GTK+ 2.0 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 developped 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.

At this time the Lisp library is developed using SBCL 1.0.53 on a Linux system with the version GTK+ 2.24 of the C library. In additon the library is tested on windows with SBCL 1.0.53 and the version GTK+ 2.24.

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 protability of GTK+, as some of the functions implemented here are not available or are nonstandard 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 offical GTK+ 2.0 Tutorioal of the C implementation.

2 Getting Started

2.1 A Simple Window

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-gtk-gtk)` from the Lisp prompt. The library is developed with the Lisp SBCL 1.0.53 on a Linux system and GTK+ 2.24. In addition the library is tested on Windows with SBCL 1.0.53 for Windows.

Information about the installation can be obtained with the function `cl-cffi-gtk-build-info`. This is an example for the output:

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

cl-cffi-gtk version: 0.0.0
cl-cffi-gtk build date: 22:25 2/26/2012
GTK+ version: 2.24.4
Machine type: X86
Machine version: Intel(R) Pentium(R) M processor 1.73GHz
Software type: Linux
Software version: 2.6.38-13-generic
Lisp implementation type: SBCL
Lisp implementation version: 1.0.53

NIL
```

The `cl-cffi-gtk` source distribution also 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 2.1.

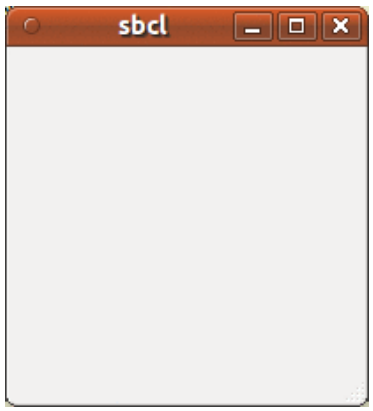


Figure 2.1: A Simple Window

The program creates a 200 x 200 pixel window. The window has the 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 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.

```
#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;
}
```

This is the corresponding Lisp program.

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

Example 2.1: A Simple Window

The program can be loaded into a Lisp session. But at first the package must be changed to `:gtk` after loading the library, so all symbols of the library are available.

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` displays the new window.

2.2 More about the Lisp binding to GTK+

Example 2.2 shows a second implementation of the simple program. This implementation uses the fact, that all GTK+ widgets are internally represented in the Lisp binding through a Lisp class. The 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.

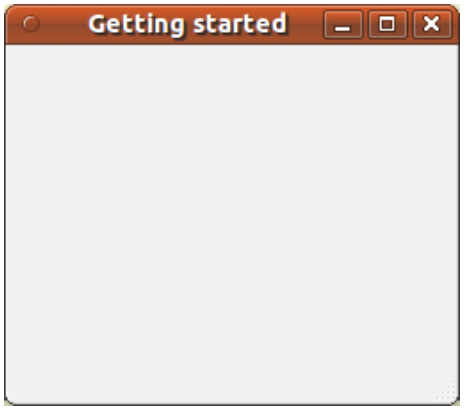


Figure 2.2: Getting started

In Example 2.2 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`. The result is shown in Figure 2.2.

```
(defun example-simple-window-2 ()
  (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 window))))
```

Example 2.2: Getting Started

The Example 2.2 shows, that the Lisp function `gtk-window-new` is not really needed. The function `gtk-window-new` is internally implemented 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.

2.3 Hello World in GTK+

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

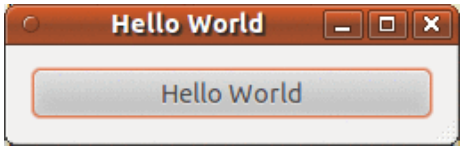


Figure 2.3: Hello World

```
#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 2.3. 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 the next section.

In Example 2.3 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 get the property `border-width`.

```
(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 window))))
```

Example 2.3: Hello World

An attentive reader notes that the function `gtk-widget-show` is not called for every single widget, which are in Example 2.3 the window and the button. The C library knows the function `gtk_widget_show_all()`, which shows the widget and all child widgets in one call. In the Lisp implementation the function `gtk-widget-show` is implemented using a keyword `:all`, which defaults to the value `T`. With the default value `T` the Lisp function `gtk-widget-show` internally calls the C function `gtk_widget_show_all()`.

Three more functions are used in Example 2.3. 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 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 17 shows how it is possible to put more than one widget into a window.

2.4 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 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 it 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.

2.5 An Upgraded Hello World

Figure 2.4 and Example 2.4 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.

Figure 2.4: Upgraded Hello World

```
#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 2.4 tries to be close to the C program. Therefore, the window and the box are created with the functions `gtk-window-new` and `gtk-h-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 2.3 the function `gtk-widget-show` is called only one time for the main window, because the default implementation of the Lisp function `gtk-widget-show` is to show all child widgets, too.

```
(defun example-upgraded-hello-world ()
  (within-main-loop
    (let ((window (gtk-window-new :toplevel))
          (box (gtk-h-box-new nil 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)
      (gtk-widget-show button)

      (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 window))))
```

Example 2.4: Upgraded Hello world

The second implementation in Example 2.5 makes even 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 class. Alternatively, the initialization of the variable `box` with the function `make-instance` is shown. In future examples of this tutorial the style shown in Example 2.5 is preferred.

```
(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-h-box
                              :homogeneous nil
                              :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 :expand t :fill t :padding 0))
      (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 :expand t :fill t :padding 0))
      (gtk-container-add window box)
      (gtk-widget-show window))))
```

Example 2.5: Second implementation of an Upgraded Hello World

A good exercise for the reader would be to insert a third "Quit" button that will exit the program. You may also wish to play with the options to `gtk-box-pack-start` while reading the next section. Try resizing the window, and observe the behavior.

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 widget so it could simply use a `gtk-container-add` call to "pack" the widget 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 horizontal box of the type `GtkHBox`, we use the function `gtk-hbox-new` or the call `(make-instance 'gtk-hbox)`, and for vertical boxes of the type `GtkVbox` the function `gtk-vbox-new` or the call `(make-instance 'gtk-vbox)`. The `gtk-box-pack-start` and `gtk-box-pack-end` functions are used to place objects inside of these containers. The `gtk-box-pack-start` function starts at the top and work its way down in a `GtkVBox`, and pack left to right in an `GtkHBox`. The function `gtk-box-pack-end` does the opposite, packing from bottom to top in a `GtkVBox`, and right to left in an `GtkHBox`. 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`.

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 a quite a bit of flexibility when placing widgets.

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.

A horizontal box is created with the function `gtk-hbox-new` and a vertical box with the function `gtk-vbox-new`. Alternatively, a box is created with the a call like `(make-instance 'gtk-hbox :homogenous val1 :spacing val2)` for a `GtkHBox`. The property `homogeneous` controls whether each widget in the box has the same width in a `GtkHBox` or the same height in a `GtkVBox`. The second argument `spacing` controls the amount of space between children in the box.

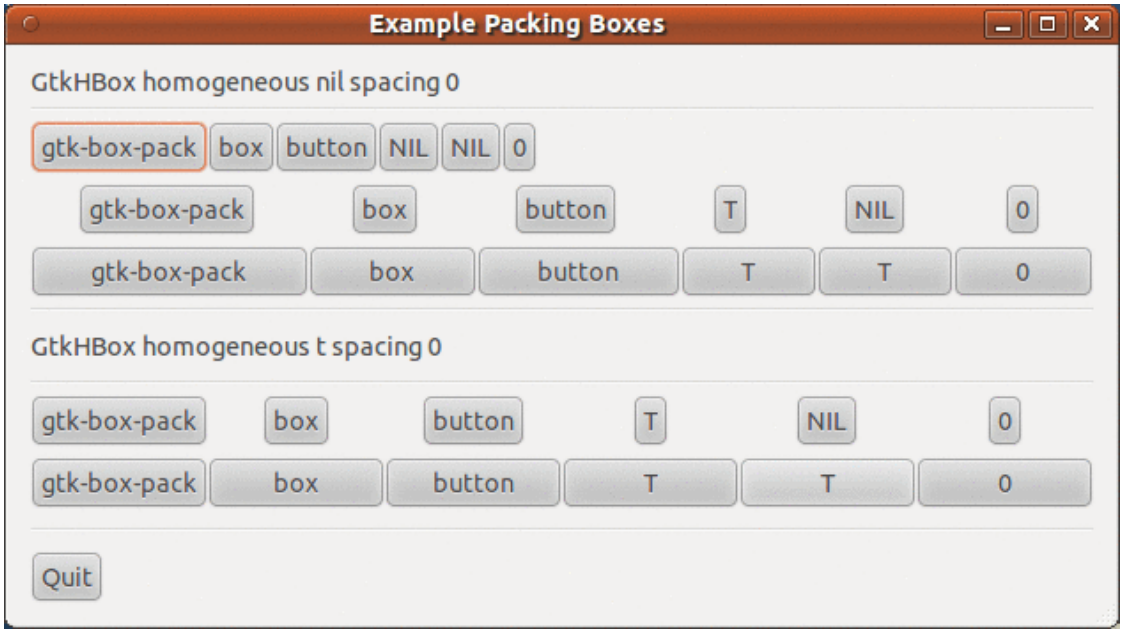


Figure 3.1: Packing Boxes, spacing is 0

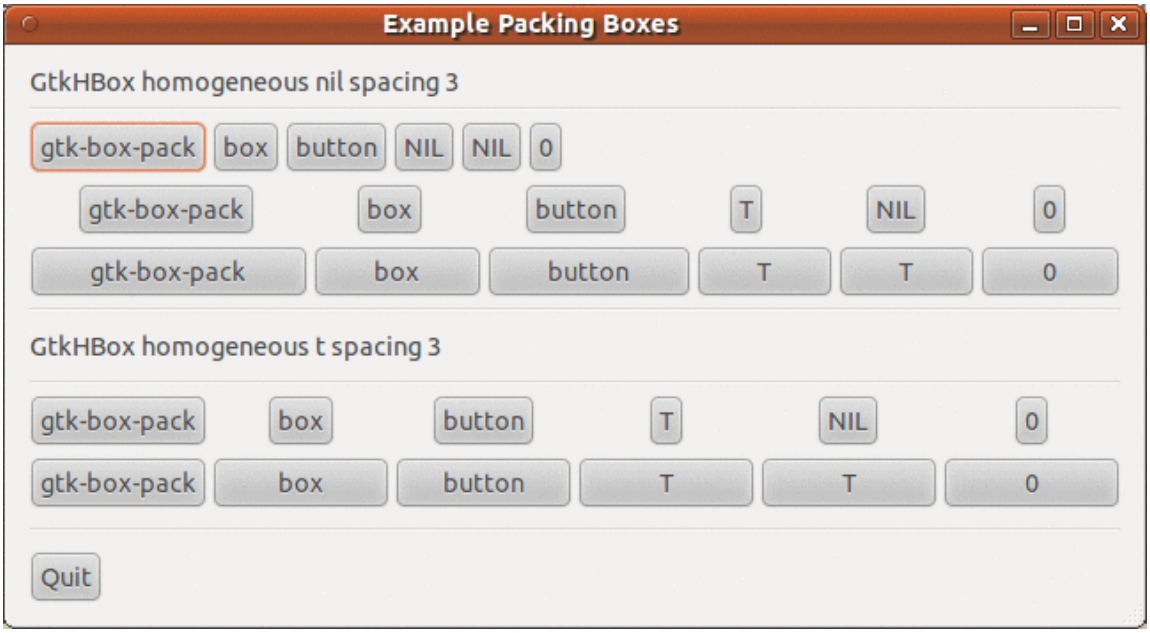


Figure 3.2: Example Packing Boxes, spacing is 3

Figure 3.1 and Figure 3.2 show two examples of packing buttons into horizontal boxes. In the second example a small space with a value of 3 is added. Each line of the examples contains one horizontal box of the type `GtkHBox` with several buttons. The first button represent 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. In the examples the child widget is a button. The

further arguments of gtk-box-pack-start are in C expand, fill and padding. In the Lisp binding to GTK+ these arguments are defined as the keywords :expand and :fill, which have a default value of T, and :padding with a default value of 0.

The keyword argument :expand 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 (T); or the box is shrunk to just fit the widgets (NIL). 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 to the gtk-box-pack functions control whether the extra space is allocated to the objects themselves (T), or as extra padding in the box around these objects (NIL). 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 and Figure 3.2 is shown in Example 3.1

Example 3.1: Example Packing Boxes

```
(defun make-box (homogeneous spacing expand fill padding)
  (let ((box (make-instance 'gtk-hbox
                            :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-packing-boxes (&optional (spacing 0))
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Packing Boxes"
                                 :type :toplevel
                                 :border-width 12
                                 :default-height 200
                                 :default-width 300))
          (vbox (make-instance 'gtk-vbox
                               :homogeneous nil
                               :spacing 6))
          (button (make-instance 'gtk-button
                                 :label "Quit"))
          (quitbox (make-instance 'gtk-hbox
                                  :homogeneous nil
                                  :spacing 0)))
      (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
                                         "GtkHBox homogeneous nil spacing ~A"
                                                 spacing)
                                         :xalign 0
                                         :yalign 0)
                          :expand nil
                          :fill nil
                          :padding 0)
```

```
(gtk-box-pack-start vbox
                    (make-instance 'gtk-h-separator)
                    :expand nil
                    :fill t
                    :padding 0)
(gtk-box-pack-start vbox
                    (make-box nil spacing nil nil 0)
                    :expand nil
                    :fill nil
                    :padding 0)
(gtk-box-pack-start vbox
                    (make-box nil spacing t nil 0)
                    :expand nil
                    :fill nil
                    :padding 0)
(gtk-box-pack-start vbox
                    (make-box nil spacing t t 0)
                    :expand nil
                    :fill nil
                    :padding 0)
(gtk-box-pack-start vbox
                    (make-instance 'gtk-h-separator)
                    :expand nil
                    :fill t
                    :padding 0)
(gtk-box-pack-start vbox
                    (make-instance 'gtk-label
                                   :label
                                   (format nil
                                   "GtkHBox homogeneous t spacing ~A"
                                           spacing)
                                   :xalign 0
                                   :yalign 0)
                    :expand nil
                    :fill nil
                    :padding 5)
(gtk-box-pack-start vbox
                    (make-instance 'gtk-h-separator)
                    :expand nil
                    :fill t
                    :padding 0)
(gtk-box-pack-start vbox
                    (make-box t spacing t nil 0)
                    :expand nil
                    :fill nil
                    :padding 0)
(gtk-box-pack-start vbox
```

```
                    (make-box t spacing t t 0)
                    :expand nil
                    :fill nil
                    :padding 0)
  (gtk-box-pack-start vbox
                    (make-instance 'gtk-h-separator)
                    :expand nil
                    :fill t
                    :padding 5)
  (gtk-box-pack-start quitbox button :expand nil :fill nil :padding 0)
  (gtk-box-pack-start vbox quitbox :expand nil :fill nil :padding 0)
  (gtk-container-add window vbox)
  (gtk-widget-show 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 3.3. Note that the coordinate system starts in the upper left hand corner.

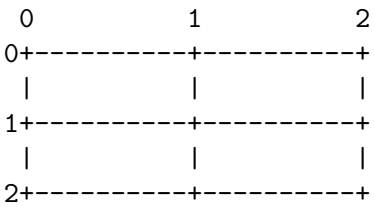


Figure 3.3: Layout of a table with `rows = 2` and `columns = 2`

To place a widget into a box, 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 in 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 `:xoptions` and `:yoptions` are of 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. The options 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: Options of the enumeraton 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. 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.

3.4 Table Packing Example

Figure 3.4 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.

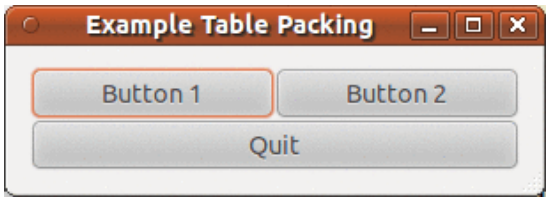


Figure 3.4: Table packing

Figure 3.5 is extended 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 a later chapter of this tutorial. The code of Figure 3.5 is shown in Example 3.3

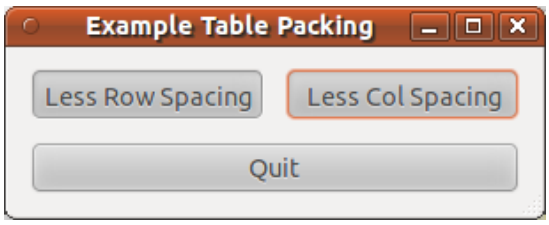


Figure 3.5: Table packing with more spacing

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))))
```

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"
```

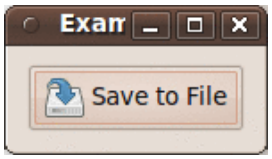
```
                  (lambda (widget)
                    (declare (ignore widget))
                    (gtk-main-quit)))
(g-signal-connect button1 "toggled"
   (lambda (widget)
     (if (gtk-toggle-button-get-active widget)
         (progn
           (gtk-table-set-row-spacings table 12)
           (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))))
```

4 Widget Overview

5 The Button Widget

5.1 Normal Buttons

We've almost seen all there is to see of the button widget. It's pretty simple. There is however more than one way to create a button. You can use the `gtk-button-new-with-label` or `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's 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 usual `gtk-box-pack-start`, and then use `gtk-container-add` to pack the box into the button.



Here's an example of using `gtk-button-new` to create a button with a image and a label in it. The code to create a box is breaken up from the rest so you can use it in your programs. There are further examples of using images later in the tutorial.

```
(defun xpm-label-box (filename text)
  (let ((box (make-instance 'gtk-h-box
                            :homogeneous nil
                            :spacing 0
                            :border-width 2))
        (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 2)
    (gtk-box-pack-start box label :expand nil :fill nil :padding 2)
    box))

(defun example-button ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Cool Button"
                                 :type :toplevel
                                 :border-width 10))
          (button (make-instance 'gtk-button))
          (box (xpm-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 window))))
```

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

The next example shows various buttons 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`.


```
(defun example-buttons ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Buttons"
                                 :type :toplevel
                                 :default-width 250
                                 :border-width 10))
          (vbox1 (make-instance 'gtk-v-box :spacing 5))
          (vbox2 (make-instance 'gtk-v-box :spacing 5))
          (hbox  (make-instance 'gtk-h-box :spacing 5)))
      (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 window))))
```

5.2 Toggle Buttons

Toggle buttons are derived from normal buttons and are very similar, except they will always be in one of two states, alternated by a click. They may be depressed, and when clicked again, they will pop back up. Click again, and they will pop back down. 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 creates a blank toggle button, and the last two, a 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 structure to determine its state. 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/NIL` value is returned from the function `gtk-toggle-button-get-active`.

5.3 Check Buttons

Check buttons inherit many properties and functions from the 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.

5.4 Radio Buttons

Radio buttons are similar to check buttons except they are grouped so that only one may be selected/depressed at a time. 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. The example below should make this clear.

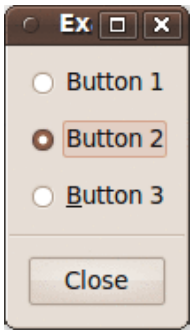
You can shorten this slightly by using the following syntax, which removes the need for a variable to hold the list of buttons:

```
(setq button2
```

```
(gtk-radio-button-new-with-label (gtk-radio-button-get-group button)
                                 "Button 2"))
```

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`.

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).



The following example creates a radio button group with three buttons.

```
(defun example-radio-buttons ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :title "Example Radio Buttons"
                                 :type :toplevel
                                 :border-width 0))
          (vbox1 (make-instance 'gtk-v-box
                                :homogeneous nil
                                :spacing 0))
          (vbox2 (make-instance 'gtk-v-box
                                :homogeneous nil
                                :spacing 10
                                :border-width 10))
          (vbox3 (make-instance 'gtk-v-box
                                :homogeneous nil
                                :spacing 10
                                :border-width 10))
          (button nil))
      (gtk-container-add window vbox1)
      (gtk-box-pack-start vbox1 vbox2 :expand t :fill t :padding 0)
```

```
(setq button (gtk-radio-button-new-with-label nil "Button 1"))
(gtk-box-pack-start vbox2 button :expand t :fill t :padding 0)

(setq button
      (gtk-radio-button-new-with-label
                                    (gtk-radio-button-get-group button)
                                    "Button 2"))
(gtk-toggle-button-set-active button t)
(gtk-box-pack-start vbox2 button :expand t :fill t :padding 0)

(setq button
      (gtk-radio-button-new-with-mnemonic
                                    (gtk-radio-button-get-group button)
                                    "_Button 3"))
(gtk-box-pack-start vbox2 button :expand t :fill t :padding 0)

(gtk-box-pack-start vbox1
                    (make-instance 'gtk-h-separator)
                    :expand nil :fill nil :padding 0)
(gtk-box-pack-start vbox1 vbox3 :expand nil :fill t :padding 0)

(gtk-box-pack-start vbox3
                    (setq button
                          (make-instance 'gtk-button :label "Close")))

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

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

6 Range Widgets

7 Miscellaneous Widgets

7.1 Labels

Labels are used a lot in GTK, and are relatively simple. 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 EventBox widget or a Button widget.

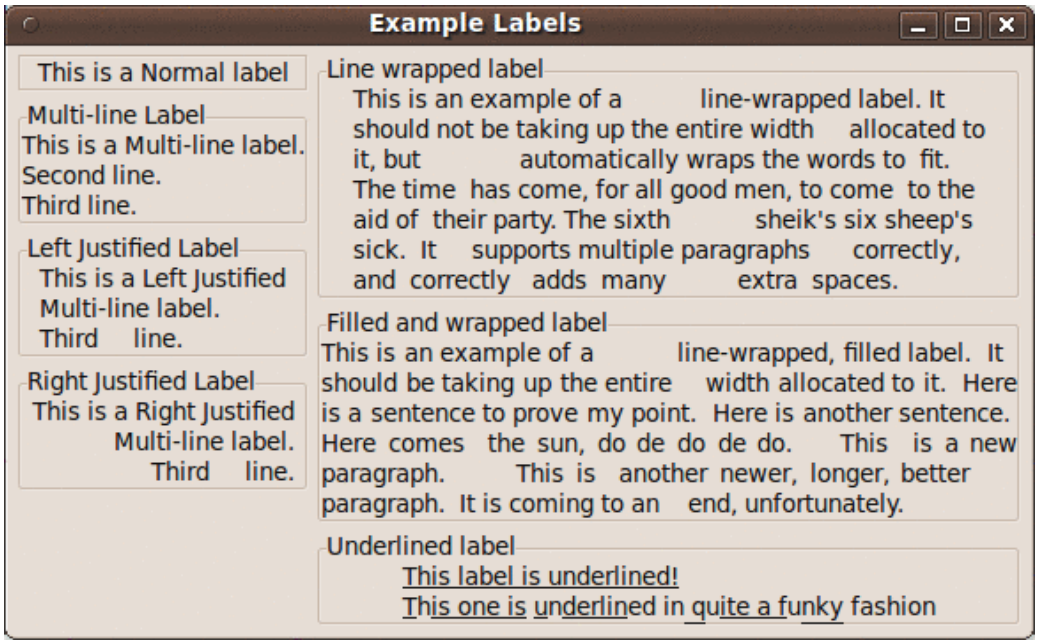
To create a new label, use `gtk-label-new` or `gtk-label-new-with-mnemonic`. The sole argument is the string you wish the label to display. To change the label's text after creation, use the function `gtk-label-set-text`. The first argument is the label you created previously, and the second is the new 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`. The label text can be justified using `gtk-label-set-justify`. The first argument is the label and the second argument one of the following keyword of the enumeration type `GtkJustification`

`:left`

`:center` (the default)

`:fill`



The label widget is also 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 wrap argument take `T` or `NIL`.

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.

Note:

If you simply want to have an underlined accelerator ("mnemonic") in your label, you should use `gtk-label-new-with-mnemonic` or `gtk-label-set-text-with_mnemonic`, not `gtk-label-set-pattern`.

Below is an example to illustrate these functions. This example makes use of the Frame widget to better demonstrate the label styles. You can ignore this for now as the Frame widget is explained later on.

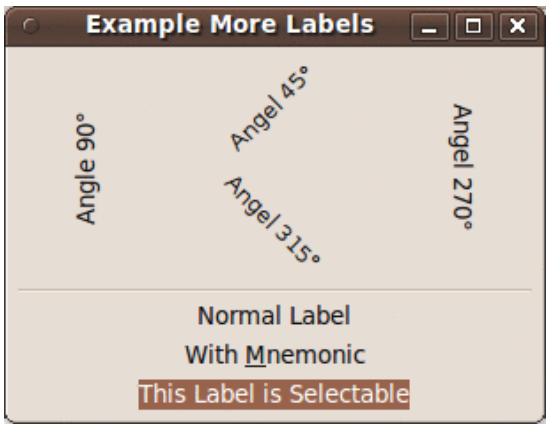
In GTK+ 2.0, label texts can contain markup for font and other text attribute changes, and labels may be selectable (for copy-and-paste). These advanced features won't be explained here.

```
(defun example-label ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Demo Label"
                                 :border-width 5))
          (vbox (make-instance 'gtk-v-box
                               :homogeneous nil
                               :spacing 5))
          (hbox (make-instance 'gtk-h-box
                               :homogeneous nil
                               :spacing 5))
          (frame (make-instance 'gtk-frame
                                :title "Normal Label"))
          (label (make-instance 'gtk-label
                                :label "This is a Normal label")))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-container-add window hbox)
      (gtk-box-pack-start hbox vbox :expand nil :fill nil :padding 0)
      (gtk-container-add frame label)
      (gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
      (setq frame (make-instance 'gtk-frame
                                 :label "Multi-line Label"))
      (setq label (make-instance 'gtk-label
                                 :label
                                 (format nil "This is a Multi-line label.~%~
                                              Second line.~%~
                                              Third line.")))
      (gtk-container-add frame label)
      (gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
```

```
(setq frame (make-instance 'gtk-frame
                           :label "Left Justified Label"))
(setq label (make-instance 'gtk-label
                           :justify :left
                           :label
                           (format nil
                                   "This is a Left Justified~%~
                                    Multi-line label.~%~
                                    Third     line.")))
(gtk-container-add frame label)
(gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
(setq frame (make-instance 'gtk-frame
                           :label "Right Justified Label"))
(setq label (make-instance 'gtk-label
                           :justify :right
                           :label
                           (format nil
                                   "This is a Right Justified~%~
                                    Multi-line label.~%~
                                    Third     line.")))
(gtk-container-add frame label)
(gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
(setq vbox (make-instance 'gtk-v-box
                          :homogeneous nil
                          :spacing 5))
(gtk-box-pack-start hbox vbox :expand nil :fill nil :padding 0)
(setq frame (make-instance 'gtk-frame
                           :label "Line wrapped label"))
(setq label (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.")))
(gtk-container-add frame label)
(gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
(setq frame (make-instance 'gtk-frame
```

```
                            :label "Filled and wrapped label"))
      (setq label (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.")))
      (gtk-container-add frame label)
      (gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
      (setq frame (make-instance 'gtk-frame
                            :label "Underlined label"))
      (setq label (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")))
      (gtk-container-add frame label)
      (gtk-box-pack-start vbox frame :expand nil :fill nil :padding 0)
      (gtk-widget-show window))))
```

The following example shows some more possibilities with labels.




```
(defun example-more-labels ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example More Labels"
                                 :default-width 300
                                 :border-width 5))
          (vbox1 (make-instance 'gtk-v-box
                                :homogeneous nil
                                :spacing 5))
          (vbox2 (make-instance 'gtk-v-box
                                :homogeneous nil
                                :spacing 5))
          (hbox (make-instance 'gtk-h-box
                               :homogeneous nil
                               :spacing 5)))
      (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-h-separator))
(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-container-add window vbox2)
(gtk-widget-show window))))
```

7.2 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`:

`:up`

`:down`

`:left`

`:right`

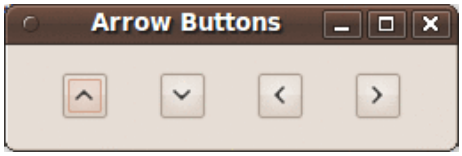
These values obviously indicate the direction in which the arrow will point. The shadow type argument is of the enumeration type `GtkShadowType` and may take one of these values:

`:in`

`:out` (the default)

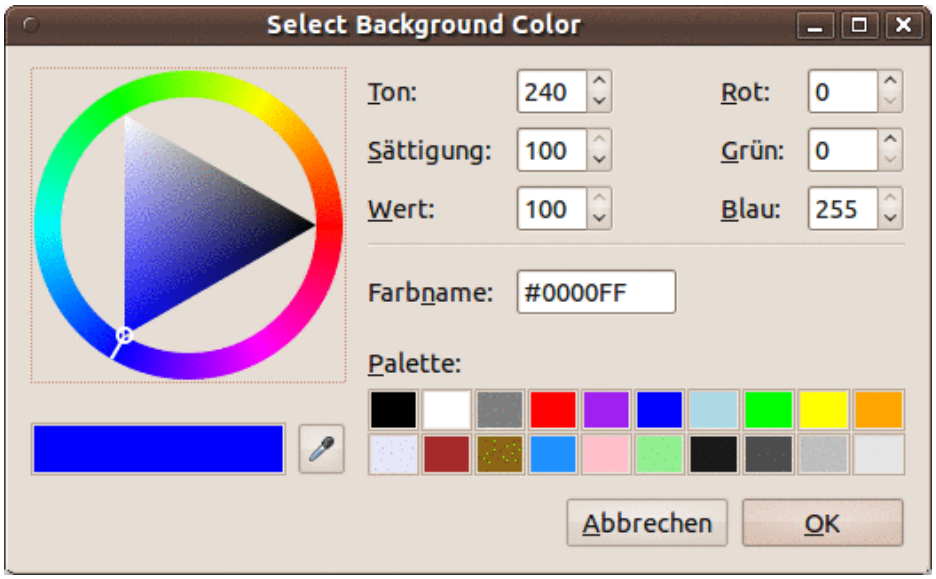
`:etched-in`
`:etched-out`

Here's a brief example to illustrate their use.



```
(defun create-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))
    button))

(defun example-arrows ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Arrow Buttons"
                                 :default-width 250
                                 :border-width 10))
          (box (make-instance 'gtk-h-box
                              :homogeneous t
                              :spacing 0
                              :border-width 5)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (gtk-main-quit)))
      (gtk-box-pack-start box
                          (create-button :up :in)
                          :expand nil :fill nil :padding 3)
      (gtk-box-pack-start box
                          (create-button :down :out)
                          :expand nil :fill nil :padding 3)
      (gtk-box-pack-start box
                          (create-button :left :etched-in)
                          :expand nil :fill nil :padding 3)
      (gtk-box-pack-start box
                          (create-button :right :etched-out)
                          :expand nil :fill nil :padding 3)
      (gtk-container-add window box)
      (gtk-widget-show window))))
```

7.3 Color Selection



The color selection widget is a widget for interactive selection of colors. This composite widget lets the user select a color by manipulating RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value) triples. This is done either by adjusting single values with sliders or entries, or by picking the desired color from a hue-saturation wheel/value bar. Optionally, the opacity of the color can also be set.

The widget comes in two flavors `GtkColorSelection` and `GtkColorSelectionDialog`. A `GtkColorSelection` widget is created with `(make-instance 'gtk-color-selection)` or the function `gtk-color-selection-new`. The function `gtk-color-selection-new` does not have an argument. The most common color selection is the dialog created with `(make-instance 'gtk-color-selection-dialog)` or the function `gtk-color-selection-dialog-new`. This function takes one argument, which is a string and specifies the title of the dialog window.

The color selection widget currently emits only one signal, `"color-changed"`, which is emitted whenever the current color in the widget changes, either when the user changes the color or if the color is set explicitly through the function `gtk-color-selection-set-current-color` or the accessor function `gtk-color-selection-current-color`.

The color selection widget supports adjusting the opacity of a color (also known as the alpha channel). The opacity control is disabled by default. Calling the function `gtk-color-selection-set-has-opacity-control` with the argument `has_opacity` set to `T` enables opacity. Likewise, `has_opacity` set to `NIL` will disable opacity.

You can set the current color explicitly by calling `gtk-color-selection-set-current_color` with an argument `color` of type `GdkColor`. Setting the opacity (alpha channel) is done with `gtk-color-selection-set-current-alpha`. The alpha value should be between 0 (fully transparent) and 65535 (fully opaque). When you need to query the current settings, typically when the `"color-changed"` signal is received, you can use the functions `gtk-color-selection-get-current-color` and `gtk-color-selection-get-current-alpha`.

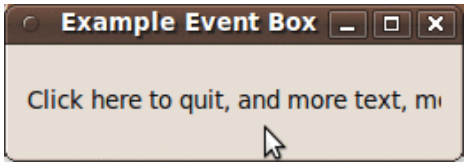
The simple example demonstrates the use of the `ColorSelectionDialog`. The program displays a window containing a drawing area. Clicking on it opens a color selection dialog, and changing the color in the color selection dialog changes the background color.

```
(let ((color (make-gdk-color :red 0
                             :blue 65535
                             :green 0)))

  (defun drawing-area-event (widget event area)
    (declare (ignore widget))
    (let ((handled nil))
      (when (eql (gdk-event-type event) :button-press)
        (let* ((colorseldlg (make-instance 'gtk-color-selection-dialog
                                           :title "Select Background Color"))
               (colorsel
                 (gtk-color-selection-dialog-color-selection colorseldlg)))
          (setq handled t)
          (gtk-color-selection-set-previous-color colorsel color)
          (gtk-color-selection-set-current-color colorsel color)
          (gtk-color-selection-set-has-palette colorsel t)
          (g-signal-connect colorsel "color-changed"
             (lambda (widget)
               (declare (ignore widget))
               (let ((color (gtk-color-selection-current-color colorsel)))
                 (gtk-widget-modify-bg area :normal color))))
          (let ((response (gtk-dialog-run colorseldlg)))
            (gtk-widget-destroy colorseldlg)
            (if (eql response :ok)
                (setq color (gtk-color-selection-get-current-color colorsel))
                (gtk-widget-modify-bg area :normal color)))))
      handled))

  (defun example-color-selection ()
    (within-main-loop
      (let ((window (make-instance 'gtk-window
                                   :title "Example Color Selection"
                                   :default-width 300))
            (area (make-instance 'gtk-drawing-area)))
        (g-signal-connect window "destroy"
                          (lambda (widget)
                            (declare (ignore widget))
                            (gtk-widget-destroy window)))
        (gtk-widget-modify-bg 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 window)))))
```

8 Container Widgets

8.1 The Event Box



Some GTK+ widgets do not have associated X windows, so they 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.

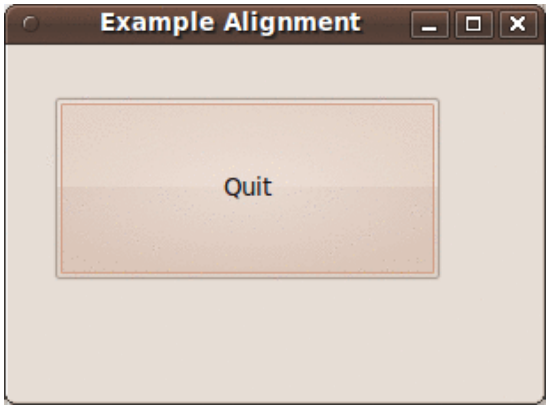
To create a new `GtkEventBox` widget, use the call `(make-instance 'gtk-event-box)` or the function `gtk-event-box-new`. A child widget can then be added to this `GtkEventBox` with the function `gtk-container-add`. With the function `gtk-widget-set-events` a signal is connected to the `GtkEventBox` widget.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

The following example 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.

```
(defun example-event-box ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Event Box"
                                 :border-width 10))
          (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))))
```

8.2 The 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.

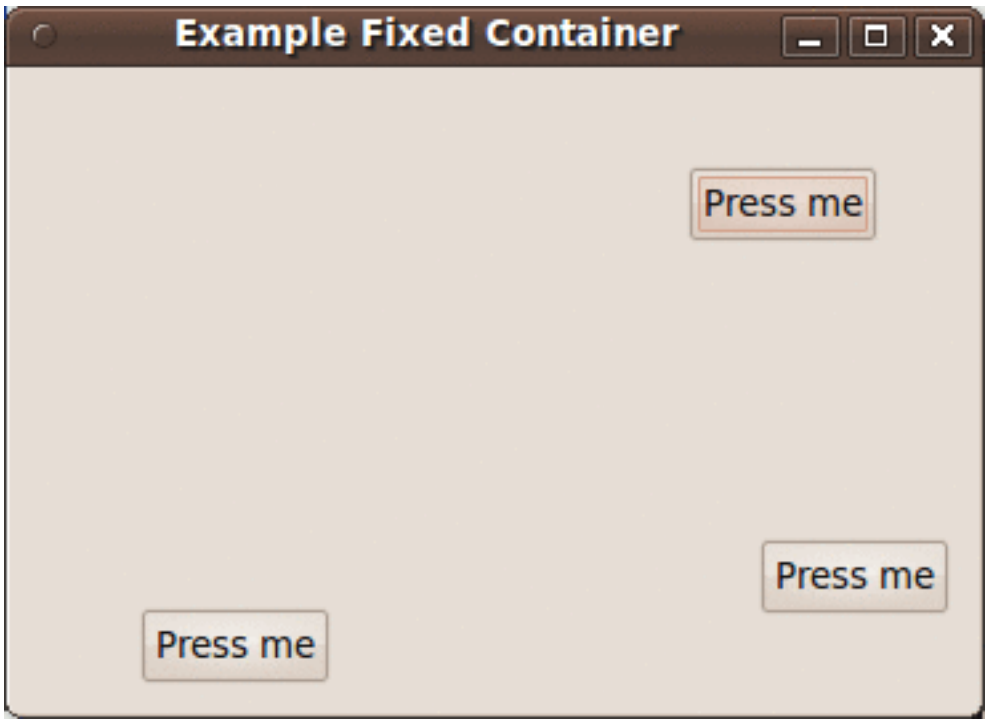
The functions `gtk-alignment-new` and `gtk-alignment-set` are associated with the `GtkAlignment` widget. The first function creates a new `GtkAlignment` widget with specified parameters. The second function allows the alignment parameters of an exisiting `GtkAlignment` widget to be altered.

All four alignment parameters are floating point numbers which can range from 0.0 to 1.0. The `xalign` and `yalign` arguments affect the position of the widget placed within the `GtkAlignment` widget. The `xscale` and `yscale` arguments affect the amount of space allocated to the widget. A child widget can be added to the `GtkAlignment` widget using the function `gtk-container-add`.

```
(defun example-alignment ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Alignment"
                                 :border-width 10))
          (button (make-instance 'gtk-button
                                 :label "Quit"))
          (alignment (make-instance 'gtk-alignment
```

```
                                :xalign 0.25
                                :yalign 0.25
                                :xscale 0.75
                                :yscale 0.50)))
    (g-signal-connect window "destroy"
                      (lambda (widget)
                        (declare (ignore widget))
                        (gtk-widget-destroy window)))
    (gtk-container-add alignment button)
    (gtk-container-add window alignment)
    (gtk-widget-show window))))
```

8.3 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. `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.

Normally, Fixed widgets do not have have their own X window. Since this is different from the behaviour of `GtkFixed` widgets in earlier releases of GTK+, the function `gtk-fixed-set-has-window` allows the creation of `GtkFixed` widgets with their own X window. The function has to be called before realizing the widget.

The following example illustrates how to use a 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 400
                                 :default-height 300
                                 :border-width 10))
          (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 300) (random 200))))
      (gtk-widget-show window))))
```

8.4 Layout Container

The Layout container is similar to the Fixed container 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 add and move 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`.

The final four functions for use with Layout widgets are for manipulating the horizontal and vertical adjustment widgets: `gtk-layout-get-hadjustment`, `gtk-layout-get-vadjustment`, `gtk-layout-set-hadjustment`, and `gtk-layout-set-vadjustment`.

This is the same example already presented for a Fixed Container using a Layout Container.

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

(defun example-layout ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Example Layout Container"
                                 :default-width 300
                                 :default-height 200
                                 :border-width 10))
          (layout (make-instance 'gtk-layout)))
      (g-signal-connect window "destroy"
                        (lambda (window)
                          (declare (ignore window))
                          (gtk-main-quit)))
      (gtk-container-add window layout)
      (dotimes (i 3)
        (let ((button (gtk-button-new-with-label "Press me")))
          (g-signal-connect button "clicked"
                            (lambda (widget)
                              (move-button widget layout)))
          (gtk-layout-put layout button (random 300) (random 200))))
      (gtk-widget-show window))))
```

8.5 Frames

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`.

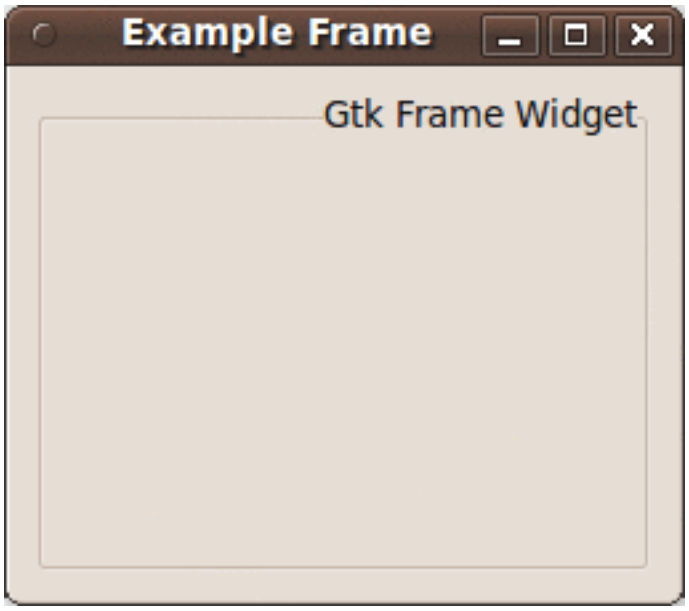
The position of the label can be changed using the function `gtk-frame-set-label-align` which has the arguments xalign and yalign which take values between 0.0 and 1.0. xalign indicates the position of the label along the top horizontal of the frame. yalign is not currently used. The default value of 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 type argument can take one of the following values:

- `:none`

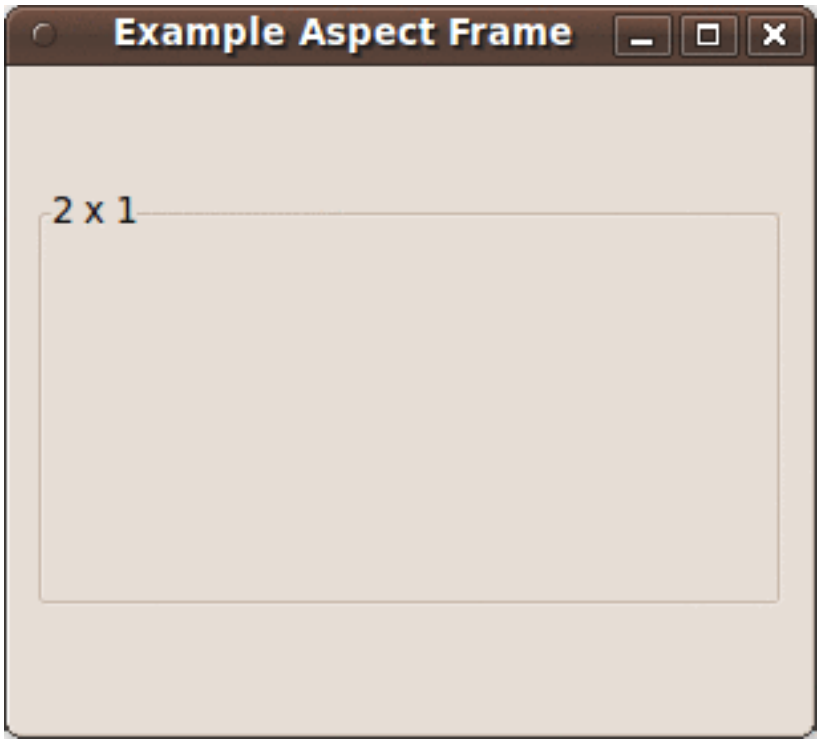
- `:in`
- `:out`
- `:etched-in (the default)`
- `:etched-out`

The following code example illustrates the use of the 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 10))
          (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))))
```

8.6 Aspect Frames



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`. xalign and yalign specify alignment as with Alignment widgets. If obey_child is TRUE, 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 AspectFrame to present a drawing area whose aspect ratio will always be 2:1, no matter how the user resizes the top-level window.

```
(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 10))
          (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))))
```

9 Menu Widget

Appendix A Tables

Appendix B Figures

Appendix C Examples

Appendix D Function and Variable Index