow the user to visually select and manipulate a value within a specific range. You might want to use a scale widget, for example, to adjust the magnifi-

cation level on a zoomed preview of a picture, or to control the brightness of a color, or to specify the number of minutes of inactivity before a screensaver takes over the screen.

As with scrollbars, the `GtkScale` widget is a horizontal or vertical scale, depending on the value of the `orientation` property. A scale can be created with the function `gtk-scale-new`, which takes two arguments. The first argument gives the direction `:horizontal` or `:vertical` and the second a `GtkAdjustment`.

The adjustment argument can either be an adjustment which has already been created with `gtk-adjustment-new`, or `NIL`, in which case, an anonymous adjustment is created with all of its values set to `0.0` (which is not very useful in this case). In order to avoid confusing yourself, you probably want to create your adjustment with a `page-size` of `0.0` so that its upper value actually corresponds to the highest value the user can select. The function `gtk-scale-new-with-range` variant take care of creating a suitable adjustment. The function takes four arguments. The first argument is again the orientation of the scale and the next arguments represent the values for creating an `GtkAdjustment` with intial values for the properties `lower`, `upper`, and `step-increment`. If you are already thoroughly confused, read the section Chapter 6 [Adjustments], page 63 again for an explanation of what exactly adjustments do and how to create and manipulate them.

Functions and Signals

Scale widgets can display their current value as a number beside the trough. The default behaviour is to show the value, but you can change this with this with the function `gtk-scale-set-draw-value`, which takes as the first argument a widget of type `GtkScale` and as the second argument `draw-value`, which is either `T` or `NIL`, with predictable consequences for either one.

The value displayed by a scale widget is rounded to one decimal point by default, as is the value field in its adjustment. You can change this with the function `gtk-scale-set-digits`. The first argument is a widget of type `GtkScale` and the second argument `digits`, where `digits` is the number of decimal places you want. You can set digits to anything you like, but no more than 13 decimal places will actually be drawn on screen.

Finally, the value can be drawn in different positions relative to the trough with the function `gtk-scale-set-value-pos`. The first argument is again a scale widget. The second argument `pos` is of the enumeration type `GtkPositionType`. Possible values are shown in Table 7.1.

If you position the value on the "side" of the trough (e.g., on the top or bottom of a horizontal scale widget), then it will follow the slider up and down the trough.

Table 7.1: The enumeration type `GtkPositionType` describes which edge of a widget a certain feature is positioned at, e.g. the tabs of a `GtkNotebook`, the handle of a `GtkHandleBox` or the label of a `GtkScale`.

`:left`	The feature is at the left edge.
`:right`	The feature is at the right edge.
`:top`	The feature is at the top edge.
`:bottom`	The feature is at the bottom edge.

7.4 Common Range Functions

The range widget class is fairly complicated internally, but, like all the base class widgets, most of its complexity is only interesting if you want to hack on it. Also, almost all of the functions and signals it defines are only really used in writing derived widgets. There are, however, a few useful functions that will work on all range widgets.

Getting and Setting Adjustments

Getting and setting the adjustment for a range widget "on the fly" is done with the functions `gtk-range-get-adjustment` and `gtk-range-set-adjustment`. The function `gtk-range-get-adjustment` returns the adjustment to which range is connected.

`gtk-range-set-adjustment` does absolutely nothing if you pass it the adjustment that range is already using, regardless of whether you changed any of its fields or not. If you pass it a new adjustment, it will unreference the old one if it exists (possibly destroying it), connect the appropriate signals to the new one, and call the function `gtk-range-adjustment-changed`, which will recalculate the size or position of the slider and redraw if necessary. As mentioned in the section on adjustments, if you wish to reuse the same adjustment, when you modify its values directly, you should emit the "changed" signal on it.

Key and Mouse bindings

All of the GTK+ range widgets react to mouse clicks in more or less the same way. Clicking button-1 in the trough will cause its adjustment's `page-increment` to be added or subtracted from its value, and the slider to be moved accordingly. Clicking mouse button-2 in the trough will jump the slider to the point at which the button was clicked. Clicking button-3 in the trough of a range or any button on a scrollbar's arrows will cause its adjustment's value to change by `step-increment` at a time.

Scrollbars are not focusable, thus have no key bindings. The key bindings for the other range widgets (which are, of course, only active when the widget has focus) are do not differentiate between horizontal and vertical range widgets.

All range widgets can be operated with the left, right, up and down arrow keys, as well as with the Page Up and Page Down keys. The arrows move the slider up and down by `step-increment`, while Page Up and Page Down move it by `page-increment`.

The user can also move the slider all the way to one end or the other of the trough using the keyboard. This is done with the Home and End keys.

7.5 Example Range Widgets

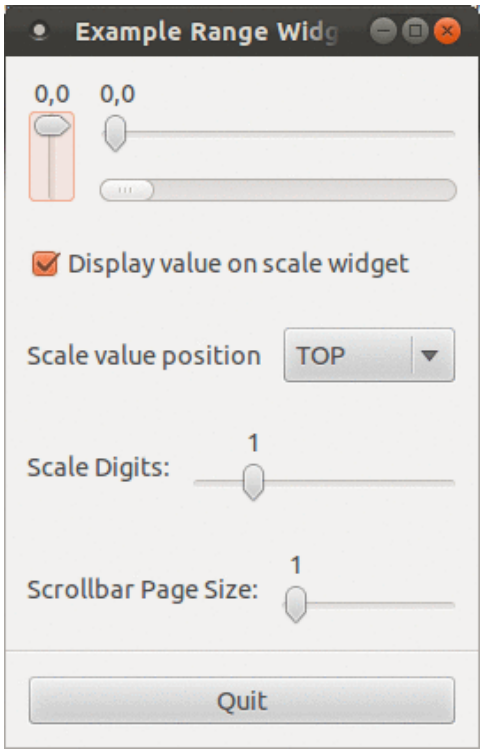

Figure 7.1: Range Widgets

Example 7.1 is a somewhat modified version of the "range controls" test from '`testgtk.c`'. It basically puts up a window with three range widgets all connected to the same adjustment, and a couple of controls for adjusting some of the parameters mentioned above and in the section on adjustments, so you can see how they affect the way these widgets work for the user.

You will notice that the program does not call `g-signal-connect` for the "delete-event", but only for the "destroy" signal. This will still perform the desired function, because an unhandled "delete-event" will result in a "destroy" signal being given to the window.

Example 7.1: Range Widgets

```
(defun example-range-widgets ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :title "Example Range Widgets"))
           (box1 (make-instance 'gtk-box
                                :orientation :vertical
                                :homogeneous nil
                                :spacing 0))
           (box2 (make-instance 'gtk-box
                                :orientation :horizontal
```

```
                          :homogeneous nil
                          :spacing 12
                          :border-width 12))
      (box3 (make-instance 'gtk-box
                          :orientation :vertical
                          :homogeneous nil
                          :spacing 12))
      (adj1 (make-instance 'gtk-adjustment
                          :value 0.0
                          :lower 0.0
                          :upper 101.0
                          :step-increment 0.1
                          :page-increment 1.0
                          :page-size 1.0))
      (vscale (make-instance 'gtk-scale
                            :orientation :vertical
                            :digits 1
                            :value-pos :top
                            :draw-value t
                            :adjustment adj1))
      (hscale (make-instance 'gtk-scale
                             :orientation :horizontal
                             :digits 1
                             :value-pos :top
                             :draw-value t
                             :width-request 200
                             :height-request -1
                             :adjustment adj1))
      (scrollbar (make-instance 'gtk-scrollbar
                               :orientation :horizontal
                               :adjustment adj1)))
;; Connect a handler for the signal "destroy" to the main window.
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
;; Packing of the global widgets hscale, vscale, and scrollbar
(gtk-container-add window box1)
(gtk-box-pack-start box1 box2)
(gtk-box-pack-start box2 vscale)
(gtk-box-pack-start box2 box3)
(gtk-box-pack-start box3 hscale)
(gtk-box-pack-start box3 scrollbar)
;; A check button to control whether the value is displayed or not.
(let ((box (make-instance 'gtk-box
                          :orientation :horizontal
                          :homogeneous nil
```

```
                          :spacing 12
                          :border-width 12))
       (button (make-instance 'gtk-check-button
                          :label "Display value on scale widget"
                          :active t)))
  (g-signal-connect button "toggled"
     (lambda (widget)
       (gtk-scale-set-draw-value
                          hscale
                          (gtk-toggle-button-get-active widget))
       (gtk-scale-set-draw-value
                          vscale
                          (gtk-toggle-button-get-active widget))))
  (gtk-box-pack-start box button)
  (gtk-box-pack-start box1 box))
;; A ComboBox to change the position of the value.
(let ((box (make-instance 'gtk-box
                          :orientation :horizontal
                          :homogeneous nil
                          :spacing 12
                          :border-width 12))
      (combo (make-instance 'gtk-combo-box-text)))
  (gtk-combo-box-text-append-text combo "TOP")
  (gtk-combo-box-text-append-text combo "BOTTOM")
  (gtk-combo-box-text-append-text combo "LEFT")
  (gtk-combo-box-text-append-text combo "RIGHT")
  (gtk-combo-box-set-active combo 0)
  (g-signal-connect combo "changed"
     (lambda (widget)
       (let ((pos (gtk-combo-box-text-get-active-text widget)))
         (format t "type      : ~A~%"
                   (g-type-from-instance (pointer widget)))
         (format t "active is : ~A~%"
                   (gtk-combo-box-get-active widget))
         (setq pos (if pos (intern pos :keyword) :top))
         (gtk-scale-set-value-pos hscale pos)
         (gtk-scale-set-value-pos vscale pos))))
  (gtk-box-pack-start box
                      (make-instance 'gtk-label
                                     :label "Scale value position")
                      :expand nil :fill nil :padding 0)
  (gtk-box-pack-start box combo)
  (gtk-box-pack-start box1 box))
;; Create a scale to change the digits of hscale and vscale.
(let* ((box (make-instance 'gtk-box
                           :orientation :horizontal
                           :homogeneous nil
```

```
                          :spacing 12
                          :border-width 12))
        (adj (make-instance 'gtk-adjustment
                          :value 1.0
                          :lower 0.0
                          :upper 5.0
                          :step-increment 1.0
                          :page-increment 1.0
                          :page-size 0.0))
        (scale (make-instance 'gtk-scale
                            :orientation :horizontal
                            :digits 0
                            :adjustment adj)))
    (g-signal-connect adj "value-changed"
       (lambda (adjustment)
         (setf (gtk-scale-digits hscale)
               (truncate (gtk-adjustment-value adjustment)))
         (setf (gtk-scale-digits vscale)
               (truncate (gtk-adjustment-value adjustment)))))
    (gtk-box-pack-start box
                        (make-instance 'gtk-label
                                       :label "Scale Digits:")
                        :expand nil :fill nil)
    (gtk-box-pack-start box scale)
    (gtk-box-pack-start box1 box))
  ;; Another hscale for adjusting the page size of the scrollbar
  (let* ((box (make-instance 'gtk-box
                          :orientation :horizontal
                          :homogeneous nil
                          :spacing 12
                          :border-width 12))
        (adj (make-instance 'gtk-adjustment
                          :value 1.0
                          :lower 1.0
                          :upper 101.0
                          :step-increment 1.0
                          :page-increment 1.0
                          :page-size 0.0))
        (scale (make-instance 'gtk-scale
                            :orientation :horizontal
                            :digits 0
                            :adjustment adj)))
    (g-signal-connect adj "value-changed"
       (lambda (adjustment)
         (setf (gtk-adjustment-page-size adj1)
               (gtk-adjustment-page-size adjustment))
         (setf (gtk-adjustment-page-increment adj1)
```

```
            (gtk-adjustment-page-increment adjustment))))
  (gtk-box-pack-start box
                      (make-instance 'gtk-label
                                     :label "Scrollbar Page Size:")
                      :expand nil :fill nil)
  (gtk-box-pack-start box scale)
  (gtk-box-pack-start box1 box))
;; Add a separator
(gtk-box-pack-start box1
                    (make-instance 'gtk-separator
                                   :orientation :horizontal)
                    :expand nil :fill t)
;; Create the quit button.
(let ((box (make-instance 'gtk-box
                          :orientation :vertical
                          :homogeneous nil
                          :spacing 12
                          :border-width 12))
      (button (make-instance 'gtk-button :label "Quit")))
  (g-signal-connect button "clicked"
                    (lambda (button)
                      (declare (ignore button))
                      (gtk-widget-destroy window)))
  (gtk-box-pack-start box button)
  (gtk-box-pack-start box1 box :expand nil))
(gtk-widget-show-all window))))
```

8 Layout Widgets

8.1 Alignment widget

Figure 8.1: Alignment Widget

The alignment widget `GtkAlignment` allows to place a widget within its window at a position and size relative to the size of the `GtkAlignment` widget itself. For example, it can be very useful for centering a widget within the window.

The `GtkAlignment` widget has the four properties `xalign`, `yalign`, `xscale`, and `yscale`. The properties are floating point numbers. The align settings are used to place the child widget within the available area. The values range from `0.0` (top or left) to `1.0` (bottom or right). Of course, if the scale settings are both set to `1.0`, the alignment settings have no effect. The scale settings are used to specify how much the child widget should expand to fill the space allocated to the `GtkAlignment`. The values can range from `0.0` (meaning the child doesn't expand at all) to `1.0` (meaning the child expands to fill all of the available space).

The properties are set when creating the `GtkAlignment` widget with the function `gtk-alignment-new`. For an existing `GtkAlignment` widget the properties can be set with the function `gtk-alignment-set`. A child widget can be added to the `GtkAlignment` widget using the function `gtk-container-add`.

In addition, the `GtkAlignment` widget has the properties `top-padding`, `bottom-padding`, `left-padding`, and `right-padding`. These properties control how many space is added to the sides of the widget. The functions `gtk-alignment-set-padding` and `gtk-alignment-get-padding` are used to set or to retrieve the values of the padding properties.

Note:

Note that the desired effect can in most cases be achieved by using the "halign", "valign" and "margin" properties on the child widget, so `GtkAlignment` should not be used in new code.

Example 8.1: Alignment Widget

```
(defun example-alignment ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Alignment"
                                 :border-width 12
                                 :width-request 300
                                 :height-request 300))
          (grid (make-instance 'gtk-grid
                               :column-spacing 12
                               :column-homogeneous t
                               :row-spacing 12
                               :row-homogeneous t)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (let ((frame (make-instance 'gtk-frame
                                  :label "xalign: 0, yalign: 0"))
            (button (make-instance 'gtk-button
                                   :label "Button"))
            (alignment (make-instance 'gtk-alignment
                                      :xalign 0.00
                                      :yalign 0.00
                                      :xscale 0.50
                                      :yscale 0.25)))
        (gtk-alignment-set-padding alignment 6 6 6 6)
        (gtk-container-add alignment button)
        (gtk-container-add frame alignment)
        (gtk-grid-attach grid frame 0 1 1 1))
      (let ((frame (make-instance 'gtk-frame
                                  :label "xalign: 0, yalign: 1"))
            (button (make-instance 'gtk-button
                                   :label "Button"))
            (alignment (make-instance 'gtk-alignment
                                      :xalign 0.00
                                      :yalign 1.00
                                      :xscale 0.50
                                      :yscale 0.25)))
        (gtk-alignment-set-padding alignment 6 6 6 6)
        (gtk-container-add alignment button)
```

```
  (gtk-container-add frame alignment)
  (gtk-grid-attach grid frame 1 1 1 1))
(let ((frame (make-instance 'gtk-frame
                            :label "xalign: 1, yalign: 0"))
      (button (make-instance 'gtk-button
                             :label "Button"))
      (alignment (make-instance 'gtk-alignment
                                :xalign 1.00
                                :yalign 0.00
                                :xscale 0.50
                                :yscale 0.25)))
  (gtk-alignment-set-padding alignment 6 6 6 6)
  (gtk-container-add alignment button)
  (gtk-container-add frame alignment)
  (gtk-grid-attach grid frame 0 2 1 1))
(let ((frame (make-instance 'gtk-frame
                            :label "xalign: 1, yalign: 1"))
      (button (make-instance 'gtk-button
                             :label "Button"))
      (alignment (make-instance 'gtk-alignment
                                :xalign 1.00
                                :yalign 1.00
                                :xscale 0.50
                                :yscale 0.25)))
  (gtk-alignment-set-padding alignment 6 6 6 6)
  (gtk-container-add alignment button)
  (gtk-container-add frame alignment)
  (gtk-grid-attach grid frame 1 2 1 1))
(gtk-container-add window grid)
(gtk-widget-show-all window))))
```

8.2 Fixed Container

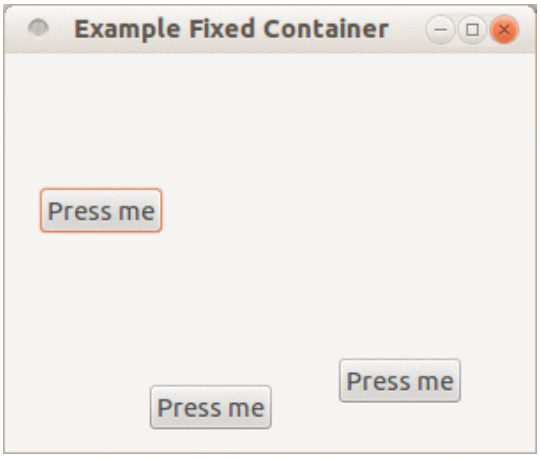

Figure 8.2: Fixed Container

The `GtkFixed` widget is a container widget which allows to place child widgets at a fixed position within the container, relative to the upper left hand corner. The position of the child widgets can be changed dynamically. Only a few functions are associated with the `GtkFixed` widget like `gtk-fixed-new`, `gtk-fixed-put`, and `gtk-fixed-move`.

The function `gtk-fixed-new` creates a new `GtkFixed` widget. The function `gtk-fixed-put` places a widget in the container fixed at the position specified by the arguments `x` and `y`. The function `gtk-fixed-move` allows the specified widget to be moved to a new position.

For most applications, you should not use this container. It keeps you from having to learn about the other GTK+ containers, but it results in broken applications. With `GtkFixed`, the following things will result in truncated text, overlapping widgets, and other display bugs:

- Themes, which may change widget sizes.
- Fonts other than the one you used to write the application will of course change the size of widgets containing text; keep in mind that users may use a larger font because of difficulty reading the default, or they may be using Windows or the framebuffer port of GTK+, where different fonts are available.
- Translation of text into other languages changes its size. Also, display of non-English text will use a different font in many cases.

In addition, the fixed widget can not properly be mirrored in right-to-left languages such as Hebrew and Arabic. i.e. normally GTK+ will flip the interface to put labels to the right of the thing they label, but it can't do that with `GtkFixed`. So your application will not be usable in right-to-left languages.

Finally, fixed positioning makes it kind of annoying to add/remove GUI elements, since you have to reposition all the other elements. This is a long-term maintenance problem for your application.

If you know none of these things are an issue for your application, and prefer the simplicity of `GtkFixed`, by all means use the widget. But you should be aware of the tradeoffs.

The following example illustrates how to use a fixed container. In this example three buttons are put into the fixed widget at random positions. A click on a button moves the button to a new random position. To retrieve the size of the fixed widget the function `gtk-widget-get-allocation` is used, which returns a rectangle of type `GtkAllocation`. The width and the height of the fixed widget are then get with the functions `gdk-rectangle-width` and `gtk-rectangle-height`.

Example 8.2: Fixed Container

```
(defun move-button (button fixed)
  (let* ((allocation (gtk-widget-get-allocation fixed))
         (width (- (gdk-rectangle-width allocation) 20))
         (height (- (gdk-rectangle-height allocation) 10)))
    (gtk-fixed-move fixed button (random width) (random height))))

(defun example-fixed ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Fixed Container"
                                 :default-width 300
                                 :default-height 200
                                 :border-width 12))
          (fixed (make-instance 'gtk-fixed)))
      (g-signal-connect window "destroy"
                        (lambda (window)
                          (declare (ignore window))
                          (gtk-main-quit)))
      (gtk-container-add window fixed)
      (dotimes (i 3)
        (let ((button (gtk-button-new-with-label "Press me")))
          (g-signal-connect button "clicked"
                            (lambda (widget)
                              (move-button widget fixed)))
          (gtk-fixed-put fixed button (random 250) (random 150))))
      (gtk-widget-show window))))
```

8.3 Layout Container

The layout container `GtkLayout` is similar to the fixed container `GtkFixed` except that it implements an infinite (where infinity is less than 2^32) scrolling area. The X window system has a limitation where windows can be at most 32767 pixels wide or tall. The layout container gets around this limitation by doing some exotic stuff using window and bit gravities, so that you can have smooth scrolling even when you have many child widgets in your scrolling area.

A layout container is created using `gtk-layout-new` which accepts the optional arguments `hadjustment` and `vadjustment` to specify adjustment objects that the layout widget will use for its scrolling.

Widgets can be added and moved in the Layout container using the functions `gtk-layout-put` and `gtk-layout-move`. The size of the layout container can be set using the function `gtk-layout-set-size`.

`GtkLayout` implements the interface `GtkScrollable`. Therefore, for manipulating the horizontal and vertical adjustment widgets the functions `gtk-scrollable-get-hadjustment`, `gtk-scrollable-get-vadjustment`, `gtk-scrollable-set-hadjustment`, and `gtk-scrollable-set-vadjustment` are available.

8.4 Frames

Figure 8.3: Frame Widget

Frames can be used to enclose one or a group of widgets with a box which can optionally be labelled. The position of the label and the style of the box can be altered to suit.

A frame can be created with `(make-instance 'gtk-frame)` or the function `gtk-frame-new`. The label is by default placed in the upper left hand corner of the frame. A value of `NIL` for the label argument will result in no label being displayed. The text of the label can be changed using the function `gtk-frame-set-label`.

The position of the label can be changed using the function `gtk-frame-set-label-align` which has the arguments `xalign` and `yalign` which take values between `0.0` and `1.0`. `xalign` indicates the position of the label along the top horizontal of the frame. `yalign` indicates the vertival position of the label. With a value of `0.5` of `yalign` the label is positioned in the middle of the line of the frame. The default value of `xalign` is `0.0` which places the label at the left hand end of the frame.

The function `gtk-frame-set-shadow-type` alters the style of the box that is used to outline the frame. The second argument is a keyword of the enumeration type `GtkShadowType`. The possible values are listed in Table 11.2.

Example 8.3 illustrates the use of the frame widget. The code of this example is shown in Example 8.3.

Example 8.3: Frame Widget

```
(defun example-frame ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
```

```
                              :type :toplevel
                              :title "Example Frame"
                              :default-width 250
                              :default-height 200
                              :border-width 12))
        (frame (make-instance 'gtk-frame
                              :label "Gtk Frame Widget"
                              :label-xalign 1.0
                              :label-yalign 0.5
                              :shadow-type :etched-in)))
  (g-signal-connect window "destroy"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-main-quit)))
  (gtk-container-add window frame)
  (gtk-widget-show window))))
```

8.5 Aspect Frames

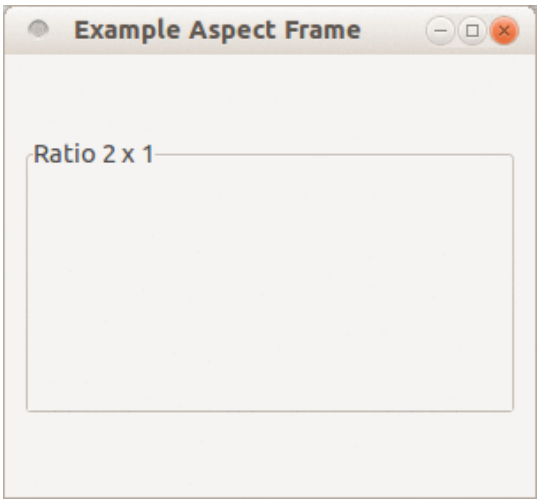

Figure 8.4: Aspect Frame Widget

The aspect frame widget is like a frame widget, except that it also enforces the aspect ratio (that is, the ratio of the width to the height) of the child widget to have a certain value, adding extra space if necessary. This is useful, for instance, if you want to preview a larger image. The size of the preview should vary when the user resizes the window, but the aspect ratio needs to always match the original image.

To create a new aspect frame use `(make-instance 'gtk-aspect-frame)` or the function `gtk-aspect-frame-new`. The arguments `xalign` and `yalign` specify alignment as with alignment widgets. If the property `obey-child` is `T`, the aspect ratio of a child widget will match the aspect ratio of the ideal size it requests. Otherwise, it is given by ratio.

The options of an existing aspect frame can be changed with the function `gtk-aspect-frame-set`.

As an example, the following program uses an aspect frame widget to present a drawing area whose aspect ratio will always be 2:1, no matter how the user resizes the top-level window.

Example 8.4: Aspect Frame Widget

```
(defun example-aspect-frame ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Aspect Frame"
                                 :default-width 300
                                 :default-height 250
                                 :border-width 12))
          (frame (make-instance 'gtk-aspect-frame
                                :label "2 x 1"
                                :xalign 0.5
                                :yalign 0.5
                                :ratio 2
                                :obey-child nil))
          (area (make-instance 'gtk-drawing-area
                               :width-request 200
                               :hight-request 200)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-container-add window frame)
      (gtk-container-add frame area)
      (gtk-widget-show window))))
```

8.6 Paned Window Widgets

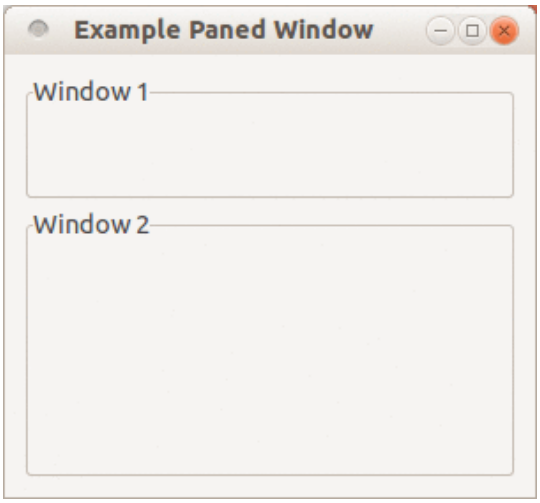

Figure 8.5: Paned Window Widgets

`GtkPaned` has two panes, arranged either horizontally or vertically. The division between the two panes is adjustable by the user by dragging a handle.

A paned window can be created with the function `gtk-paned-new`, which takes as an argument a value of type `GtkOrientation`. With the value `:horizontal` a horizontal paned window is created, and with the value `:vertical` a vertical paned window.

Child widgets are added to the panes of the widget with the functions `gtk-paned-pack1` and `gtk-paned-pack2` or the functions `gtk-paned-add1` and `gtk-paned-add2`. The division between the two children is set by default from the size requests of the children, but it can be adjusted by the user.

A paned widget draws a separator between the two child widgets and a small handle that the user can drag to adjust the division. It does not draw any relief around the children or around the separator. (The space in which the separator is called the gutter.) Often, it is useful to put each child inside a `GtkFrame` with the shadow type set to `:in` so that the gutter appears as a ridge. No separator is drawn if one of the children is missing.

Each child has two options that can be set, `resize` and `shrink`. If `resize` is `T`, then when the `GtkPaned` is resized, that child will expand or shrink along with the paned widget. If `shrink` is `T`, then that child can be made smaller than its requisition by the user. Setting `shrink` to `NIL` allows the application to set a minimum size. If `resize` is `NIL` for both children, then this is treated as if resize is `T` for both children.

The application can set the position of the slider as if it were set by the user, by calling `gtk-paned-set-position`.

Figure 8.5 shows a simple example. The corresponding code is shown in Example 8.5

Example 8.5: Paned Window Widgets

```
(defun example-paned-window ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
```

```
                          :type :toplevel
                          :title "Example Paned Window"
                          :border-width 12))
    (paned (make-instance 'gtk-paned
                         :orientation :vertical))
    (frame1 (make-instance 'gtk-frame :label "Window 1"))
    (frame2 (make-instance 'gtk-frame :label "Window 2")))
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(gtk-widget-set-size-request window 300 250)
(gtk-container-add window paned)
(gtk-paned-add1 paned frame1)
(gtk-paned-add2 paned frame2)
(gtk-widget-show-all window))))
```

8.7 Viewports

The `GtkViewport` widget acts as an adaptor class, implementing scrollability for child widgets that lack their own scrolling capabilities. Use `GtkViewport` to scroll child widgets such as `GtkGrid`, `GtkBox`, and so on.

If a widget has native scrolling abilities, such as `GtkTextView`, `GtkTreeView` or `GtkIconview`, it can be added to a `GtkScrolledWindow` with `gtk-container-add`. If a widget does not, you must first add the widget to a `GtkViewport`, then add the viewport to the scrolled window. The convenience function `gtk-scrolled-window-add-with-viewport` does exactly this, so you can ignore the presence of the viewport.

The `GtkViewport` will start scrolling content only if allocated less than the child widget's minimum size in a given orientation.

A viewport is created with the function `gtk-viewport-new`. The function takes two arguments to specify the horizontal and vertical adjustments that the widget is to use when you create the widget. It will create its own if you pass `NIL` as the value of the arguments.

`GtkViewport` implement the interface `GtkScrollable`. Therefore, the you can get and set the adjustments after the widget has been created using the one of the four functions `gtk-scrollable-get-hadjustment`, `gtk-scrollable-get-vadjustment`, `gtk-scrollable-set-hadjustment`, and `gtk-scrollable-set-vadjustment`.

The only other viewport function is `gtk-viewport-set-shadow-type` used to alter its appearance. The second argument is of type `GtkShadowType`. Possible values are listed in Table 11.2.

8.8 Scrolled Windows

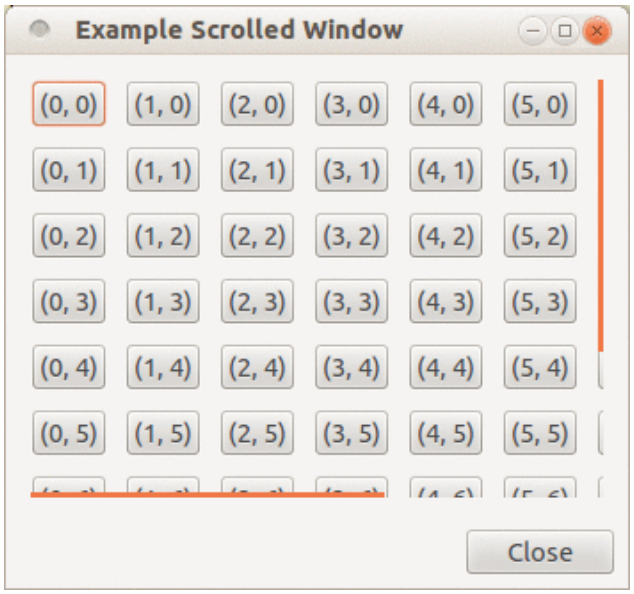

Figure 8.6: Scrolled Window

`GtkScrolledWindow` is a `GtkBin` subclass: it's a container the accepts a single child widget. `GtkScrolledWindow` adds scrollbars to the child widget and optionally draws a beveled frame around the child widget.

The scrolled window can work in two ways. Some widgets have native scrolling support; these widgets implement the `GtkScrollable` interface. Widgets with native scroll support include `GtkTreeView`, `GtkTextView`, and `GtkLayout`.

For widgets that lack native scrolling support, the `GtkViewport` widget acts as an adaptor class, implementing scrollability for child widgets that lack their own scrolling capabilities. Use `GtkViewport` to scroll child widgets such as `GtkGrid`, `GtkBox`, and so on.

If a widget has native scrolling abilities, it can be added to the `GtkScrolledWindow` with `gtk-container-add`. If a widget does not, you must first add the widget to a `GtkViewport`, then add the `GtkViewport` to the scrolled window. The convenience function `gtk-scrolled-window-add-with-viewport` does exactly this, so you can ignore the presence of the viewport.

The position of the scrollbars is controlled by the scroll adjustments. See `GtkAdjustment` for the fields in an adjustment - for `GtkScrollbar`, used by `GtkScrolledWindow`, the "value" field represents the position of the scrollbar, which must be between the "lower" field and "upper - page-size". The "page-size" field represents the size of the visible scrollable area. The "step-increment" and "page-increment" fields are used when the user asks to step down (using the small stepper arrows) or page down (using for example the PageDown key).

If a `GtkScrolledWindow` doesn't behave quite as you would like, or doesn't have exactly the right layout, it's very possible to set up your own scrolling with `GtkScrollbar` and for example a `GtkGrid`.

The function `gtk-scrolled-window-new` is used to create a new scrolled window, where the first argument is the adjustment for the horizontal direction, and the second, the adjustment for the vertical direction. These are almost always set to `NIL`.

The function `gtk-scrolled-window-set-policy` sets the policy to be used with respect to the scrollbars. The first argument is the scrolled window you wish to change. The second sets the policy for the horizontal scrollbar, and the third the policy for the vertical scrollbar.

The policy is of the enumeration type `GtkPolicyType` and may be one of `:automatic` or `:always`. `:automatic` will automatically decide whether you need scrollbars, whereas `:always` will always leave the scrollbars there. All possible values of `GtkPolicyType` are listed in Table 8.1.

Table 8.1: The values of the enumeration type `GtkPolicyType` determines when a scroll bar will be visible.

`:always`	The scrollbar is always visible.
`:automatic`	The scrollbar will appear and disappear as necessary. For example, when all of a GtkCList can not be seen.
`:never`	The scrollbar will never appear.

Example 8.6 is a simple example that packs a table with 100 toggle buttons into a scrolled window. Try playing with resizing the window. You will notice how the scrollbars react. You may also wish to use the `gtk-widget-set-size-request` call to set the default size of the window or other widgets.

Example 8.6: Scrolled Window

```
(defun example-scrolled-window ()
  (within-main-loop
    (let ((window (make-instance 'gtk-dialog
                                 :type :toplevel
                                 :title "Example Scrolled Window"
                                 :border-width 0
                                 :width-request 350
                                 :height-request 300))
          (scrolled (make-instance 'gtk-scrolled-window
                                   :border-width 12
                                   :hscrollbar-policy :automatic
                                   :vscrollbar-policy :always))
          (table (make-instance 'gtk-table
                                :n-rows 10
                                :n-columns 10
                                :row-spacing 10
                                :column-spacing 10
                                :homogeneous nil)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
```

```
                    (declare (ignore widget))
                    (gtk-main-quit)))
(gtk-box-pack-start (gtk-dialog-get-content-area window) scrolled)
(gtk-scrolled-window-add-with-viewport scrolled table)
(dotimes (i 10)
  (dotimes (j 10)
    (gtk-table-attach table
                      (make-instance 'gtk-button
                                     :label
                                     (format nil "(~d, ~d)" i j))
                      i (+ i 1) j (+ j 1))))
(let ((button (make-instance 'gtk-button
                             :label "Close"
                             :can-default t)))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-widget-destroy window)))
  (gtk-box-pack-start (gtk-dialog-get-action-area window) button)
  (gtk-widget-grab-default button))
(gtk-widget-show window))))
```

8.9 Button Boxes

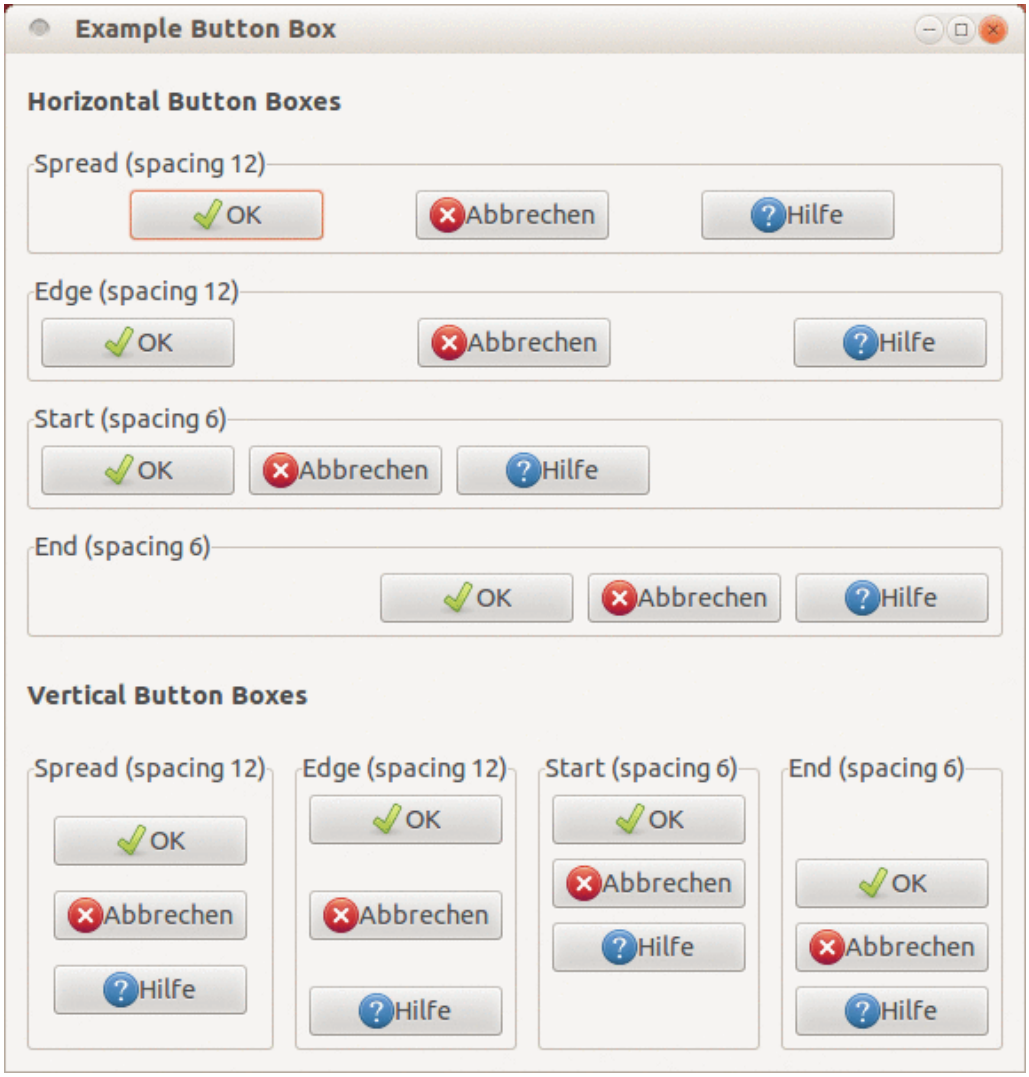

Figure 8.7: Button Boxes

A button box should be used to provide a consistent layout of buttons throughout your application. The layout/spacing can be altered by the programmer, or if desired, by the user to alter the 'feel' of a program to a small degree.

`gtk-button-box-get-layout` and `gtk-button-box-set-layout` retrieve and alter the method used to spread the buttons in a button box across the container, respectively.

The main purpose of `GtkButtonBox` is to make sure the children have all the same size. `GtkButtonBox` gives all children the same size, but it does allow 'outliers' to keep their own larger size. To force all children to be strictly the same size without exceptions, you can set the `"homogeneous"` property to `T`.

To excempt individual children from homogeneous sizing regardless of their 'outlier' status, you can set the `"non-homogeneous"` child property.

You can create a new button box with the function `gtk-button-box-new`, which creates a horizontal or vertical box depending on the argument `orientation` which takes the values `:horizontal` or `:vertical`, respectively.

Buttons are added to a Button Box using the usual function `gtk-container-add`.

Example 8.7 is an example that illustrates all the different layout settings for button boxes.

Example 8.7: Button Boxes

```
(defun create-bbox (orientation title spacing layout)
  (let ((frame (make-instance 'gtk-frame
                              :label title))
        (bbox (make-instance 'gtk-button-box
                             :orientation orientation
                             :border-width 6
                             :layout-style layout
                             :spacing spacing)))
  (gtk-container-add bbox (gtk-button-new-from-stock "gtk-ok"))
  (gtk-container-add bbox (gtk-button-new-from-stock "gtk-cancel"))
  (gtk-container-add bbox (gtk-button-new-from-stock "gtk-help"))
  (gtk-container-add frame bbox)
  frame))

(defun example-button-box ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Button Box"
                                 :border-width 12))
          (vbox1 (make-instance 'gtk-box
                                :orientation :vertical
                                :homogeneous nil
                                :spacing 12))
          (vbox2 (make-instance 'gtk-box
                                :orientation :vertical
                                :homogeneous nil
                                :spacing 12))
          (hbox (make-instance 'gtk-box
                               :orientation :horizontal
                               :homogeneous nil
                               :spacing 12)))
      ;; Set gtk-button-images to T. This allows buttons with text and image.
      (setf (gtk-settings-gtk-button-images (gtk-settings-get-default)) t)
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
```

```
;; Create Horizontal Button Boxes
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :ypad 6
                                   :xalign 0
                                   :use-markup t
                                   :label
                                   "<b>Horizontal Button Boxes</b>")
                    :expand nil
                    :fill nil)
;; Create the first Horizontal Box
(gtk-box-pack-start vbox2
                    (create-bbox :horizontal
                                 "Spread (spacing 12)"
                                 12
                                 :spread))
;; Create the second Horizontal Box
(gtk-box-pack-start vbox2
                    (create-bbox :horizontal
                                 "Edge (spacing 12)"
                                 12
                                 :edge))
;; Create the third Horizontal Box
(gtk-box-pack-start vbox2
                    (create-bbox :horizontal
                                 "Start (spacing 6)"
                                 6
                                 :start))
;; Create the fourth Horizontal Box
(gtk-box-pack-start vbox2
                    (create-bbox :horizontal
                                 "End (spacing 6)"
                                 6
                                 :end))
(gtk-box-pack-start vbox1 vbox2)
;; Create Vertical Button Boxes
(gtk-box-pack-start vbox1
                    (make-instance 'gtk-label
                                   :ypad 12
                                   :xalign 0
                                   :use-markup t
                                   :label
                                   "<b>Vertical Button Boxes</b>")
                    :expand nil
                    :fill nil)
;; Create the first Vertical Box
(gtk-box-pack-start hbox
```

```
                    (create-bbox :vertical
                                 "Spread (spacing 12)"
                                 12
                                 :spread))
;; Create the second Vertical Box
(gtk-box-pack-start hbox
                    (create-bbox :vertical
                                 "Edge (spacing 12)"
                                 12
                                 :edge))
;; Create the third Vertical Box
(gtk-box-pack-start hbox
                    (create-bbox :vertical
                                 "Start (spacing 6)"
                                 6
                                 :start))
;; Create the fourth Vertical Box
(gtk-box-pack-start hbox
                    (create-bbox :vertical
                                 "End (spacing 6)"
                                 6
                                 :end))
(gtk-box-pack-start vbox1 hbox)
(gtk-container-add window vbox1)
(gtk-widget-show-all window))))
```

8.10 Toolbar

A toolbar is created with a call to `gtk-toolbar-new`.

A toolbar can contain instances of a subclass of `GtkToolItem`. To add a `GtkToolItem` to the a toolbar, use `gtk-toolbar-insert`. To remove an item from the toolbar use `gtk-container-remove`. To add a button to the toolbar, add an instance of `GtkToolButton`.

Toolbar items can be visually grouped by adding instances of `GtkSeparatorToolItem` to the toolbar. If the GtkToolbar child property "expand" is TRUE and the property "draw" is set to FALSE, the effect is to force all following items to the end of the toolbar.

Creating a context menu for the toolbar can be done by connecting to the "popup-context-menu" signal.

8.11 Notebook

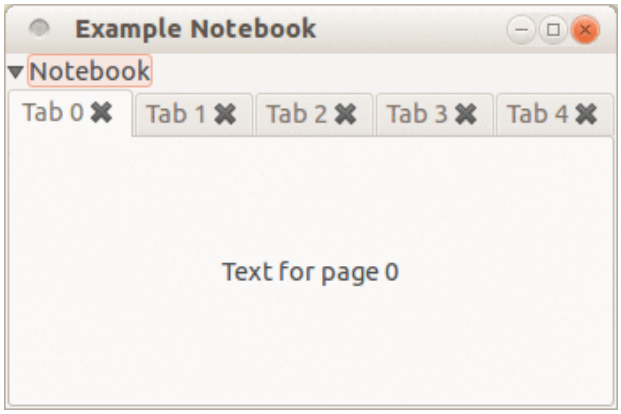

Figure 8.8: Notebook

The `GtkNotebook` widget is a `GtkContainer` whose children are pages that can be switched between using tab labels along one edge.

There are many configuration options for `GtkNotebook`. Among other things, you can choose on which edge the tabs appear (see `gtk-notebook-set-tab-pos`), whether, if there are too many tabs to fit the notebook should be made bigger or scrolling arrows added (see `gtk-notebook-set-scrollable`), and whether there will be a popup menu allowing the users to switch pages (see `gtk-notebook-popup-enable`, `gtk-notebook-popup-disable`).

Example 8.8: Notebook

```
(defun example-notebook ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Notebook"
                                 :type :toplevel
                                 :default-width 250
                                 :default-height 200))
          (expander (make-instance 'gtk-expander
                                   :expanded t
                                   :label "Notebook"))
          (notebook (make-instance 'gtk-notebook
                                   :enable-popup t)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (dotimes (i 5)
        (let ((page (make-instance 'gtk-label
                                   :label
                                   (format nil
                                           "Text for page ~A" i)))
```

```
          (tab-label (make-instance 'gtk-label
                                    :label (format nil "Tab ~A" i)))
          (tab-button (make-instance 'gtk-button
                                     :image
                                     (make-instance 'gtk-image
                                                    :stock
                                                    "gtk-close"
                                                    :icon-size 1)
                                     :relief :none)))
      (g-signal-connect tab-button "clicked"
         (let ((page page))
           (lambda (button)
             (declare (ignore button))
             (format t "Removing page ~A~%" page)
             (gtk-notebook-remove-page notebook page))))
      (let ((tab-hbox (make-instance 'gtk-box
                                     :orientation :horizontal)))
        (gtk-box-pack-start tab-hbox tab-label)
        (gtk-box-pack-start tab-hbox tab-button)
        (gtk-widget-show-all tab-hbox)
        (gtk-notebook-add-page notebook page tab-hbox))))
  (gtk-container-add expander notebook)
  (gtk-container-add window expander)
  (gtk-widget-show-all window))))
```

9 Multiline Text Editor

9.1 Text Widget Overview

GTK+ has an extremely powerful framework for multiline text editing. The primary objects involved in the process are `GtkTextBuffer`, which represents the text being edited, and `GtkTextView`, a widget which can display a `GtkTextBuffer`. Each buffer can be displayed by any number of views.

One of the important things to remember about text in GTK+ is that it's in the UTF-8 encoding. This means that one character can be encoded as multiple bytes. Character counts are usually referred to as offsets, while byte counts are called indexes. If you confuse these two, things will work fine with ASCII, but as soon as your buffer contains multibyte characters, bad things will happen.

Text in a buffer can be marked with tags. A tag is an attribute that can be applied to some range of text. For example, a tag might be called "bold" and make the text inside the tag bold. However, the tag concept is more general than that; tags don't have to affect appearance. They can instead affect the behavior of mouse and key presses, "lock" a range of text so the user can't edit it, or countless other things. A tag is represented by a `GtkTextTag` object. One `GtkTextTag` can be applied to any number of text ranges in any number of buffers.

Each tag is stored in a `GtkTextTagTable`. A tag table defines a set of tags that can be used together. Each buffer has one tag table associated with it; only tags from that tag table can be used with the buffer. A single tag table can be shared between multiple buffers, however.

Tags can have names, which is convenient sometimes (for example, you can name your tag that makes things bold "bold"), but they can also be anonymous (which is convenient if you're creating tags on-the-fly).

Most text manipulation is accomplished with iterators, represented by a `GtkTextIter`. An iterator represents a position between two characters in the text buffer. `GtkTextIter` is a struct designed to be allocated on the stack; it's guaranteed to be copiable by value and never contain any heap-allocated data. Iterators are not valid indefinitely; whenever the buffer is modified in a way that affects the number of characters in the buffer, all outstanding iterators become invalid. (Note that deleting 5 characters and then reinserting 5 still invalidates iterators, though you end up with the same number of characters you pass through a state with a different number).

Because of this, iterators can't be used to preserve positions across buffer modifications. To preserve a position, the `GtkTextMark` object is ideal. You can think of a mark as an invisible cursor or insertion point; it floats in the buffer, saving a position. If the text surrounding the mark is deleted, the mark remains in the position the text once occupied; if text is inserted at the mark, the mark ends up either to the left or to the right of the new text, depending on its gravity. The standard text cursor in left-to-right languages is a mark with right gravity, because it stays to the right of inserted text.

Like tags, marks can be either named or anonymous. There are two marks built-in to `GtkTextBuffer`; these are named "insert" and "selection_bound" and refer to the insertion point and the boundary of the selection which is not the insertion point, respectively. If no

text is selected, these two marks will be in the same position. You can manipulate what is selected and where the cursor appears by moving these marks around. If you want to place the cursor in response to a user action, be sure to use `gtk-text-buffer-place-cursor`, which moves both at once without causing a temporary selection (moving one then the other temporarily selects the range in between the old and new positions).

Text buffers always contain at least one line, but may be empty (that is, buffers can contain zero characters). The last line in the text buffer never ends in a line separator (such as newline); the other lines in the buffer always end in a line separator. Line separators count as characters when computing character counts and character offsets. Note that some Unicode line separators are represented with multiple bytes in UTF-8, and the two-character sequence `"\r\n"` is also considered a line separator.

9.2 Simple Example

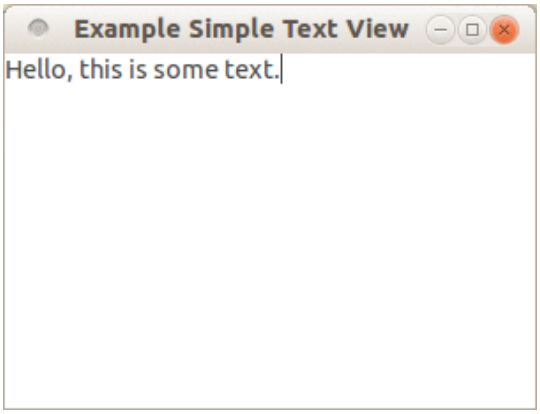

Figure 9.1: Most Simple Text View

The simplest usage of `GtkTextView` might look like in Example 9.1. The output is shown in Figure 9.1.

Example 9.1: Most Simple Text View

```
(defun example-simple-text-view ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :title "Example Simple Text View"
                                  :default-width 300))
           (view (make-instance 'gtk-text-view))
           (buffer (gtk-text-view-get-buffer view)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-text-buffer-set-text buffer "Hello, this is some text.")
      (gtk-container-add window view)
```

```
    (gtk-widget-show-all window))))
```

In many cases it's also convenient to first create the buffer with `gtk-text-buffer-new`, then create a widget for that buffer with `gtk-text-view-new-with-buffer`. Or you can change the buffer the widget displays after the widget is created with `gtk-text-view-set-buffer`.

9.3 Example of Changing Text Attributes

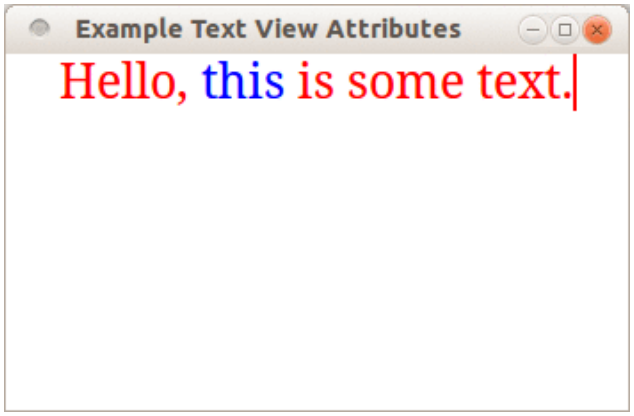

Figure 9.2: Changing Text Attributes of a Text View

There are two ways to affect text attributes in `GtkTextView`. You can change the default attributes for a given `GtkTextView`, and you can apply tags that change the attributes for a region of text. For text features that come from the theme such as font and foreground color use standard `GtkWidget` functions such as `gtk-widget-modify-font` or `gtk-widget-override-text`. For other attributes there are dedicated methods on `GtkTextView` such as `gtk-text-view-set-tabs`.

Example 9.2: Changing Text Attributes of a Text View

```
(defun example-text-view-attributes ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :title "Example Text View Attributes"
                                  :default-width 350))
           (view (make-instance 'gtk-text-view))
           (buffer (gtk-text-view-get-buffer view)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-text-buffer-set-text buffer "Hello, this is some text.")
      ;; Change default font throughout the widget
      (gtk-widget-override-font
                            view
```

```
                        (pango-font-description-from-string "Serif 20"))
;; Change default color throughout the widget
(gtk-widget-override-color view
                           :normal
                           (gdk-rgba-parse "red"))
;; Change left margin throughout the widget
(gtk-text-view-set-left-margin view 30)
;; Use a tag to change the color for just one part of the widget
(let ((tag (make-instance 'gtk-text-tag
                          :name "blue_foreground"
                          :foreground "blue"))
      (start (gtk-text-buffer-get-iter-at-offset buffer 7))
      (end (gtk-text-buffer-get-iter-at-offset buffer 12)))
  ;; Add the tag to the tag table of the buffer
  (gtk-text-tag-table-add (gtk-text-buffer-get-tag-table buffer) tag)
  ;; Apply the tag to a region of the text in the buffer
  (gtk-text-buffer-apply-tag buffer tag start end))
;; Add the view to the window and show all
(gtk-container-add window view)
(gtk-widget-show-all window))))
```

10 Selecting Colors, Files and Fonts

10.1 Selecting Colors

10.1.1 Representing Colors

Colors are represented as a structure of the type `GdkRGBA`, which is defined in the library GDK. The stucture has the properties `red`, `green`, `blue`, and `alpha` to represent rgba colors. It is based on cairo's way to deal with colors and mirrors its behavior. All values are in the range from `0.0d0` to `1.0d0` inclusive. So the color `(0.0d0, 0.0d0, 0.0d0, 0.0d0)` represents transparent black and `(1.0d0, 1.0d0, 1.0d0, 1.0d0)` is opaque white. Other values will be clamped to this range when drawing.

To create a representation of the color red use `(make-gdk-rgba :red 1.0d0)`. Alternatively, the function `gdk-rgba-parse` parses a textual representation of a color, filling in the red, green, blue and alpha fields of the `GdkRGBA` structure. The string can be either one of:

- A standard name (Taken from the X11 rgb.txt file).
- A hex value in the form 'rgb' 'rrggbb' 'rrrgggbbb' or 'rrrrggggbbbb'
- A RGB color in the form 'rgb(r,g,b)' (In this case the color will have full opacity)
- A RGBA color in the form 'rgba(r,g,b,a)'

Where 'r', 'g', 'b' and 'a' are respectively the red, green, blue and alpha color values. In the last two cases, r g and b are either integers in the range 0 to 255 or precentage values in the range 0% to 100%, and a is a floating point value in the range 0 to 1.

Conversely, the function `gdk-rgba-to-string` returns a textual specification of the rgba color in the form rgb (r, g, b) or rgba (r, g, b, a), where 'r', 'g', 'b' and 'a' represent the red, green, blue and alpha values respectively. r, g, and b are represented as integers in the range 0 to 255, and a is represented as floating point value in the range 0 to 1.

These string forms are string forms those supported by the CSS3 colors module, and can be parsed by gdk_rgba_parse().

Note that this string representation may loose some precision, since r, g and b are represented as 8-bit integers. If this is a concern, you should use a different representation.

A simple example is the representation of the color red, which can be created with `(gdk-rgba-parse "Red"` from a string and converted back to a textual with (gdk-rgba-to-string (gdk-rgba-parse "Red")). The result of the last function is `"rgba(255,0,0,0)"`.

Note, that GTK+ knows a second representation of colors as a structure of type `GdkColor`. The implementation is semilar. The widgets for choosing a color know both representations.

10.1.2 Color Button and Color Chooser Dialog

Figure 10.1: Color Selecting Dialog

The `GtkColorButton` is a button which displays the currently selected color and allows to open a color selection dialog to change the color. It is suitable widget for selecting a color in a preference dialog. It implements the `GtkColorChooser` interface.

Example 10.1 shows a simple implementation of a `GtkColorButton`. The example displays a button with the predefined color gray. When clicking the button, a color selection dialog is opened. The dialog is shown in Figure 10.1.

Example 10.1: Color Button

```
(let ((color (gdk-rgba-parse "Gray")))
  (defun example-color-button ()
    (within-main-loop
      (let ((window (make-instance 'gtk-window
                                   :title "Example Color Button"
                                   :border-width 12
                                   :default-width 250
                                   :default-height 200))
            (button (make-instance 'gtk-color-button
                                   :rgba color)))
        (g-signal-connect window "destroy"
                          (lambda (widget)
                            (declare (ignore widget))
                            (gtk-main-quit)))
        (g-signal-connect button "color-set"
           (lambda (widget)
             (let ((rgba (gtk-color-chooser-get-rgba widget)))
               (format t "Selected color is ~A~%"
```

```
                    (gdk-rgba-to-string rgba)))))
        (gtk-container-add window button)
        (gtk-widget-show-all window)))))
```

The `GtkColorChooserDialog` widget is a dialog for choosing a color. It implements the `GtkColorChooser` interface. To provide a dialog in the `GtkColorChooserDialog` the `GtkColorChooserWidget` is used.

By default, the chooser presents a prefined palette of colors, plus a small number of settable custom colors. It is also possible to select a different color with the single-color editor. To enter the single-color editing mode, use the context menu of any color of the palette, or use the '+' button to add a new custom color.

The chooser automatically remembers the last selection, as well as custom colors.

To change the initially selected color, use `gtk-color-chooser-set_rgba`. To get the selected color use `gtk-color-chooser-get-rgba`.

Example 10.2 shows how to replace the default color palette and the default gray palette with the function `gtk-color-chooser-add-palette`.

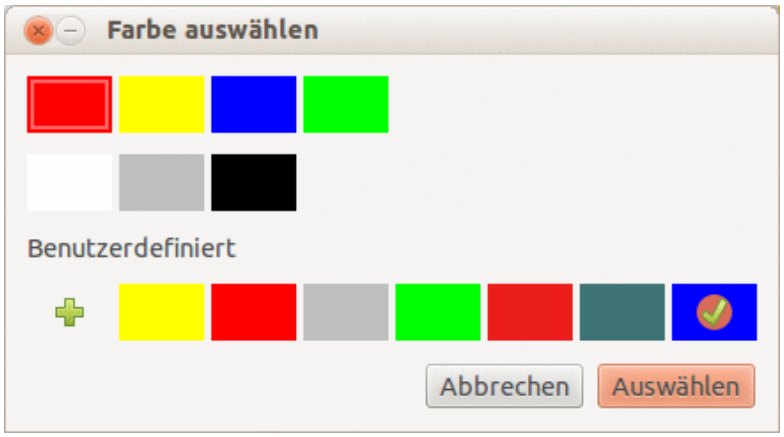

Figure 10.2: Color Selecting Dialog with a custom color and gray palette

Example 10.2: Color Chooser Dialog

```
(let ((color (gdk-rgba-parse "Blue"))
      ;; Color palette with 4 rgba colors
      (palette1 (list (gdk-rgba-parse "Red")
                      (gdk-rgba-parse "Yellow")
                      (gdk-rgba-parse "Blue")
                      (gdk-rgba-parse "Green")))
      ;; Gray palette with 3 rgba grays
      (palette2 (list (gdk-rgba-parse "White")
                      (gdk-rgba-parse "Gray")
                      (gdk-rgba-parse "Black"))))
  (defun drawing-area-event (widget event area)
    (declare (ignore widget))
    (let ((handled nil))
      (when (eql (gdk-event-type event) :button-press)
```

```
      (let ((dialog (make-instance 'gtk-color-chooser-dialog
                                   :color color
                                   :use-alpha nil)))
        (setq handled t)
        ;; Add a custom palette to the dialog
        (gtk-color-chooser-add-palette dialog :vertical 1 palette1)
        ;; Add a second coustom palette to the dialog
        (gtk-color-chooser-add-palette dialog :vertical 1 palette2)
        ;; Run the color chooser dialog
        (let ((response (gtk-dialog-run dialog)))
          (when (eql response :ok)
            (setq color (gtk-color-chooser-get-rgba dialog)))
          ;; Set the color of the area widget
          (gtk-widget-override-background-color area :normal color)
          ;; Destroy the color chooser dialog
          (gtk-widget-destroy dialog))))
    handled))

(defun example-color-chooser-dialog ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Color Chooser Dialog"
                                 :default-width 300))
          (area (make-instance 'gtk-drawing-area)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-widget-override-background-color area :normal color)
      (gtk-widget-set-events area :button-press-mask)
      (g-signal-connect area "event"
                        (lambda (widget event)
                          (drawing-area-event widget event area)))
      (gtk-container-add window area)
      (gtk-widget-show-all window)))))
```

10.2 File Chooser Dialog and File Chooser Button

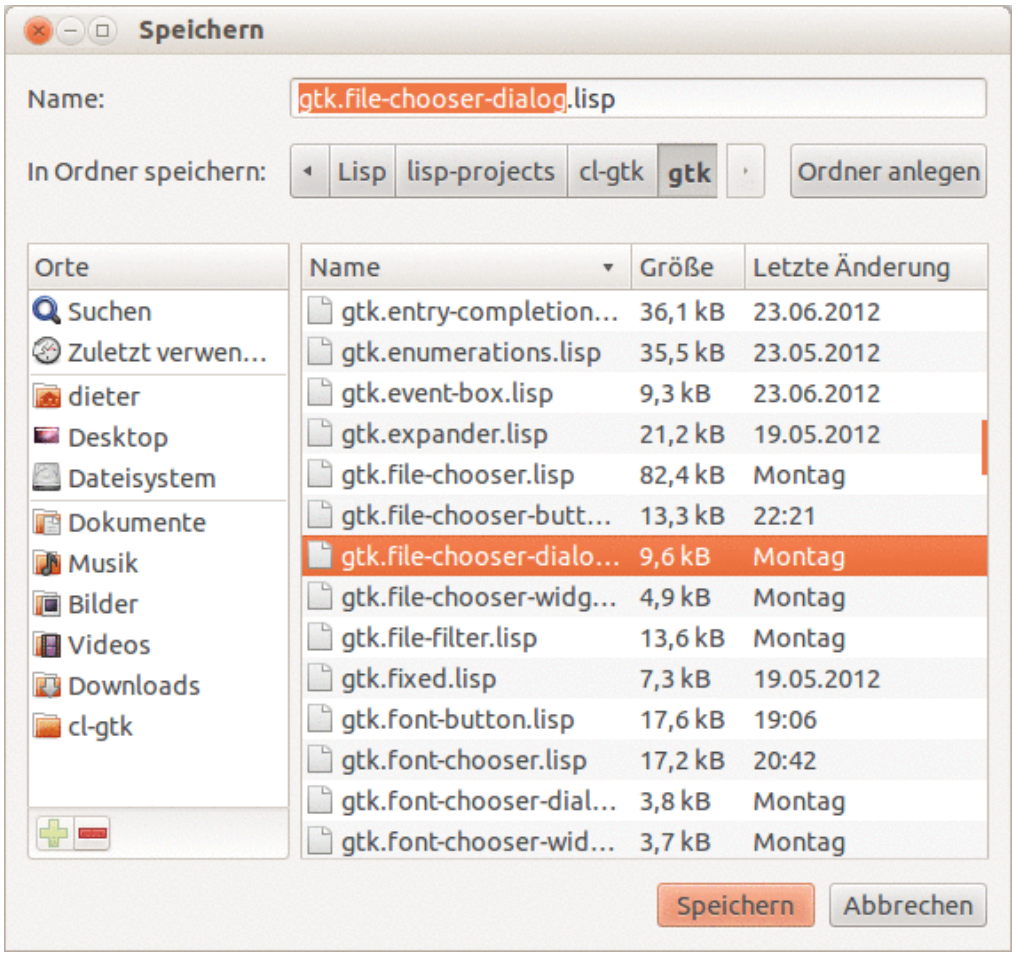

Figure 10.3: File Chooser Dialog

`GtkFileChooser` is an interface that can be implemented by file selection widgets. In GTK+, the main objects that implement this interface are `GtkFileChooserWidget`, `GtkFileChooserDialog`, and `GtkFileChooserButton`. You do not need to write an object that implements the `GtkFileChooser` interface unless you are trying to adapt an existing file selector to expose a standard programming interface.

`GtkFileChooser` allows for shortcuts to various places in the filesystem. In the default implementation these are displayed in the left pane. It may be a bit confusing at first that these shortcuts come from various sources and in various flavours, so lets explain the terminology here:

Bookmarks
: are created by the user, by dragging folders from the right pane to the left pane, or by using the "Add". Bookmarks can be renamed and deleted by the user.

Shortcuts
: can be provided by the application or by the underlying filesystem abstraction (e.g. both the gnome-vfs and the Windows filesystems provide "Desktop" shortcuts). Shortcuts cannot be modified by the user.

Volumes are provided by the underlying filesystem abstraction. They are the "roots" of the filesystem.

File Names and Encodings

When the user is finished selecting files in a `GtkFileChooser`, your program can get the selected names either as filenames or as URIs. For URIs, the normal escaping rules are applied if the URI contains non-ASCII characters. However, filenames are always returned in the character set specified by the `G_FILENAME_ENCODING` environment variable. Please see the Glib documentation for more details about this variable.

Note: This means that while you can pass the result of `gtk-file-chooser-get-filename` to open(2) or fopen(3), you may not be able to directly set it as the text of a `GtkLabel` widget unless you convert it first to UTF-8, which all GTK+ widgets expect. You should use `g-filename-to-utf8` to convert filenames into strings that can be passed to GTK+ widgets.

Adding a Preview Widget

You can add a custom preview widget to a file chooser and then get notification about when the preview needs to be updated. To install a preview widget, use `gtk-file-chooser-set-preview-widget`. Then, connect to the "update-preview" signal to get notified when you need to update the contents of the preview.

Your callback should use `gtk-file-chooser-get-preview-filename` to see what needs previewing. Once you have generated the preview for the corresponding file, you must call `gtk-file-chooser-set-preview-widget-active` with a boolean flag that indicates whether your callback could successfully generate a preview.

Adding Extra Widgets

You can add extra widgets to a file chooser to provide options that are not present in the default design. For example, you can add a toggle button to give the user the option to open a file in read-only mode. You can use `gtk-file-chooser-set-extra-widget` to insert additional widgets in a file chooser.

Note: If you want to set more than one extra widget in the file chooser, you can use a container such as a `GtkVBox` or a `GtkTable` and include your widgets in it. Then, set the container as the whole extra widget.

`GtkFileChooserDialog` is a dialog box suitable for use with "File/Open" or "File/Save as" commands. This widget works by putting a `GtkFileChooserWidget` inside a `GtkDialog`. It exposes the `GtkFileChooserIface` interface, so you can use all of the `GtkFileChooser` functions on the file chooser dialog as well as those for `GtkDialog`.

Note that `GtkFileChooserDialog` does not have any methods of its own. Instead, you should use the functions that work on a `GtkFileChooser`.

Setting up a file chooser dialog

The enumeration `GtkFileChooserAction` describes whether a `GtkFileChooser` is being used to open existing files or to save to a possibly new file. These are the cases in which you may need to use a `GtkFileChooserDialog`.

- To select a file for opening, as for a File/Open command. Use the keyword `:open` for the slot `:action`, when creating a file chooser dialog.

- To save a file for the first time, as for a File/Save command. Use the keyword `:save`, and suggest a name such as `"Untitled"` with `gtk-file-chooser-set-current-name`.
- To save a file under a different name, as for a File/Save As command. Use the keyword `:save`, and set the existing filename with `gtk-file-chooser-set-file-name`.
- To choose a folder instead of a file. Use the keyword `select-folder`.

Response Codes

`GtkFileChooserDialog` inherits from `GtkDialog`, so buttons that go in its action area have response codes such as `:accept` and `:canel`. For example, you could create a file chooser dialog as follows:

```
(let ((dialog (make-instance 'gtk-file-chooser-dialog
                             :title "Speichern"
                             :action :save)))
  (gtk-dialog-add-button dialog "gtk-save" :accept)
  (gtk-dialog-add-button dialog "gtk-cancel" :cancel)
  [...]
```

This will create buttons for `"Cancel"` and `"Save"` that use stock response identifiers from `GtkResponseType`. For most dialog boxes you can use your own custom response codes rather than the ones in `GtkResponseType`, but `GtkFileChooserDialog` assumes that its `"accept"`-type action, e.g. an `"Open"` or `"Save"` button, will have one of the following response codes `:accept`, `:ok`, `:yes`, or `:apply`.

This is because `GtkFileChooserDialog` must intercept responses and switch to folders if appropriate, rather than letting the dialog terminate - the implementation uses these known response codes to know which responses can be blocked if appropriate. To summarize, make sure you use a stock response code when you use `GtkFileChooserDialog` to ensure proper operation.

Example 10.3 shows an example for selecting a file for save. The dialog is shown in Figure 10.3.

Example 10.3: File Chooser Dialog

```
(defun example-file-chooser-dialog ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example File Chooser Dialog"
                                 :type :toplevel
                                 :border-width 12
                                 :default-width 300
                                 :default-height 100))
          (button (make-instance 'gtk-button
                                 :label "Select a file for save ..."
                                 :image
                                 (gtk-image-new-from-stock "gtk-save"
                                                           :button))))
      ;; Handle the signal "destroy" for the window.
      (g-signal-connect window "destroy"
```

```
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-main-quit)))
    ;; Handle the signal "clicked" for the button.
    (g-signal-connect button "clicked"
       (lambda (widget)
         (declare (ignore widget))
         (let ((dialog (make-instance 'gtk-file-chooser-dialog
                                      :title "Speichern"
                                      :action :save)))
           (gtk-dialog-add-button dialog "gtk-save" :accept)
           (gtk-dialog-add-button dialog "gtk-cancel" :cancel)
           (when (eq (gtk-dialog-run dialog) :accept)
             (format t "Saved to file ~A~%"
                     (gtk-file-chooser-filename dialog)))
           (gtk-widget-destroy dialog))))
    (gtk-container-add window button)
    (gtk-widget-show-all window))))
```

The `GtkFileChooserButton` is a widget that lets the user select a file. It implements the `GtkFileChooser` interface. Visually, 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.

Example 10.4 shows an example for a file chooser button to open a file.

Example 10.4: File Chooser Button

```
(defun example-file-chooser-button ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example File Chooser Button"
                                 :type :toplevel
                                 :border-width 12
                                 :default-width 300
                                 :default-height 100))
          (button (make-instance 'gtk-file-chooser-button
                                 :action :open)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (g-signal-connect button "file-set"
                        (lambda (widget)
                          (declare (ignore widget))
                          (format t "File set: ~A~%"
                                  (gtk-file-chooser-filename button))))
      (gtk-container-add window button)
      (gtk-widget-show-all window))))
```

10.3 Font Chooser Button and Font Chooser Dialog

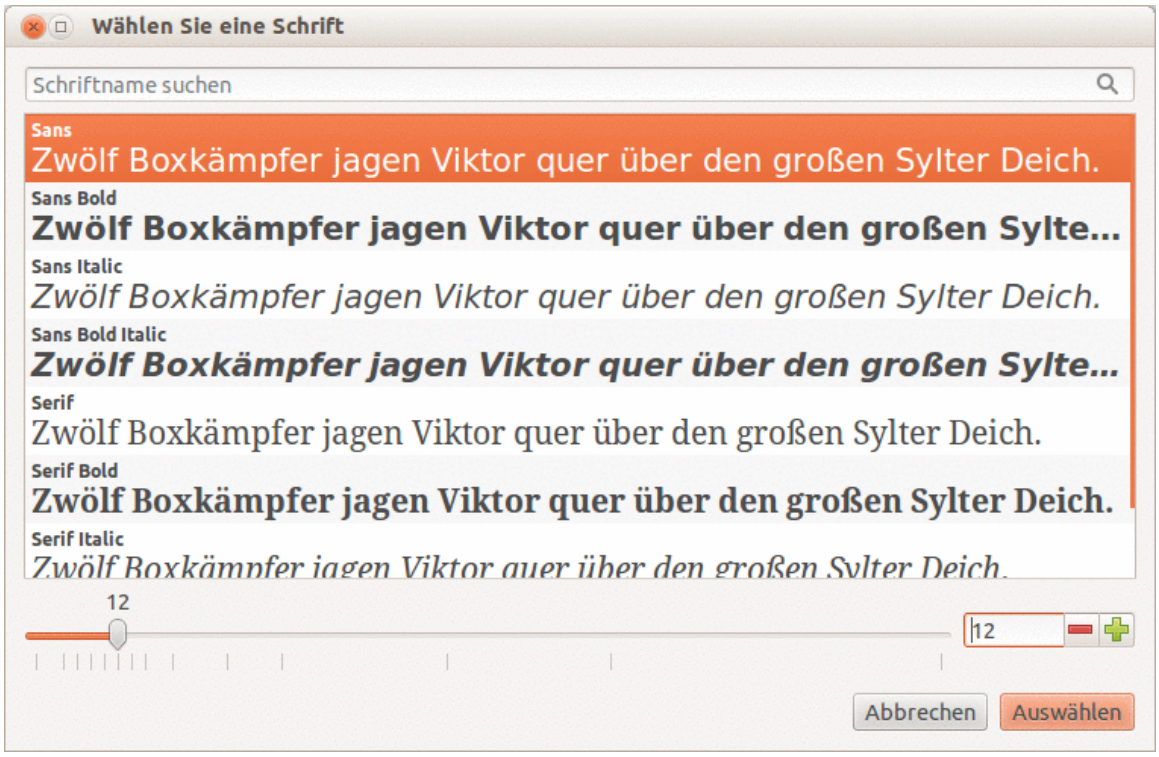

Figure 10.4: Font Chooser Dialog

The `GtkFontChooserWidget` widget lists the available fonts, styles and sizes, allowing the user to select a font. It is used in the `GtkFontChooserDialog` widget to provide a dialog box for selecting fonts.

The `GtkFontChooserDialog` widget is a dialog for selecting a font. It implements the `GtkFontChooser` interface.

To set the font which is initially selected, use `gtk-font-chooser-set-font` or `gtk-font-chooser-set-font-desc`.

To get the selected font use `gtk-font-chooser-get-font` or `gtk-font-chooser-get-font-desc`.

To change the text which is shown in the preview area, use `gtk-font-chooser-set-preview-text()`.

The `GtkFontButton` is a button which displays the currently selected font and allows to open a font chooser dialog to change the font. It is a suitable widget for selecting a font in a preference dialog.

Example 10.5: Font Chooser Dialog with a filter to select fonts

```
(defun font-filter (family face)
  (declare (ignore face))
  (member (pango-font-family-get-name family)
          '("Sans" "Serif")
```

```
          :test #'equal))

(defun example-font-button ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Font Chooser Button"
                                 :type :toplevel
                                 :border-width 12
                                 :default-width 300
                                 :default-height 100))
          (button (make-instance 'gtk-font-button)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      ;; Set a filter function to select fonts for the font chooser
      (gtk-font-chooser-set-filter-func button #'font-filter)
      (g-signal-connect button "font-set"
         (lambda (widget)
           (declare (ignore widget))
           (format t "Font is set:~%")
           (format t "   Font name   : ~A~%"
                   (gtk-font-chooser-get-font button))
           (format t "   Font family : ~A~%"
                   (pango-font-family-get-name
                     (gtk-font-chooser-get-font-family button)))
           (format t "   Font face   : ~A~%"
                   (pango-font-face-get-face-name
                     (gtk-font-chooser-get-font-face button)))
           (format t "   Font size   : ~A~%"
                   (gtk-font-chooser-get-font-size button))))
      (gtk-container-add window button)
      (gtk-widget-show-all window))))
```

11 Miscellaneous Widgets

11.1 Event Box

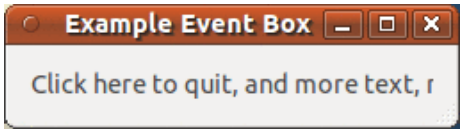

Figure 11.1: Event Box

Some GTK+ widgets do not have associated X windows, so these widgets just draw on their parents. Because of this, they cannot receive events and if they are incorrectly sized, they do not clip so you can get messy overwriting. If you require more from these widgets, the `GtkEventBox` widget is for you.

At first glance, the `GtkEventBox` widget might appear to be totally useless. It draws nothing on the screen and responds to no events. However, it does serve a function - it provides an X window for its child widget. This is important as many GTK+ widgets do not have an associated X window. Not having an X window saves memory and improves performance, but also has some drawbacks. A widget without an X window cannot receive events, and does not perform any clipping on its contents. Although the name `GtkEventBox` emphasizes the event-handling function, the widget can also be used for clipping.

To create a new `GtkEventBox` widget, use the call `(make-instance 'gtk-event-box)` or the function `gtk-event-box-new`. A child widget can then be added to this `GtkEventBox` with the function `gtk-container-add`. With the function `gtk-widget-set-events` the events are set for the event box which can be connected to a signal handler. To create the resources associated with an event box, the function `gtk-widget-realize` has to be called explicitly for the `GtkEventBox` widget.

Example 11.1 demonstrates both uses of a `GtkEventBox` widget - a label is created that is clipped to a small box, and set up so that a mouse-click on the label causes the program to exit. Resizing the window reveals varying amounts of the label.

In addition, Example 11.1 shows how to change the mouse pointer over a window. Every widget has an associated window of type `GdkWindow`, which can be get with the function `gtk-widget-window`. The function `gdk-window-set-cursor` sets a mouse pointer for this `GdkWindow`. A new mouse pointer is created with the function `gdk-cursor-new`. The functions takes one argument, which is a keyword for a predefined mouse pointer. In Example 11.1 the mouse pointer `:hand1` is choosen. This new mouse pointer is than associated to the `GdkWindow` with the function `gdk-window-set-cursor`.

Example 11.1: Event Box

```
(defun example-event-box ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Event Box"
                                 :default-width 250
```

```
                              :border-width 12))
      (eventbox (make-instance 'gtk-event-box))
      (label (make-instance 'gtk-label
                            :width-request 120
                            :height-request 20
                            :label
                            "Click here to quit, and more text, more")))
  (g-signal-connect window "destroy"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-main-quit)))
  (gtk-container-add window eventbox)
  (gtk-container-add eventbox label)
  (gtk-widget-set-events eventbox :button-press-mask)
  (g-signal-connect eventbox "button-press-event"
                    (lambda (widget event)
                      (declare (ignore widget event))
                      (gtk-widget-destroy window)))
  (gtk-widget-realize eventbox)
  (gdk-window-set-cursor (gtk-widget-window eventbox)
                         (gdk-cursor-new :hand1))
  (gtk-widget-show window))))
```

11.2 Arrows

Figure 11.2: Arrows

The arrow widget draws an arrowhead, facing in a number of possible directions and having a number of possible styles. It can be very useful when placed on a button in many applications. Like the label widget, it emits no signals.

There are only two functions for manipulating an arrow widget `gtk-arrow-new` and `gtk-arrow-set`. The first creates a new arrow widget with the indicated type and appearance. The second allows these values to be altered retrospectively. The type of an arrow can be one of the following values of the enumeration type `GtkArrowType` in Table 11.1.

Table 11.1: Values of the type `GtkArrowType` to indicate the direction in which a `GtkArrow` should point.

`:up`	Represents an upward pointing arrow.
`:down`	Represents a downward pointing arrow.
`:left`	Represents a left pointing arrow.

`:right` Represents a right pointing arrow.

`:none` No arrow.

These values obviously indicate the direction in which the arrow will point. The shadow type argument is of the enumeration type `GtkShadowType` and may take one of the the values in Table 11.2.

Table 11.2: Values of the enumeration type `GtkShadowType` used to change the appearance of an outline typically provided by a `GtkFrame`.

`:none` No outline.

`:in` The outline is bevelled inwards.

`:out` The outline is bevelled outwards like a button. This is the default.

`:etched-in`
The outline has a sunken 3d appearance.

`:etched-out`
The outline has a raised 3d appearance.

Example 11.2 shows a brief example to illustrate the use of arrows in buttons. In addition, this example introduces the function `gtk-widget-set-tooltip-text`, which attaches a tooltip to the button widget. The tooltip pops up, when the mouse is over the button.

Example 11.2: Buttons with Arrows

```
(defun create-arrow-button (arrow-type shadow-type)
  (let ((button (make-instance 'gtk-button)))
    (gtk-container-add button
                       (make-instance 'gtk-arrow
                                      :arrow-type arrow-type
                                      :shadow-type shadow-type))
    (gtk-widget-set-tooltip-text button
                                 (format nil
                                         "Arrow of type ~A"
                                         (symbol-name arrow-type)))
    button))

(defun example-arrows ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Arrow Buttons"
                                 :default-width 250
                                 :border-width 12))
          (box (make-instance 'gtk-hbox
                              :homogeneous t
                              :spacing 0
                              :border-width 6)))
```

```
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(gtk-box-pack-start box
                    (create-arrow-button :up :in)
                    :expand nil :fill nil :padding 3)
(gtk-box-pack-start box
                    (create-arrow-button :down :out)
                    :expand nil :fill nil :padding 3)
(gtk-box-pack-start box
                    (create-arrow-button :left :etched-in)
                    :expand nil :fill nil :padding 3)
(gtk-box-pack-start box
                    (create-arrow-button :right :etched-out)
                    :expand nil :fill nil :padding 3)
(gtk-container-add window box)
(gtk-widget-show window))))
```

11.3 Dialogs

11.3.1 General Dialog

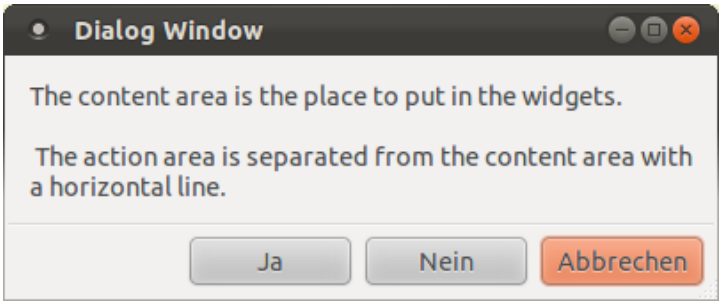

Figure 11.3: General Dialog Window

The dialog widget is just a window with a few things pre-packed into it. The dialog widget is of type `GtkDialog` which is represented by the Lisp class `gtk-dialog`. A dialog widget can be created with the function `gtk-dialog-new` or the call `(make-instance 'gtk-dialog)`. The function `gtk-dialog-new` does not take an argument. The C library knows in addition the function `gtk_dialog_new_with_buttons()` which is not implemented for the Lisp binding.

The dialog widget consists of an content-area which is of type `GtkVBox`. The content area can be filled with the content of a dialog. At the button of the window the dialog widget has an action-area which takes the desired buttons of the dialog. The action area can be separated with a horizontal from the content area setting the property `has-separator` of `GtkDialog`.

The function `gtk-dialog-get-content-area` gets the content area of a dialog. Because the content area is a vertical box of type `GtkVBox` any desired widgets can be added to the

content area with the functions `gtk-box-pack-start` or `gtk-box-pack-end`. To display the content area it is necessary to call the function `gtk-widget-show` explicitly. The function `create-dialog` in Example 11.3 shows how to fill widgets into a dialog widget.

The action area can be filled with the desired buttons for the dialog window. Standard buttons can be added with the function `gtk-dialog-add-button`. The function takes three arguments. The first argument is the dialog window the button is added to. The second argument is a string which is the text of the button or a stock id. The last argument is of the enumeration type `GtkResponseType` and defines the response type of the button. Possible values of `GtkResponseType` are shown in Table 11.3.

Alternatively to the function `gtk-dialog-add-button` buttons can be added with the functions `gtk-box-pack-start` or `box-pack-end` to the action area. The action area is of type `GtkHBox` and can be get with the function `gtk-dialog-get-action-area`.

Table 11.3: Predefined values for use as response ids in `gtk-dialog-add-button`. All predefined values are negative, GTK+ leaves positive values for application-defined response ids.

`:none`	Returned if an action widget has no response id, or if the dialog gets programmatically hidden or destroyed.
`:reject`	Generic response id, not used by GTK+ dialogs
`:accept`	Generic response id, not used by GTK+ dialogs
`:event`	Returned if the dialog is deleted
`:ok`	Returned by OK buttons in GTK+ dialogs
`:cancel`	Returned by Cancel buttons in GTK+ dialogs
`:close`	Returned by Close buttons in GTK+ dialogs
`:yes`	Returned by Yes buttons in GTK+ dialogs
`:no`	Returned by No buttons in GTK+ dialogs
`:apply`	Returned by Apply buttons in GTK+ dialogs
`:help`	Returned by Help buttons in GTK+ dialogs

After creation and configuration of the dialog window the dialog is executed with the function `gtk-dialog-run`. The function takes the dialog window of type `GtkDialog` as the only argument. After closing the dialog window with one of the buttons the response is returned as an integer value of type `GtkResponseType`.

11.3.2 Message Dialog

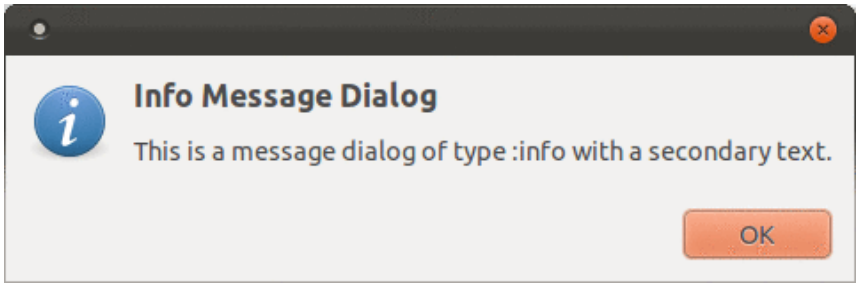

Figure 11.4: Message Dialog

A message dialog `GtkMessageDialog` is a subclass of the more general class `GtkDialog` and gives an easy way to display messages to the user. Figure 11.4 shows an example for an informational message.

A message dialog is created with the call `make-instance 'gtk-message-dialog`. The C functions `gtk_message_dialog_new()` and `gtk_message_dialog_new_with_markup()` have no corresponding Lisp implementation. Various properties control the appearance of a message dialog. The function `create-message-dialog` in Example 11.3 shows the settings of the properties `message-type`, `buttons`, `text`, and `secondary-text`. The type of a message dialog is one of the values of `GtkMessageType`. The possible values are listed in Table 11.4. Predefined buttons of type `GtkButtonsType` for a message dialog are listed in Table 11.5.

Table 11.4: The type of message being displayed in the dialog.

`:info`	Informational message
`:warning`	Nonfatal warning message
`:question`	Question requiring a choice
`:error`	Fatal error message
`:other`	None of the above, doesn't get an icon

Table 11.5: Prebuilt sets of buttons for a message dialog. If none of these choices are appropriate, simply use `:none` then call `gtk-dialog-add_buttons`. Please note that `:ok`, `:yes-no`, and `:canel` are discouraged by the GNOME Human Interface Guidelines.

`:none`	no buttons at all
`:ok`	an OK button
`:close`	a Close button
`:canel`	a Cancel button
`:yes-no`	Yes and No buttons
`:ok-cancel`	OK and Cancel buttons

11.3.3 About Dialog

Figure 11.5: About Dialog

The `GtkAboutDialog` offers a simple way to display information about a program like its logo, name, copyright, website and license. It is also possible to give credits to the authors, documenters, translators and artists who have worked on the program. An about dialog is typically opened when the user selects the About option from the Help menu. All parts of the dialog are optional.

About dialogs often contain links and email addresses. `GtkAboutDialog` displays these as clickable links. By default, it calls `gtk_show_uri()` when a user clicks one. The behavior can be overridden with the `"activate-link"` signal.

To make constructing a `GtkAboutDialog` as convenient as possible, the C library knows the function `gtk_show_about_dialog()` which constructs and shows a dialog and keeps it around so that it can be shown again. This function is not implemented in the Lisp binding.

Note that GTK+ sets a default title of `_("About %s")` on the dialog window, where `%s` is replaced by the name of the application, but in order to ensure proper translation of the title, applications should set the title property explicitly when constructing a `GtkAboutDialog`.

It is possible to show a `GtkAboutDialog` like any other `GtkDialog`, e.g. using `gtk-dialog-run`. In this case, you might need to know that the 'Close' button returns the `:cancel` response id.

Example 11.3: Examples for a general, a message, and an about dialog.

```
(defun license-text ()
  (format nil
          "This program is free software: you can redistribute it and/or ~
          modify it under the terms of the GNU Lesser General Public ~
          License for Lisp as published by the Free Software Foundation, ~
          either version 3 of the License, or (at your option) any later ~
          version and with a preamble to the GNU Lesser General Public ~
          License that clarifies the terms for use with Lisp programs and ~
          is referred as the LLGPL.~%~% ~
          This program is distributed in the hope that it will be useful, ~
          but WITHOUT ANY WARRANTY; without even the implied warranty of ~
```

```
          MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the ~
          GNU Lesser General Public License for more details. ~%~% ~
          You should have received a copy of the GNU Lesser General Public ~
          License along with this program and the preamble to the Gnu ~
          Lesser General Public License.  If not, see ~
          <http://www.gnu.org/licenses/> and ~
          <http://opensource.franz.com/preamble.html>."))

(defun create-dialog ()
  (let ((dialog (make-instance 'gtk-dialog
                               :title "Dialog Window"
                               :has-separator t)))
    ;; Add a border width to the vbox of the content area
    (gtk-container-set-border-width (gtk-dialog-get-content-area dialog) 12)
    ;; Add a label widget with text to the content area
    (let ((vbox (make-instance 'gtk-vbox :border-width 12))
          (label (make-instance 'gtk-label
                                :wrap t
                                :label
                                (format nil
                                        "The content area is the place to ~
                                         put in the widgets.~%~% ~
                                         The action area is separated from ~
                                         the content area with a horizontal ~
                                         line."))))
      (gtk-box-pack-start vbox label)
      (gtk-box-pack-start (gtk-dialog-get-content-area dialog) vbox)
      ;; Show the content area of the dialog
      (gtk-widget-show (gtk-dialog-get-content-area dialog)))
    ;; Add buttons with a stock id to the action area
    (gtk-dialog-add-button dialog "gtk-yes" :yes)
    (gtk-dialog-add-button dialog "gtk-no" :no)
    (gtk-dialog-add-button dialog "gtk-cancel" :cancel)
    (gtk-dialog-set-default-response dialog :cancel)
    ;; Change the order of the buttons
    (gtk-dialog-set-alternative-button-order dialog
                                             (list :yes :cancel :no))
    ;; Run the dialog and print the message on the console
    (format t "Response was: ~S~%" (gtk-dialog-run dialog))
    ;; Destroy the dialog
    (gtk-widget-destroy dialog)))

(defun create-message-dialog ()
  (let ((dialog (make-instance 'gtk-message-dialog
                               :message-type :info
                               :buttons :ok
                               :text "Info Message Dialog"
```

```
                                :secondary-text
                                (format nil
                                        "This is a message dialog of type ~
                                         :info with a secondary text.")))
    ;; Run the message dialog
    (gtk-dialog-run dialog)
    ;; Destroy the message dialog
    (gtk-widget-destroy dialog)))

(defun create-about-dialog ()
  (let ((dialog (make-instance 'gtk-about-dialog
                               :program-name "Example Dialog"
                               :version "0.00"
                               :copyright "(c) Dieter Kaiser"
                               :website
                               "github.com/crategus/cl-cffi-gtk"
                               :website-label "Project web site"
                               :license (license-text)
                               :authors '("Kalyanov Dmitry"
                                          "Dieter Kaiser")
                               :documenters '("Dieter Kaiser")
                               :artists '("None")
                               :logo-icon-name
                               "applications-development"
                               :wrap-license t)))
    ;; Run the about dialog
    (gtk-dialog-run dialog)
    ;; Destroy the about dialog
    (gtk-widget-destroy dialog)))

(defun example-dialog ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Dialog"
                                 :default-width 250
                                 :border-width 12))
          (vbox (make-instance 'gtk-vbox
                               :spacing 6)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-container-add window vbox)
      (let ((button (make-instance 'gtk-button
                                   :label "Open a Dialog Window")))
        (gtk-box-pack-start vbox button)
```

```
  (g-signal-connect button "clicked"
     (lambda (widget)
       (declare (ignore widget))
       ;; Create and show the dialog
       (create-dialog))))
(let ((button (make-instance 'gtk-button
                             :label "Open a Message Dialog")))
  (gtk-box-pack-start vbox button)
  (g-signal-connect button "clicked"
     (lambda (widget)
       (declare (ignore widget))
       ;; Create and show the message dialog
       (create-message-dialog))))
(let ((button (make-instance 'gtk-button
                             :label "Open an About Dialog")))
  (gtk-box-pack-start vbox button)
  (g-signal-connect button "clicked"
     (lambda (widget)
       (declare (ignore widget))
       ;; Create and show the about dialog
       (create-about-dialog))))
(gtk-box-pack-start vbox
                    (make-instance 'gtk-hseparator))
;; Create a quit button
(let ((button (make-instance 'gtk-button
                             :label "Quit")))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-widget-destroy window)))
  (gtk-box-pack-start vbox button))
(gtk-widget-show window))))
```

11.4 Text Entries

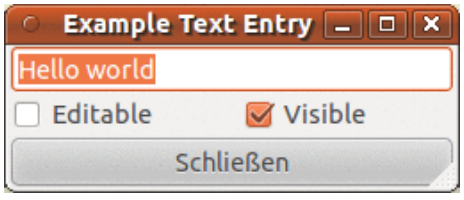

Figure 11.6: Text Entry

The entry widget allows text to be typed and displayed in a single line text box. The text may be set with function calls that allow new text to replace, prepend or append the current contents of the Entry widget.

Create a new Entry widget with the function `gtk-entry-new`. The function `gtk-entry-set-text` alters the text which is currently within the entry widget. 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 which contains some more functions for manipulating the contents.

The contents of the entry can be retrieved by using a call to the function `gtk-entry-get-text`. This is useful in the callback functions described below.

If we do not want the contents of the entry to be changed by someone typing into it, we can change its editable state with the function `gtk-editable-set-editable`. This function allows us to toggle the editable state of the entry widget by passing in a `T` or `NIL` value for the editable argument.

If we are using the entry where we do not want the text entered to be visible, for example when a password is being entered, we can use the function `gtk-entry-set-visibility`, which also takes a boolean flag.

A region of the text may be set as selected by using the function `gtk-editable-select-region`. This would most often be used after setting some default text in an Entry, making it easy for the user to remove it.

If we want to catch when the user has entered text, we can connect to the activate or changed signal. Activate is raised when the user hits the enter key within the entry widget. Changed is raised when the text changes at all, e.g., for every character entered or removed.

Example 11.4 is an example of using an entry widget.

Example 11.4: Text Entry

```
(defun example-text-entry ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :title "Example Text Entry"
                                  :default-width 250))
           (vbox (make-instance 'gtk-vbox))
           (hbox (make-instance 'gtk-hbox))
           (entry (make-instance 'gtk-entry
                                 :text "Hello"
                                 :max-length 50))
           (pos (gtk-entry-get-text-length entry)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (g-signal-connect entry "activate"
                        (lambda (widget)
                          (declare (ignore widget))
                          (format t "Entry contents: ~A"
                                  (gtk-entry-get-text entry))))
      (gtk-editable-insert-text entry " world" pos)
```

```
(gtk-editable-select-region entry 0 (gtk-entry-get-text-length entry))
(gtk-box-pack-start vbox entry :expand t :fill t :padding 0)
(let ((check (gtk-check-button-new-with-label "Editable")))
  (g-signal-connect check "toggled"
     (lambda (widget)
       (declare (ignore widget))
       (gtk-editable-set-editable
                                entry
                                (gtk-toggle-button-get-active check))))
  (gtk-box-pack-start hbox check))
(let ((check (gtk-check-button-new-with-label "Visible")))
  (gtk-toggle-button-set-active check t)
  (g-signal-connect check "toggled"
     (lambda (widget)
       (declare (ignore widget))
       (gtk-entry-set-visibility
                                entry
                                (gtk-toggle-button-get-active check))))
  (gtk-box-pack-start hbox check))
(gtk-box-pack-start vbox hbox)
(let ((button (gtk-button-new-from-stock "gtk-close")))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-widget-destroy window)))
  (gtk-box-pack-start vbox button))
(gtk-container-add window vbox)
(gtk-widget-show window))))
```

11.5 Spin Buttons

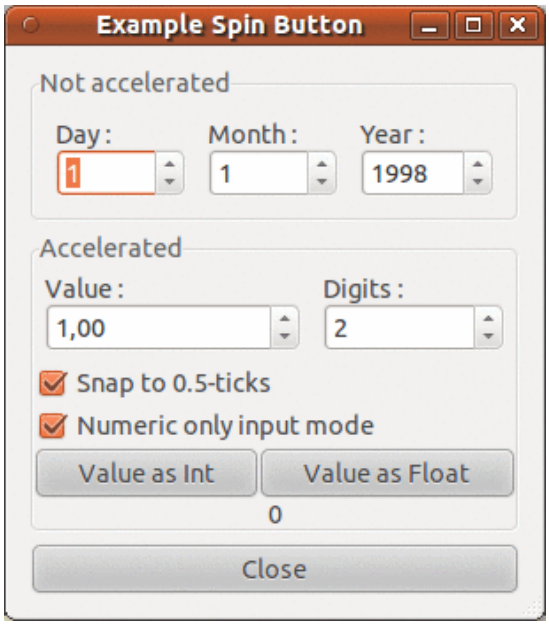

Figure 11.7: Spin Button

The spin button widget is generally used to allow the user to select a value from a range of numeric values. It consists of a text entry box with up and down arrow buttons attached to the side. Selecting one of the buttons causes the value to "spin" up and down the range of possible values. The entry box may also be edited directly to enter a specific value.

The spin button allows the value to have zero or a number of decimal places and to be incremented or decremented in configurable steps. The action of holding down one of the buttons optionally results in an acceleration of change in the value according to how long it is depressed.

The spin button uses an adjustment object to hold information about the range of values that the spin button can take. This makes for a powerful spin button widget.

Recall that an adjustment object is created with the function `gtk-adjustment-new`, which has the arguments `value`, `lower`, `step-increment`, `page-increment`, and `page-size`. These properties of an adjustment are used by the spin button in the following way:

`value` initial value for the Spin Button

`lower` lower range value

`upper` upper range value

`step-increment`
value to increment/decrement when pressing mouse button 1 on a button

`page_increment`
value to increment/decrement when pressing mouse button 2 on a button

`page_size`
unused

Additionally, mouse button 3 can be used to jump directly to the upper or lower values when used to select one of the buttons. A spin button is created with the function `gtk-spin-button-new`, which as the arguments `adjustment`, `climb-rate`, and `digits`.

The `climb-rate` argument take a value between 0.0 and 1.0 and indicates the amount of acceleration that the spin button has. The `digits` argument specifies the number of decimal places to which the value will be displayed.

A spin button can be reconfigured after creation using the function `gtk-spin-button-configure`. The first argument specifies the spin button that is to be reconfigured. The other arguments are as specified for the function `gtk-spin-button-new`.

The adjustment can be set and retrieved independently using the two functions `gtk-spin-button-set-adjustment` and `gtk-spin-button-get-adjustment`.

The number of decimal places can also be altered using the function `gtk-spin-button-set-digits` and the value that a spin button is currently displaying can be changed using the function `gtk-spin-button-set-value`.

The current value of a spin button can be retrieved as either a floating point or integer value with the functions `gtk-spin-button-get-value` and `gtk-spin-button-get-value-as-int`.

If you want to alter the value of a spin button relative to its current value, then the function `gtk-spin-button-spin` can be used, which has the three arguments `spin-button`, `direction`, and `increment`. The argument `direction` is of the enumeration type `GtkSpinType`, which can take one of the values shown in Table 11.6.

Table 11.6: The values of the `GtkSpinType` enumeration are used to specify the change to make in `gtk-spin-button-spin`

`:step-forward`
: Increment by the adjustments step increment.

`:backward`
: Decrement by the adjustments step increment.

`:forward` Increment by the adjustments page increment.

`:page-backward`
: Decrement by the adjustments page increment.

`:home` Go to the adjustments lower bound.

`:end` Go to the adjustments upper bound.

`:user-defined`
: Change by a specified amount.

`:step-forward` and `:step-backward` change the value of the spin button by the amount specified by increment, unless increment is equal to 0, in which case the value is changed by the value of `step-increment` in the adjustment.

`:page-forward` and `:page-backward` simply alter the value of the spin button by `increment`.

`:home` sets the value of the spin button to the bottom of the adjustments range and `:end` sets the value of the spin button to the top of the adjustments range.

`:user-defined` simply alters the value of the spin button by the specified amount.

We move away from functions for setting and retrieving the range attributes of the spin button now, and move onto functions that affect the appearance and behavior of the spin button widget itself.

The first of these functions is `gtk-spin-button-set-numeric`, which is used to constrain the text box of the spin button such that it may only contain a numeric value. This prevents a user from typing anything other than numeric values into the text box of a spin button.

You can set whether a Spin Button will wrap around between the upper and lower range values with the function `gtk-spin-button-set-wrap`. You can set a spin button to round the value to the nearest `step-increment`, which is set within the adjustment object used with the spin button. This is accomplished with the function `gtk-spin-button-set-snap-to-ticks`.

The update policy of a spin button can be changed with the function `gtk-spin-button-set-update-policy`. The possible values of policy are either `:always` or `:if-valid`. These policies affect the behavior of a Spin Button when parsing inserted text and syncing its value with the values of the adjustment.

In the case of `:if-valid` the spin button value only gets changed if the text input is a numeric value that is within the range specified by the adjustment. Otherwise the text is reset to the current value. In case of `:always` we ignore errors while converting text into a numeric value.

Finally, you can explicitly request that a spin button update itself with the function `gtk-spin-button-update`.

Example 11.5: Spin Button

```
(defun example-spin-button ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Spin Button"
                                 :default-width 300))
          (vbox (make-instance 'gtk-vbox
                               :homogeneous nil
                               :spacing 6
                               :border-width 12))
          (vbox1 (make-instance 'gtk-vbox
                                :homogeneous nil
                                :spacing 0
                                :border-width 6))
          (vbox2 (make-instance 'gtk-vbox
                                :homogeneous nil
                                :spacing 0
                                :boder-width 6))
          (hbox (make-instance 'gtk-hbox))
          (frame1 (make-instance 'gtk-frame
                                 :label "Not accelerated"))
```

```
    (frame2 (make-instance 'gtk-frame
                           :label "Accelerated"))
    (label (make-instance 'gtk-label
                          :label "0")))
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(let ((vbox (make-instance 'gtk-vbox))
      (spinner (make-instance 'gtk-spin-button
                              :adjustment
                              (make-instance 'gtk-adjustment
                                             :value 1.0
                                             :lower 1.0
                                             :upper 31.0
                                             :step-increment 1.0
                                             :page-increment 5.0
                                             :page-size 0.0)
                              :climb-rate 0
                              :digits 0
                              :wrap t)))
  (gtk-box-pack-start vbox
                      (make-instance 'gtk-label
                                     :label "Day :"
                                     :xalign 0
                                     :yalign 0.5)
                      :expand nil)
  (gtk-box-pack-start vbox spinner :expand nil)
  (gtk-box-pack-start hbox vbox :padding 6))
(let ((vbox (make-instance 'gtk-vbox))
      (spinner (make-instance 'gtk-spin-button
                              :adjustment
                              (make-instance 'gtk-adjustment
                                             :value 1.0
                                             :lower 1.0
                                             :upper 12.0
                                             :step-increment 1.0
                                             :page-increment 5.0
                                             :page-size 0.0)
                              :climb-rate 0
                              :digits 0
                              :wrap t)))
  (gtk-box-pack-start vbox
                      (make-instance 'gtk-label
                                     :label "Month :"
                                     :xalign 0
                                     :yalign 0.5)
```

```
                      :expand nil)
  (gtk-box-pack-start vbox spinner :expand nil)
  (gtk-box-pack-start hbox vbox :padding 6))
(let ((vbox (make-instance 'gtk-vbox))
      (spinner (make-instance 'gtk-spin-button
                              :adjustment
                              (make-instance 'gtk-adjustment
                                             :value 1.0
                                             :lower 1998.0
                                             :upper 2100.0
                                             :step-increment 1.0
                                             :page-increment 100.0
                                             :page-size 0.0)
                              :climb-rate 0
                              :digits 0
                              :wrap t)))
  (gtk-box-pack-start vbox
                      (make-instance 'gtk-label
                                     :label "Year :"
                                     :xalign 0
                                     :yalign 0.5)
                      :expand nil)
  (gtk-box-pack-start vbox spinner :expand nil :fill t)
  (gtk-box-pack-start hbox vbox :padding 6))
(gtk-box-pack-start vbox1 hbox :padding 6)
(gtk-container-add frame1 vbox1)
(gtk-box-pack-start vbox frame1)
(setq hbox (make-instance 'gtk-hbox))
(let ((vbox (make-instance 'gtk-vbox))
      (spinner1 (make-instance 'gtk-spin-button
                               :adjustment
                               (make-instance 'gtk-adjustment
                                              :value 1.0
                                              :lower -10000.0
                                              :upper  10000.0
                                              :step-increment 0.5
                                              :page-increment 100.0
                                              :page-size 0.0)
                               :climb-rate 1.0
                               :digits 2
                               :wrap t))
      (spinner2 (make-instance 'gtk-spin-button
                               :adjustment
                               (make-instance 'gtk-adjustment
                                              :value 2
                                              :lower 1
                                              :upper 5
```

```
                                          :step-increment 1
                                          :page-increment 1
                                          :page-size 0)
                              :climb-rate 0.0
                              :digits 0
                              :wrap t)))
(gtk-box-pack-start vbox
                    (make-instance 'gtk-label
                                   :label "Value :"
                                   :xalign 0
                                   :yalign 0.5)
                    :fill t)
(gtk-box-pack-start vbox spinner1 :expand nil)
(gtk-box-pack-start hbox vbox :padding 6)
(g-signal-connect spinner2 "value-changed"
   (lambda (widget)
     (declare (ignore widget))
     (gtk-spin-button-set-digits
                      spinner1
                      (gtk-spin-button-get-value-as-int spinner2))))
(setq vbox (make-instance 'gtk-vbox))
(gtk-box-pack-start vbox
                    (make-instance 'gtk-label
                                   :label "Digits :"
                                   :xalign 0
                                   :yalign 0.5)
                    :expand nil)
(gtk-box-pack-start vbox spinner2 :expand nil)
(gtk-box-pack-start hbox vbox :padding 6)
(gtk-box-pack-start vbox2 hbox :padding 6)
(let ((check (make-instance 'gtk-check-button
                            :label "Snap to 0.5-ticks"
                            :active t)))
  (g-signal-connect check "clicked"
     (lambda (widget)
       (gtk-spin-button-set-snap-to-ticks
                             spinner1
                             (gtk-toggle-button-get-active widget))))
  (gtk-box-pack-start vbox2 check))
(let ((check (make-instance 'gtk-check-button
                            :label "Numeric only input mode"
                            :active t)))
  (g-signal-connect check "clicked"
     (lambda (widget)
       (gtk-spin-button-set-numeric
                             spinner1
                             (gtk-toggle-button-get-active widget))))
```

```
    (gtk-box-pack-start vbox2 check))
  (gtk-container-add frame2 vbox2)
  (setq hbox (make-instance 'gtk-hbox))
  (let ((button (gtk-button-new-with-label "Value as Int")))
    (g-signal-connect button "clicked"
       (lambda (widget)
         (declare (ignore widget))
         (gtk-label-set-text
                label
                (format nil "~A"
                        (gtk-spin-button-get-value-as-int spinner1)))))
     (gtk-box-pack-start hbox button))
  (let ((button (gtk-button-new-with-label "Value as Float")))
    (g-signal-connect button "clicked"
       (lambda (widget)
         (declare (ignore widget))
         (gtk-label-set-text
                      label
                      (format nil "~A"
                              (gtk-spin-button-get-value spinner1)))))
    (gtk-box-pack-start hbox button))
  (gtk-box-pack-start vbox2 hbox)
  (gtk-box-pack-start vbox2 label))
(gtk-box-pack-start vbox frame2)
(let ((button (make-instance 'gtk-button
                             :label "Close")))
  (g-signal-connect button "clicked"
                    (lambda (widget)
                      (declare (ignore widget))
                      (gtk-widget-destroy window)))
  (gtk-box-pack-start vbox button))
(gtk-container-add window vbox)
(gtk-widget-show window))))
```

11.6 Combo Box

11.6.1 General Combo Box

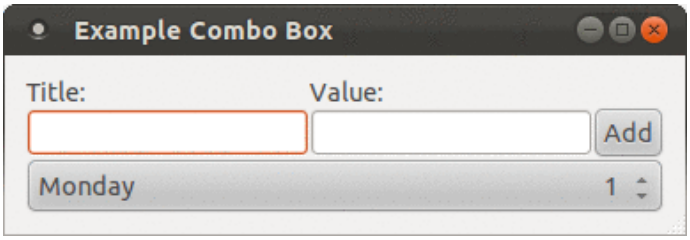

Figure 11.8: Combo Box

A GtkComboBox is a widget that allows the user to choose from a list of valid choices. The GtkComboBox displays the selected choice. When activated, the GtkComboBox displays a popup which allows the user to make a new choice. The style in which the selected value is displayed, and the style of the popup is determined by the current theme. It may be similar to a Windows-style combo box.

The GtkComboBox uses the model-view pattern; the list of valid choices is specified in the form of a tree model, and the display of the choices can be adapted to the data in the model by using cell renderers, as you would in a tree view. This is possible since GtkComboBox implements the GtkCellLayout interface. The tree model holding the valid choices is not restricted to a flat list, it can be a real tree, and the popup will reflect the tree structure.

To allow the user to enter values not in the model, the has-entry property allows the GtkComboBox to contain a GtkEntry. This entry can be accessed by calling gtk-bin-get-child on the combo box.

For a simple list of textual choices, the model-view API of GtkComboBox can be a bit overwhelming. In this case, GtkComboBoxText offers a simple alternative. Both GtkComboBox and GtkComboBoxText can contain an entry.

Example 11.6: Combo Box

```
(defstruct tvi
  title
  value)

(defun example-combo-box ()
  (within-main-loop
    (let* ((window (make-instance 'gtk-window
                                  :type :toplevel
                                  :border-width 12
                                  :title "Example Combo Box"))
           (model (make-instance 'array-list-store))
           (combo-box (make-instance 'gtk-combo-box :model model))
           (title-label (make-instance 'gtk-label :label "Title:"))
           (value-label (make-instance 'gtk-label :label "Value:"))
           (title-entry (make-instance 'gtk-entry))
           (value-entry (make-instance 'gtk-entry))
           (button (make-instance 'gtk-button :label "Add"))
           (table (make-instance 'gtk-table
                                 :n-rows 3
                                 :n-columns 3)))
      ;; Define two columns
      (store-add-column model "gchararray" #'tvi-title)
      (store-add-column model "gint" #'tvi-value)
      ;; Fill in data into the columns
      (store-add-item model (make-tvi :title "Monday" :value 1))
      (store-add-item model (make-tvi :title "Tuesday" :value 2))
      (store-add-item model (make-tvi :title "Wednesday" :value 3))
      (store-add-item model (make-tvi :title "Thursday" :value 4))
```

```
(store-add-item model (make-tvi :title "Friday" :value 5))
(store-add-item model (make-tvi :title "Saturday" :value 6))
(store-add-item model (make-tvi :title "Sunday" :value 7))
;; Set the first entry to active
(gtk-combo-box-set-active combo-box 0)
;; Define the signal handlers
(g-signal-connect window "destroy"
                  (lambda (w)
                    (declare (ignore w))
                    (gtk-main-quit)))
(g-signal-connect button "clicked"
   (lambda (widget)
     (declare (ignore widget))
     (store-add-item model
                     (make-tvi :title
                               (gtk-entry-text title-entry)
                               :value
                               (or (parse-integer
                                     (gtk-entry-text value-entry)
                                     :junk-allowed t)
                                   0)))))
(g-signal-connect combo-box "changed"
   (lambda (widget)
     (declare (ignore widget))
     (show-message (format nil "You clicked on row ~A~%"
                           (gtk-combo-box-get-active combo-box)))))
;; Create renderers for the cells
(let ((renderer (make-instance 'gtk-cell-renderer-text
                               :text "A text")))
  (gtk-cell-layout-pack-start combo-box renderer :expand t)
  (gtk-cell-layout-add-attribute combo-box renderer "text" 0))
(let ((renderer (make-instance 'gtk-cell-renderer-text
                               :text "A number")))
  (gtk-cell-layout-pack-start combo-box renderer :expand nil)
  (gtk-cell-layout-add-attribute combo-box renderer "text" 1))
;; Align the labels
(gtk-misc-set-alignment title-label 0.0 0.0)
(gtk-misc-set-alignment value-label 0.0 0.0)
;; Put the widgets into the table
(gtk-table-attach table title-label 0 1 0 1)
(gtk-table-attach table value-label 1 2 0 1)
(gtk-table-attach table title-entry 0 1 1 2)
(gtk-table-attach table value-entry 1 2 1 2)
(gtk-table-attach table button      2 3 1 2)
(gtk-table-attach table combo-box   0 3 2 3)
;; Put the table into the window
(gtk-container-add window table)
```

```
    ;; Show the window
    (gtk-widget-show window))))
```

11.6.2 Combo Box Text

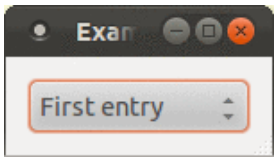

Figure 11.9: Combo Box Text

A `GtkComboBoxText` is a simple variant of `GtkComboBox` that hides the model-view complexity for simple text-only use cases.

To create a `GtkComboBoxText`, use `gtk-combo-box-text-new` or `gtk-combo-box-text-new-with-entry`.

You can add items to a `GtkComboBoxText` with `gtk-combo-box-text-append_text`, `gtk-combo-box-text-insert-text` or `gtk-combo-box-text-prepend-text` and remove options with `gtk-combo-box-text-remove`.

If the `GtkComboBoxText` contains an entry (via the `has-entry` property), its contents can be retrieved using `gtk-combo-box-text-get-active-text`. The entry itself can be accessed by calling `gtk-bin-get-child` on the combo box.

Example 11.7: Combo Box Text

```
(defun example-combo-box-text ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :border-width 12
                                 :title "Example Combo Box Text"))
          (combo (make-instance 'gtk-combo-box-text)))
      (gtk-combo-box-text-append-text combo "First entry")
      (gtk-combo-box-text-append-text combo "Second entry")
      (gtk-combo-box-text-append-text combo "Third entry")
      (gtk-combo-box-set-active combo 0)
      (gtk-container-add window combo)
      (gtk-widget-show window))))
```

11.7 Calendar

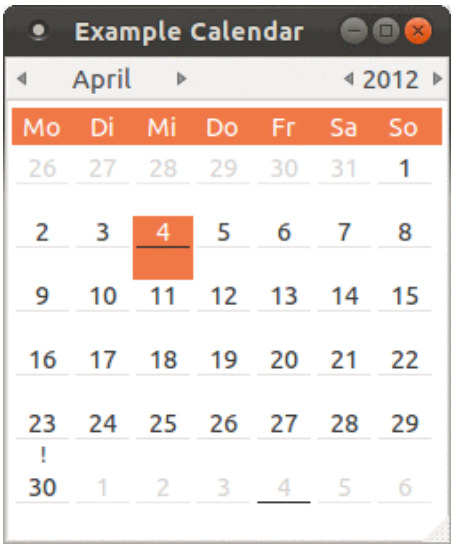

Figure 11.10: Calendar

`GtkCalendar` is a widget that displays a Gregorian calendar, one month at a time. It can be created with `gtk-calendar-new`.

The month and year currently displayed can be altered with `gtk-calendar-select-month`. The exact day can be selected from the displayed month using `gtk-calendar-select_day`.

To place a visual marker on a particular day, use `gtk-calendar-mark-day` and to remove the marker, `gtk-calendar-unmark-day`. Alternative, all marks can be cleared with `gtk-calendar-clear-marks`.

The way in which the calendar itself is displayed can be altered using `gtk-calendar-set-display-options`.

The selected date can be retrieved from a `GtkCalendar` using `gtk-calendar-get-date`.

Users should be aware that, although the Gregorian calendar is the legal calendar in most countries, it was adopted progressively between 1582 and 1929. Display before these dates is likely to be historically incorrect.

Example 11.8: Calendar

```
(defun calendar-detail (calendar year month day)
  (declare (ignore calendar year month))
  (when (= day 23)
    "!"))

(defun example-calendar ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Calendar"
                                 :type :toplevel
```

```
                            :default-width 250
                            :default-height 100))
    (calendar (make-instance 'gtk-calendar
                             :detail-function #'calendar-detail)))
(g-signal-connect window "destroy"
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(g-signal-connect calendar "day-selected"
                  (lambda (widget)
                    (declare (ignore widget))
                    (format t "selected: year ~A month ~A day ~A~%"
                            (gtk-calendar-year calendar)
                            (gtk-calendar-month calendar)
                            (gtk-calendar-day calendar))))
(gtk-container-add window calendar)
(gtk-widget-show window))))
```

12 Menu Widget

12.1 Manual Menu Creation

GtkMenuShell

A GtkMenuShell is the abstract base class used to derive the GtkMenu and GtkMenuBar subclasses.

A GtkMenuShell is a container of GtkMenuItem objects arranged in a list which can be navigated, selected, and activated by the user to perform application functions. A GtkMenuItem can have a submenu associated with it, allowing for nested hierarchical menus.

GtkMenuBar

The GtkMenuBar is a subclass of GtkMenuShell which contains one or more GtkMenuItems. The result is a standard menu bar which can hold many menu items.

GtkMenu

A GtkMenu is a GtkMenuShell that implements a drop down menu consisting of a list of GtkMenuItem objects which can be navigated and activated by the user to perform application functions.

A GtkMenu is most commonly dropped down by activating a GtkMenuItem in a GtkMenuBar or popped up by activating a GtkMenuItem in another GtkMenu.

A GtkMenu can also be popped up by activating a GtkOptionMenu. Other composite widgets such as the GtkNotebook can pop up a GtkMenu as well.

Applications can display a GtkMenu as a popup menu by calling the gtk_menu_popup() function. The example below shows how an application can pop up a menu when the 3rd mouse button is pressed.

GtkMenuItem

The GtkMenuItem widget and the derived widgets are the only valid childs for menus. Their function is to correctly handle highlighting, alignment, events and submenus.

As it derives from GtkBin it can hold any valid child widget, altough only a few are really useful.

GtkCheckMenuItem

A GtkCheckMenuItem is a menu item that maintains the state of a boolean value in addition to a GtkMenuItem usual role in activating application code.

A check box indicating the state of the boolean value is displayed at the left side of the GtkMenuItem. Activating the GtkMenuItem toggles the value.

GtkImageMenuItem

A GtkImageMenuItem is a menu item which has an icon next to the text label.

Note that the user can disable display of menu icons, so make sure to still fill in the text label.

GtkSeparatorMenuItem

The GtkSeparatorMenuItem is a separator used to group items within a menu. It displays a horizontal line with a shadow to make it appear sunken into the interface.

GtkTearoffMenuItem

A GtkTearoffMenuItem is a special GtkMenuItem which is used to tear off and reattach its menu.

When its menu is shown normally, the GtkTearoffMenuItem is drawn as a dotted line indicating that the menu can be torn off. Activating it causes its menu to be torn off and displayed in its own window as a tearoff menu.

When its menu is shown as a tearoff menu, the GtkTearoffMenuItem is drawn as dotted line which has a left pointing arrow graphic indicating that the tearoff menu can be reattached. Activating it will erase the tearoff menu window.

```
(defun example-menu ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :default-width 200
                                 :default-height 200
                                 :title "Example Menu Widget"))
          (vbox (make-instance 'gtk-vbox)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      ;; Create the menu-bar and the items of the menu-bar.
      (let ((menubar (make-instance 'gtk-menu-bar
                                    :visible t
                                    :can-focus nil))
            ;; Item file of the menu-bar.
            (item-file (make-instance 'gtk-menu-item
                                      :label "_Datei"
                                      :use-underline t))
            ;; Item edit of the menu-bar.
            (item-edit (make-instance 'gtk-menu-item
                                      :label "_Bearbeiten"
                                      :use-underline t))
            ;; Item help of the menu-bar.
            (item-help (make-instance 'gtk-menu-item
                                      :label "_Hilfe"
                                      :use-underline t)))
        ;; Create submenu for the item file.
        (let ((submenu (make-instance 'gtk-menu
                                      :visible t
                                      :can-focus nil))
              (item-file-new (make-instance 'gtk-image-menu-item
                                       :label "gtk-new"
                                       :use-underline t
                                       :use-stock t))
              (item-file-open (make-instance 'gtk-image-menu-item
                                        :label "gtk-open"
```

```
                                :use-underline t
                                :use-stock t))
        (item-file-save (make-instance 'gtk-image-menu-item
                                :label "gtk-save"
                                :user-underline t
                                :use-stock t))
        (item-file-save-as (make-instance 'gtk-image-menu-item
                                    :label "gtk-save-as"
                                    :user-underline t
                                    :use-stock t))
        (item-file-quit (make-instance 'gtk-image-menu-item
                                   :label "gtk-quit"
                                   :user-underline t
                                   :use-stock t)))
    ;; Add the items to to the submenu.
    (gtk-menu-shell-append submenu item-file-new)
    (gtk-menu-shell-append submenu item-file-open)
    (gtk-menu-shell-append submenu item-file-save)
    (gtk-menu-shell-append submenu item-file-save-as)
    ;; Insert a GtkSeparatorMenuItem.
    (gtk-menu-shell-append submenu (gtk-separator-menu-item-new))
    ;; Add the item file quit to the submenu
    (gtk-menu-shell-append submenu item-file-quit)
    ;; Set the submenu of the item file.
    (gtk-menu-item-set-submenu item-file submenu))
  ;; Create submenu for the item edit.
  (let ((submenu (make-instance 'gtk-menu
                                :visible t
                                :can-focus nil))
        (item-edit-cut (make-instance 'gtk-image-menu-item
                                   :label "gtk-cut"
                                   :use-underline t
                                   :use-stock t))
        (item-edit-copy (make-instance 'gtk-image-menu-item
                                    :label "gtk-copy"
                                    :use-underline t
                                    :use-stock t))
        (item-edit-paste (make-instance 'gtk-image-menu-item
                                     :label "gtk-paste"
                                     :user-underline t
                                     :use-stock t))
        (item-edit-delete (make-instance 'gtk-image-menu-item
                                      :label "gtk-delete"
                                      :user-underline t
                                      :use-stock t)))
    ;; Add the items to to the submenu.
    (gtk-menu-shell-append submenu item-edit-cut)
```

```
          (gtk-menu-shell-append submenu item-edit-copy)
          (gtk-menu-shell-append submenu item-edit-paste)
          (gtk-menu-shell-append submenu item-edit-delete)
          ;; Set the submenu of the item edit.
          (gtk-menu-item-set-submenu item-edit submenu))
        ;; Create submenu for the item help.
        (let ((submenu (make-instance 'gtk-menu
                                      :visible t
                                      :can-focus nil))
              (item-help-about (make-instance 'gtk-image-menu-item
                                              :label "gtk-about"
                                              :use-underline t
                                              :use-stock t)))
          ;; Add the items to to the submenu.
          (gtk-menu-shell-append submenu item-help-about)
          ;; Set the submenu of the item help.
          (gtk-menu-item-set-submenu item-help submenu))
        ;; Add the items file, edit, and help into the menubar.
        (gtk-menu-shell-append menubar item-file)
        (gtk-menu-shell-append menubar item-edit)
        (gtk-menu-shell-append menubar item-help)
        ;; Pack the menubar into the vbox.
        (gtk-box-pack-start vbox menubar))
      ;; Pack the vbox into the window.
      (gtk-container-add window vbox)
      ;; Show the window.
      (gtk-widget-show window))))

<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <requires lib="gtk+" version="2.24"/>
  <object class="GtkWindow" id="window">
    <property name="can_focus">False</property>
    <property name="title" translatable="yes">Example Menu Builder</property>
    <property name="has_resize_grip">False</property>
    <child>
      <object class="GtkMenuBar" id="menubar">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <child>
          <object class="GtkMenuItem" id="File">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <property name="use_action_appearance">False</property>
            <property name="label" translatable="yes">_Datei</property>
            <property name="use_underline">True</property>
            <child type="submenu">
```

```
<object class="GtkMenu" id="menu1">
  <property name="visible">True</property>
  <property name="can_focus">False</property>
  <child>
    <object class="GtkImageMenuItem" id="imagemenuitem1">
      <property name="label">gtk-new</property>
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="use_action_appearance">False</property>
      <property name="use_underline">True</property>
      <property name="use_stock">True</property>
    </object>
  </child>
  <child>
    <object class="GtkImageMenuItem" id="imagemenuitem2">
      <property name="label">gtk-open</property>
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="use_action_appearance">False</property>
      <property name="use_underline">True</property>
      <property name="use_stock">True</property>
    </object>
  </child>
  <child>
    <object class="GtkImageMenuItem" id="imagemenuitem3">
      <property name="label">gtk-save</property>
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="use_action_appearance">False</property>
      <property name="use_underline">True</property>
      <property name="use_stock">True</property>
    </object>
  </child>
  <child>
    <object class="GtkImageMenuItem" id="imagemenuitem4">
      <property name="label">gtk-save-as</property>
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="use_action_appearance">False</property>
      <property name="use_underline">True</property>
      <property name="use_stock">True</property>
    </object>
  </child>
  <child>
    <object class="GtkSeparatorMenuItem"
              id="separatormenuitem1">
      <property name="visible">True</property>
```

```
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
            </object>
          </child>
          <child>
            <object class="GtkImageMenuItem" id="imagemenuitem5">
              <property name="label">gtk-quit</property>
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
            </object>
          </child>
        </object>
      </child>
    </object>
  </child>
  <child>
    <object class="GtkMenuItem" id="Edit">
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="use_action_appearance">False</property>
      <property name="label" translatable="yes">_Bearbeiten</property>
      <property name="use_underline">True</property>
      <child type="submenu">
        <object class="GtkMenu" id="menu2">
          <property name="visible">True</property>
          <property name="can_focus">False</property>
          <child>
            <object class="GtkImageMenuItem" id="imagemenuitem6">
              <property name="label">gtk-cut</property>
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
            </object>
          </child>
          <child>
            <object class="GtkImageMenuItem" id="imagemenuitem7">
              <property name="label">gtk-copy</property>
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
```

```
            </object>
          </child>
          <child>
            <object class="GtkImageMenuItem" id="imagemenuitem8">
              <property name="label">gtk-paste</property>
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
            </object>
          </child>
          <child>
            <object class="GtkImageMenuItem" id="imagemenuitem9">
              <property name="label">gtk-delete</property>
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
            </object>
          </child>
        </object>
      </child>
    </object>
  </child>
  <child>
    <object class="GtkMenuItem" id="Help">
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="use_action_appearance">False</property>
      <property name="label" translatable="yes">_Hilfe</property>
      <property name="use_underline">True</property>
      <child type="submenu">
        <object class="GtkMenu" id="menu3">
          <property name="visible">True</property>
          <property name="can_focus">False</property>
          <child>
            <object class="GtkImageMenuItem" id="imagemenuitem10">
              <property name="label">gtk-about</property>
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="use_action_appearance">False</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
            </object>
          </child>
```

```
            </object>
          </child>
        </object>
      </child>
    </object>
  </child>
</object>
</interface>
```

```
(defun example-menu-builder ()
  (within-main-loop
    (let ((builder (make-instance 'gtk-builder)))
      (gtk-builder-add-from-file builder "example-menu-builder.ui")
      (g-signal-connect (gtk-builder-get-object builder "window") "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-widget-show (gtk-builder-get-object builder "window")))))
```

Appendix A Reference of Widgets

`GtkAdjustment` [Class]

A representation of an adjustable bounded value

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkAdjustment
```

Functions

```
gtk_adjustment_new
gtk_adjustment_get_value
gtk_adjustment_set_value
gtk_adjustment_clamp_page
gtk_adjustment_changed
gtk_adjustment_value_changed
gtk_adjustment_configure
gtk_adjustment_get_lower
gtk_adjustment_get_page_increment
gtk_adjustment_get_page_size
gtk_adjustment_get_step_increment
gtk_adjustment_get_minimum_increment
gtk_adjustment_get_upper
gtk_adjustment_set_lower
gtk_adjustment_set_page_increment
gtk_adjustment_set_page_size
gtk_adjustment_set_step_increment
gtk_adjustment_set_upper
```

Properties

```
"lower"                 gdouble           : Read / Write
"page-increment"        gdouble           : Read / Write
"page-size"             gdouble           : Read / Write
"step-increment"        gdouble           : Read / Write
"upper"                 gdouble           : Read / Write
"value"                 gdouble           : Read / Write
```

Signals

```
"changed"                                 : No Recursion
"value-changed"                           : No Recursion
```

`GtkAlignment` [Class]

A widget which controls the alignment and size of its child

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
```

```
                +----GtkContainer
                      +----GtkBin
                            +----GtkAlignment
```

Functions

```
gtk_alignment_new
gtk_alignment_set
gtk_alignment_get_padding
gtk_alignment_set_padding
```

Implemented Interfaces

`GtkAlignment` implements `AtkImplementorIface` and `GtkBuildable`.

Properties

```
"bottom-padding"         guint              : Read / Write
"left-padding"           guint              : Read / Write
"right-padding"          guint              : Read / Write
"top-padding"            guint              : Read / Write
"xalign"                 gfloat             : Read / Write
"xscale"                 gfloat             : Read / Write
"yalign"                 gfloat             : Read / Write
"yscale"                 gfloat             : Read / Write
```

`GtkAspectFrame` [Class]

A frame that constrains its child to a particular aspect ratio

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkBin
                         +----GtkFrame
                               +----GtkAspectFrame
```

Functions

```
gtk_aspect_frame_new
gtk_aspect_frame_set
```

Implemented Interfaces

`GtkAspectFrame` implements `AtkImplementorIface` and `GtkBuildable`.

Properties

```
"obey-child"             gboolean           : Read / Write
"ratio"                  gfloat             : Read / Write
"xalign"                 gfloat             : Read / Write
"yalign"                 gfloat             : Read / Write
```

`GtkBox` [Class]

A container box

Object Hierarchy

```
GObject
  +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkBox
                         +----GtkAppChooserWidget
                         +----GtkButtonBox
                         +----GtkColorChooserWidget
                         +----GtkColorSelection
                         +----GtkFileChooserButton
                         +----GtkFileChooserWidget
                         +----GtkFontChooserWidget
                         +----GtkFontSelection
                         +----GtkHBox
                         +----GtkInfoBar
                         +----GtkRecentChooserWidget
                         +----GtkStatusbar
                         +----GtkVBox
```

Functions

```
gtk_box_new
gtk_box_pack_start
gtk_box_pack_end
gtk_box_get_homogeneous
gtk_box_set_homogeneous
gtk_box_get_spacing
gtk_box_set_spacing
gtk_box_reorder_child
gtk_box_query_child_packing
gtk_box_set_child_packing
```

Implemented Interfaces

GtkBox implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

Properties

```
"homogeneous"           gboolean          : Read / Write
"spacing"               gint              : Read / Write
```

Child Properties

```
"expand"                gboolean          : Read / Write
"fill"                  gboolean          : Read / Write
"pack-type"             GtkPackType       : Read / Write
"padding"               guint             : Read / Write
"position"              gint              : Read / Write
```

`GtkContainer` [Class]

Base class for widgets which contain other widgets.

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkBin
                   +----GtkBox
                   +----GtkFixed
                   +----GtkGrid
                   +----GtkPaned
                   +----GtkIconView
                   +----GtkLayout
                   +----GtkMenuShell
                   +----GtkNotebook
                   +----GtkSocket
                   +----GtkTable
                   +----GtkTextView
                   +----GtkToolbar
                   +----GtkToolItemGroup
                   +----GtkToolPalette
                   +----GtkTreeView
```

Functions

```
gtk_container_add
gtk_container_remove
gtk_container_add_with_properties
gtk_container_get_resize_mode
gtk_container_set_resize_mode
gtk_container_check_resize
gtk_container_foreach
gtk_container_get_children
gtk_container_get_path_for_child
gtk_container_set_reallocate_redraws
gtk_container_get_focus_child
gtk_container_set_focus_child
gtk_container_get_focus_vadjustment
gtk_container_set_focus_vadjustment
gtk_container_get_focus_hadjustment
gtk_container_set_focus_hadjustment
gtk_container_resize_children
gtk_container_child_type
gtk_container_child_get
gtk_container_child_set
gtk_container_child_get_property
gtk_container_child_set_property
gtk_container_child_get_valist
gtk_container_child_set_valist
gtk_container_child_notify
```

```
gtk_container_forall
gtk_container_get_border_width
gtk_container_set_border_width
gtk_container_propagate_draw
gtk_container_get_focus_chain
gtk_container_set_focus_chain
gtk_container_unset_focus_chain
gtk_container_class_find_child_property
gtk_container_class_install_child_property
gtk_container_class_list_child_properties
gtk_container_class_handle_border_width
```

Implemented Interfaces

GtkContainer implements `AtkImplementorIface` and `GtkBuildable`.

Properties

```
"border-width"          guint              : Read / Write
"child"                 GtkWidget*         : Write
"resize-mode"           GtkResizeMode      : Read / Write
```

Signals

```
"add"                                      : Run First
"check-resize"                             : Run Last
"remove"                                   : Run First
"set-focus-child"                          : Run First
```

`GtkFixed` [Class]

A container which allows you to position widgets at fixed coordinates

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkFixed
```

Functions

```
gtk_fixed_new
gtk_fixed_put
gtk_fixed_move
```

Implemented Interfaces

`GtkFixed` implements `AtkImplementorIface` and `GtkBuildable`.

Child Properties

```
"x"                     gint               : Read / Write
"y"                     gint               : Read / Write
```

`GtkFrame` [Class]

A bin with a decorative frame and optional label

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkBin
                         +----GtkFrame
                               +----GtkAspectFrame
```

Functions

```
gtk_frame_new
gtk_frame_set_label
gtk_frame_set_label_widget
gtk_frame_set_label_align
gtk_frame_set_shadow_type
gtk_frame_get_label
gtk_frame_get_label_align
gtk_frame_get_label_widget
gtk_frame_get_shadow_type
```

Implemented Interfaces

`GtkFrame` implements `AtkImplementorIface` and `GtkBuildable`.

Properties

```
"label"                  gchar*             : Read / Write
"label-widget"           GtkWidget*         : Read / Write
"label-xalign"           gfloat             : Read / Write
"label-yalign"           gfloat             : Read / Write
"shadow-type"            GtkShadowType      : Read / Write
```

`GtkLabel` [Class]

A widget that displays a small to medium amount of text.

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkMisc
                   +----GtkLabel
                         +----GtkAccelLabel
```

Functions

```
gtk_label_new
gtk_label_set_text
gtk_label_set_attributes
gtk_label_set_markup
gtk_label_set_markup_with_mnemonic
gtk_label_set_pattern
gtk_label_set_justify
gtk_label_set_ellipsize
gtk_label_set_width_chars
```

```
gtk_label_set_max_width_chars
gtk_label_set_line_wrap
gtk_label_set_line_wrap_mode
gtk_label_get_layout_offsets
gtk_label_get_mnemonic_keyval
gtk_label_get_selectable
gtk_label_get_text
gtk_label_new_with_mnemonic
gtk_label_select_region
gtk_label_set_mnemonic_widget
gtk_label_set_selectable
gtk_label_set_text_with_mnemonic
gtk_label_get_attributes
gtk_label_get_justify
gtk_label_get_ellipsize
gtk_label_get_width_chars
gtk_label_get_max_width_chars
gtk_label_get_label
gtk_label_get_layout
gtk_label_get_line_wrap
gtk_label_get_line_wrap_mode
gtk_label_get_mnemonic_widget
gtk_label_get_selection_bounds
gtk_label_get_use_markup
gtk_label_get_use_underline
gtk_label_get_single_line_mode
gtk_label_get_angle
gtk_label_set_label
gtk_label_set_use_markup
gtk_label_set_use_underline
gtk_label_set_single_line_mode
gtk_label_set_angle
gtk_label_get_current_uri
gtk_label_set_track_visited_links
gtk_label_get_track_visited_links
```

Implemented Interfaces

`GtkLabel` implements `AtkImplementorIface` and `GtkBuildable`.

Properties

```
"angle"                gdouble            : Read / Write
"attributes"           PangoAttrList*     : Read / Write
"cursor-position"      gint               : Read
"ellipsize"            PangoEllipsizeMode : Read / Write
"justify"              GtkJustification   : Read / Write
"label"                gchar*             : Read / Write
"max-width-chars"      gint               : Read / Write
"mnemonic-keyval"      guint              : Read
```

```
"mnemonic-widget"        GtkWidget*         : Read / Write
"pattern"                gchar*             : Write
"selectable"             gboolean           : Read / Write
"selection-bound"        gint               : Read
"single-line-mode"       gboolean           : Read / Write
"track-visited-links"    gboolean           : Read / Write
"use-markup"             gboolean           : Read / Write
"use-underline"          gboolean           : Read / Write
"width-chars"            gint               : Read / Write
"wrap"                   gboolean           : Read / Write
"wrap-mode"              PangoWrapMode      : Read / Write
```

Signals

```
"activate-current-link"                     : Action
"activate-link"                             : Run Last
"copy-clipboard"                            : Action
"move-cursor"                               : Action
"populate-popup"                            : Run Last
```

`GtkLayout` [Class]

Infinite scrollable area containing child widgets and/or custom drawing

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkLayout
```

Functions

```
gtk_layout_new
gtk_layout_put
gtk_layout_move
gtk_layout_set_size
gtk_layout_get_size
gtk_layout_get_hadjustment
gtk_layout_get_vadjustment
gtk_layout_set_hadjustment
gtk_layout_set_vadjustment
gtk_layout_get_bin_window
```

Implemented Interfaces

`GtkLayout` implements `AtkImplementorIface`, `GtkBuildable` and `GtkScrollable`.

Properties

```
"height"                 guint              : Read / Write
"width"                  guint              : Read / Write
```

Child Properties

```
"x"                      gint               : Read / Write
"y"                      gint               : Read / Write
```

`GtkPaned` [Class]

A widget with two adjustable panes

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkPaned
                         +----GtkHPaned
                         +----GtkVPaned
```

Functions

```
gtk_paned_new
gtk_paned_add1
gtk_paned_add2
gtk_paned_pack1
gtk_paned_pack2
gtk_paned_get_child1
gtk_paned_get_child2
gtk_paned_set_position
gtk_paned_get_position
gtk_paned_get_handle_window
```

Implemented Interfaces

`GtkPaned` implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

Properties

```
"max-position"          gint              : Read
"min-position"          gint              : Read
"position"              gint              : Read / Write
"position-set"          gboolean          : Read / Write
```

Child Properties

```
"resize"                gboolean          : Read / Write
"shrink"                gboolean          : Read / Write
```

Style Properties

```
"handle-size"           gint              : Read
```

Signals

```
"accept-position"                         : Action
"cancel-position"                         : Action
"cycle-child-focus"                       : Action
"cycle-handle-focus"                      : Action
"move-handle"                             : Action
"toggle-handle-focus"                     : Action
```

`GtkProgressBar` [Class]

A widget which indicates progress visually

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkProgressBar
```

functions

```
gtk_progress_bar_new
gtk_progress_bar_pulse
gtk_progress_bar_set_fraction
gtk_progress_bar_get_fraction
gtk_progress_bar_set_inverted
gtk_progress_bar_get_inverted
gtk_progress_bar_set_show_text
gtk_progress_bar_get_show_text
gtk_progress_bar_set_text
gtk_progress_bar_get_text
gtk_progress_bar_set_ellipsize
gtk_progress_bar_get_ellipsize
gtk_progress_bar_set_pulse_step
gtk_progress_bar_get_pulse_step
```

Implemented Interfaces

`GtkProgressBar` implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

Properties

```
"ellipsize"              PangoEllipsizeMode : Read / Write
"fraction"               gdouble            : Read / Write
"inverted"               gboolean           : Read / Write
"pulse-step"             gdouble            : Read / Write
"show-text"              gboolean           : Read / Write
"text"                   gchar*             : Read / Write
```

Style Properties

```
"min-horizontal-bar-height" gint            : Read / Write
"min-horizontal-bar-width"  gint            : Read / Write
"min-vertical-bar-height"   gint            : Read / Write
"min-vertical-bar-width"    gint            : Read / Write
"xspacing"                  gint            : Read / Write
"yspacing"                  gint            : Read / Write
```

`GtkRange` [Class]

Base class for widgets which visualize an adjustment

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkRange
                   +----GtkScale
```

```
                    +----GtkScrollbar
```

Functions

```
gtk_range_get_fill_level
gtk_range_get_restrict_to_fill_level
gtk_range_get_show_fill_level
gtk_range_set_fill_level
gtk_range_set_restrict_to_fill_level
gtk_range_set_show_fill_level
gtk_range_get_adjustment
gtk_range_set_adjustment
gtk_range_get_inverted
gtk_range_set_inverted
gtk_range_get_value
gtk_range_set_value
gtk_range_set_increments
gtk_range_set_range
gtk_range_get_round_digits
gtk_range_set_round_digits

gtk_range_set_lower_stepper_sensitivity
gtk_range_get_lower_stepper_sensitivity
gtk_range_set_upper_stepper_sensitivity
gtk_range_get_upper_stepper_sensitivity
gtk_range_get_flippable
gtk_range_set_flippable
gtk_range_get_min_slider_size
gtk_range_get_range_rect
gtk_range_get_slider_range
gtk_range_get_slider_size_fixed
gtk_range_set_min_slider_size
gtk_range_set_slider_size_fixed
```

Implemented Interfaces

GtkRange implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

Properties

```
"adjustment"               GtkAdjustment*  : Read / Write /Construct
"fill-level"               gdouble         : Read / Write
"inverted"                 gboolean        : Read / Write
"lower-stepper-sensitivity" GtkSensitivityType : Read / Write
"restrict-to-fill-level"   gboolean        : Read / Write
"round-digits"             gint            : Read / Write
"show-fill-level"          gboolean        : Read / Write
"upper-stepper-sensitivity" GtkSensitivityType : Read / Write
```

Style Properties

```
"arrow-displacement-x"     gint            : Read
"arrow-displacement-y"     gint            : Read
```

```
"arrow-scaling"             gfloat          : Read
"slider-width"              gint            : Read
"stepper-size"              gint            : Read
"stepper-spacing"           gint            : Read
"trough-border"             gint            : Read
"trough-under-steppers"     gboolean        : Read
```

Signals

```
"adjust-bounds"                             : Run Last
"change-value"                              : Run Last
"move-slider"                               : Action
"value-changed"                             : Run Last
```

GtkScrollbar [Class]

A Scrollbar

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkRange
                   +----GtkScrollbar
                         +----GtkHScrollbar
                         +----GtkVScrollbar
```

Functions

```
gtk_scrollbar_new
```

Implemented Interfaces

GtkScrollbar implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

Style Properties

```
"fixed-slider-length"            gboolean   : Read
"has-backward-stepper"           gboolean   : Read
"has-forward-stepper"            gboolean   : Read
"has-secondary-backward-stepper" gboolean   : Read
"has-secondary-forward-stepper"  gboolean   : Read
"min-slider-length"              gint       : Read
```

GtkSeparator [Class]

A separator widget

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkSeparator
                   +----GtkHSeparator
                   +----GtkVSeparator
```

Functions

```
gtk_separator_new
```

Implemented Interfaces

GtkSeparator implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

`GtkStatusbar` [Class]

Report messages of minor importance to the user

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkBox
                         +----GtkStatusbar
```

Functions

```
gtk_statusbar_new
gtk_statusbar_get_context_id
gtk_statusbar_push
gtk_statusbar_pop
gtk_statusbar_remove
gtk_statusbar_remove_all
gtk_statusbar_get_message_area
```

Implemented Interfaces

`GtkStatusbar` implements `AtkImplementorIface`, `GtkBuildable` and `GtkOrientable`.

Style Properties

```
"shadow-type"            GtkShadowType      : Read
```

Signals

```
"text-popped"                               : Run Last
"text-pushed"                               : Run Last
```

`GtkWindow` [Class]

Toplevel which can contain other widgets.

Object Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GtkWidget
             +----GtkContainer
                   +----GtkBin
                         +----GtkWindow
                               +----GtkDialog
                               +----GtkApplicationWindow
                               +----GtkAssistant
                               +----GtkOffscreenWindow
```

```
                                  +----GtkPlug
```

Functions

```
gtk_window_new
gtk_window_set_title
gtk_window_set_wmclass
gtk_window_set_resizable
gtk_window_get_resizable
gtk_window_add_accel_group
gtk_window_remove_accel_group
gtk_window_activate_focus
gtk_window_activate_default
gtk_window_set_modal
gtk_window_set_default_size
gtk_window_set_default_geometry
gtk_window_set_geometry_hints
gtk_window_set_gravity
gtk_window_get_gravity
gtk_window_set_position
gtk_window_set_transient_for
gtk_window_set_attached_to
gtk_window_set_destroy_with_parent
gtk_window_set_hide_titlebar_when_maximized
gtk_window_set_screen
gtk_window_get_screen
gtk_window_is_active
gtk_window_has_toplevel_focus
gtk_window_list_toplevels
gtk_window_add_mnemonic
gtk_window_remove_mnemonic
gtk_window_mnemonic_activate
gtk_window_activate_key
gtk_window_propagate_key_event
gtk_window_get_focus
gtk_window_set_focus
gtk_window_get_default_widget
gtk_window_set_default
gtk_window_present
gtk_window_present_with_time
gtk_window_iconify
gtk_window_deiconify
gtk_window_stick
gtk_window_unstick
gtk_window_maximize
gtk_window_unmaximize
gtk_window_fullscreen
gtk_window_unfullscreen
```

```
gtk_window_set_keep_above
gtk_window_set_keep_below
gtk_window_begin_resize_drag
gtk_window_begin_move_drag
gtk_window_set_decorated
gtk_window_set_deletable
gtk_window_set_mnemonic_modifier
gtk_window_set_type_hint
gtk_window_set_skip_taskbar_hint
gtk_window_set_skip_pager_hint
gtk_window_set_urgency_hint
gtk_window_set_accept_focus
gtk_window_set_focus_on_map
gtk_window_set_startup_id
gtk_window_set_role
gtk_window_get_decorated
gtk_window_get_deletable
gtk_window_get_default_icon_list
gtk_window_get_default_icon_name
gtk_window_get_default_size
gtk_window_get_destroy_with_parent
gtk_window_get_hide_titlebar_when_maximized
gtk_window_get_icon
gtk_window_get_icon_list
gtk_window_get_icon_name
gtk_window_get_mnemonic_modifier
gtk_window_get_modal
gtk_window_get_position
gtk_window_get_role
gtk_window_get_size
gtk_window_get_title
gtk_window_get_transient_for
gtk_window_get_attached_to
gtk_window_get_type_hint
gtk_window_get_skip_taskbar_hint
gtk_window_get_skip_pager_hint
gtk_window_get_urgency_hint
gtk_window_get_accept_focus
gtk_window_get_focus_on_map
gtk_window_get_group
gtk_window_has_group
gtk_window_get_window_type
gtk_window_move
gtk_window_parse_geometry
gtk_window_reshow_with_initial_size
gtk_window_resize
gtk_window_resize_to_geometry
```

```
gtk_window_set_default_icon_list
gtk_window_set_default_icon
gtk_window_set_default_icon_from_file
gtk_window_set_default_icon_name
gtk_window_set_icon
gtk_window_set_icon_list
gtk_window_set_icon_from_file
gtk_window_set_icon_name
gtk_window_set_auto_startup_notification
gtk_window_get_opacity
gtk_window_set_opacity
gtk_window_get_mnemonics_visible
gtk_window_set_mnemonics_visible
gtk_window_get_focus_visible
gtk_window_set_focus_visible
gtk_window_set_has_resize_grip
gtk_window_get_has_resize_grip
gtk_window_resize_grip_is_visible
gtk_window_get_resize_grip_area
gtk_window_get_application
gtk_window_set_application
gtk_window_set_has_user_ref_count
```

Implemented Interfaces

GtkWindow implements `AtkImplementorIface` and `GtkBuildable`.

Properties

```
"accept-focus"           gboolean           : Read / Write
"application"            GtkApplication*    : Read / Write
"decorated"              gboolean           : Read / Write
"default-height"         gint               : Read / Write
"default-width"          gint               : Read / Write
"deletable"              gboolean           : Read / Write
"destroy-with-parent"    gboolean           : Read / Write
"focus-on-map"           gboolean           : Read / Write
"focus-visible"          gboolean           : Read / Write
"gravity"                GdkGravity         : Read / Write
"has-resize-grip"        gboolean           : Read / Write
"has-toplevel-focus"     gboolean           : Read
"icon"                   GdkPixbuf*         : Read / Write
"icon-name"              gchar*             : Read / Write
"is-active"              gboolean           : Read
"mnemonics-visible"      gboolean           : Read / Write
"modal"                  gboolean           : Read / Write
"opacity"                gdouble            : Read / Write
"resizable"              gboolean           : Read / Write
"resize-grip-visible"    gboolean           : Read
"role"                   gchar*             : Read / Write
```

```
  "screen"                GdkScreen*          : Read / Write
  "skip-pager-hint"       gboolean            : Read / Write
  "skip-taskbar-hint"     gboolean            : Read / Write
  "startup-id"            gchar*              : Write
  "title"                 gchar*              : Read / Write
  "transient-for"         GtkWindow*          : Read / Write / Construct
  "type"                  GtkWindowType       : Read / Write / Construct
  "type-hint"             GdkWindowTypeHint   : Read / Write
  "urgency-hint"          gboolean            : Read / Write
  "window-position"       GtkWindowPosition   : Read / Write
```

Style Properties

```
  "resize-grip-height"    gint                : Read / Write
  "resize-grip-width"     gint                : Read / Write
```

Signals

```
  "activate-default"                          : Action
  "activate-focus"                            : Action
  "keys-changed"                              : Run First
  "set-focus"                                 : Run Last
```

Appendix B Tables

Appendix C Figures

Appendix D Examples

Appendix E Function and Variable Index

G

H

I

J

L

M

O

P

R

S

T

V

W