GTK+ 2.0 Tutorial for Lisp

Version 0.0

Short Contents

Table of Contents

1 Introduction

The library cl-cffi-gtk is a Lisp binding to the GTK (GIMP Toolkit) which is a library for creating graphical user interfaces. It is licensed using the LGPL which has been adopted for the cl-cffi-gtk 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 this work is to document the library much more complete and to do the implementation as consistent as possible. Most informations about GTK can be gained by reading the C documentation. Therefore, the C documentation from www.gtk.org is included into the Lisp files to document the Lisp binding to the GTK library. This way the calling conventions are easier to determine and missing functionality is easier to detect.

The GTK library is called the GIMP toolkit because it 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, it is implemented using the idea of classes and callback functions (pointers to functions).

There is also a third component called GLib which contains a few replacements for some standard calls, as well as some additional functions for handling linked lists, etc. The replacement functions are used to increase GTK's portability, 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 GTK's class hierarchy, 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. There are GTK bindings for many other languages including C++, Guile, Perl, Python, TOM, Ada95, Objective C, Free Pascal, Eiffel, Java and C#. If you intend to use another language's bindings to GTK, look at that binding's documentation first.

2 Getting Started

The first thing to do, of course, is to download the cl-cffi-gtk source and to install it. The latest version is always available from the repository at `github.com/crategus/cl-cffi-gtk`. cl-cffi-gtk can be loaded with the command `(asdf:operate 'asdf:load-op :cl-gtk-gtk)` from the Lisp prompt.

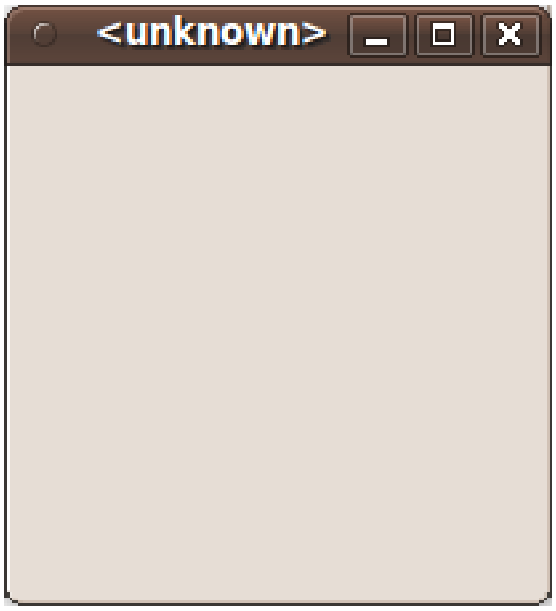
The library is developed with the Lisp SBCL 1.0.53 on a Linux system and GTK+ 2.24. In addition the library is tested on Windows with SBCL 1.0.53 for Windows.

[...

Add more about installation on Linux and Windows and required libraries

...]

The cl-cffi-gtk source distribution also contains the complete source to all of the examples used in this tutorial. To begin this introduction to GTK, the simplest program possible is shown.



This program will create a 200 x 200 pixel window, The window has the the title <unknown>. The usual actions of a window are possible. The window can be sized and be 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. It is more compact due to the style of Lisp.

```
(defun example-simple-window ()
  (within-main-loop
    (let ((window (gtk-window-new :toplevel)))
      (gtk-widget-show window))))
```

The program can be loaded into a Lisp session. But at first the package should be changed to 'cl-gtk-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 this macro corresponds to the C functions `gtk_init` and `gtk_main` which initialize and start a GTK program.

Only two functions are need 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. That is all.

At this place a second implementation is shown. 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. Therefore, the 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. For this example, the property `:type` with the keyword `:toplevel` creates again a toplevel window. In addition a title is set and the width of the window is a little enlarged.

```
(defun example-simple-window-2 ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Getting started"
                                 :default-width 250)))
      (gtk-widget-show window))))
```

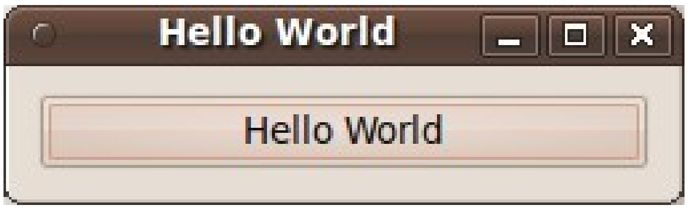
The function `gtk-window-new` is not really needed and is internally implemented simply as

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

Later we will see that a Lisp program can be formulated much more compact with the help of `make-instance`.

Hello World in GTK

Now for a program with a button. It is the classic hello world for GTK. Again the C program is shown first to learn more about the differences between a C and a Lisp implementation.



```
#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. One difference is, that the function `make-instance` is used to create a window and a button. Another point is, that the definition of callback functions is avoided. The callback functions are very short and implemented through lambda-functions in Lisp. More about signals and callbacks follows in the next section.

```
(defun example-hello-world ()
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Hello World"
                                 :default-width 250
                                 :border-width 10))
          (button (make-instance 'gtk-button :label "Hello World")))
      (g-signal-connect button "clicked"
                        (lambda (button)
                          (declare (ignore button))
                          (format t "Hello world.~%")
                          (gtk-widget-destroy window)))
      (g-signal-connect window "delete-event"
                        (lambda (window event)
                          (declare (ignore window event))
                          (format t "Delete Event Occured.~%")
                          t))
      (g-signal-connect window "destroy-event"
                        (lambda (window event)
                          (declare (ignore window event))
                          (gtk-main-quit)))
      (gtk-container-add window button)
      (gtk-widget-show window))))
```

Theory of Signals and Callbacks

Before we look in detail at `example-hello-world`, we'll discuss signals and callbacks. GTK is an event driven toolkit, which means it 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 similiar function which has the arguments `widget`, `name`, and `func`.

```
(g-signal-connect widget name 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 comparable Lisp implementation we simply declare a lambda function as a callback function which is passed as the third argument.

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

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

```
(defun big-event-handler (widget event arg1 arg2 arg3)
  [ code of the big event handler] )


   (g-signal-connect window "destroy-event"
                     (lambda (widget event)
                       (big-event-handler widget event 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-event" #'big-event-handler)
```

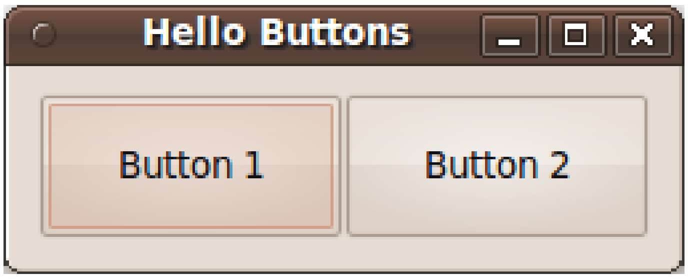
Furthermore, the C function

```
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.

An Upgraded Hello World

Let's take a look at a slightly improved helloworld with better examples of callbacks. This will also introduce the next topic, packing widgets. First, the C program is shown.



```
#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;
}
```

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.

The following Lisp implementation tries to be as close as possible 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`. All these functions have corresponding C functions which have a `_` instead of a `-` in the name, e.g. the C function `gtk_window_set_title` has the equivalent Lisp function `gtk-window-set-title`. There is another important point about the Lisp implementation. Because `title` is a property of the class `gtk-window`, it is not necessary to use the accessor functions `gtk-window-set-title` or `gtk-window-get-title` in Lisp. The slots of the class can be read or write with the accessor of the slot which is `gtk-window-title`, e.g. `(gtk-window-set-title window title)` is equivalent to `(setf (gtk-window-title window) title)`.

```
(defun example-upgraded-hello-world ()
  (within-main-loop
    (let ((window (gtk-window-new :toplevel))
          (box    (gtk-h-box-new nil 5))
          (button  nil))
      (g-signal-connect window "delete_event"
                        (lambda (widget event)
                          (declare (ignore widget event))
                          (gtk-main-quit)))
      (gtk-window-set-title window "Hello Buttons")
      (gtk-window-set-default-size window 250 75)
      (gtk-container-set-border-width window 10)
      (gtk-container-add window box)

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

The second implementation of the C program makes more use of the Lisp style. The window is created with the Lisp function `make-instance`. All desired properties of the window are initialized by assigning default 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 this style 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 10))
          (box    (make-instance 'gtk-h-box
                                 :homogeneous nil
                                 :spacing 5))
          (button  nil))
      (g-signal-connect window "delete_event"
                        (lambda (widget event)
                          (declare (ignore widget event))
                          (gtk-main-quit)))
      (gtk-container-add window box)

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

3 Packing Widgets

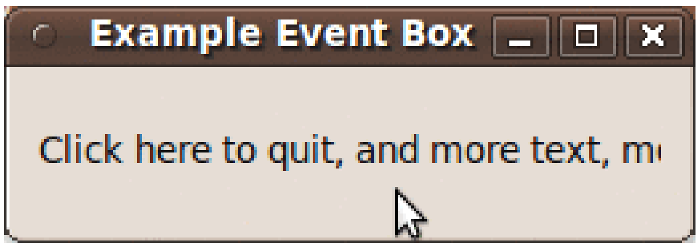
4 Widget Overview

5 The Button Widget

6 Range Widgets

7 Container Widgets

7.1 The EventBox



Some GTK widgets don't 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 don't clip so you can get messy overwriting, etc. If you require more from these widgets, the EventBox is for you.

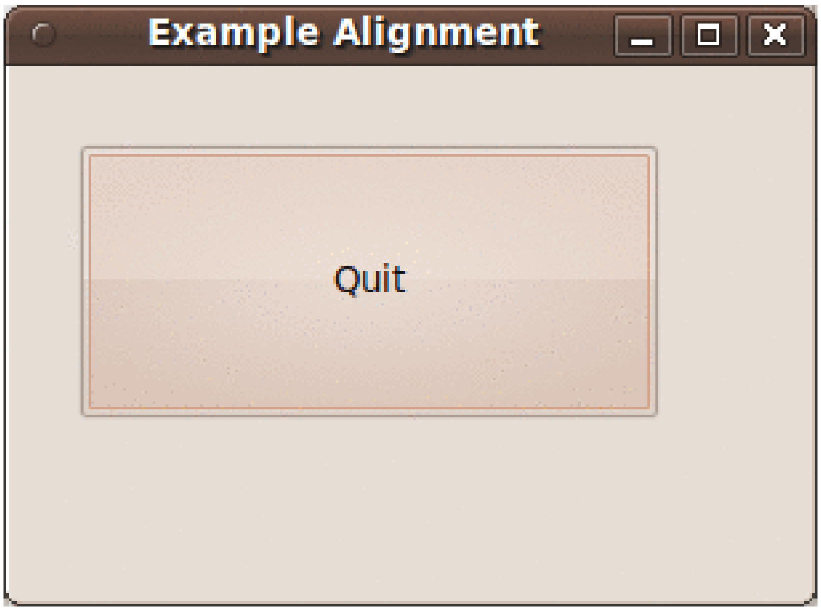
At first glance, the EventBox widget might appear to be totally useless. It draws nothing on the screen and responds to no events. However, it does serve a function - it provides an X window for its child widget. This is important as many GTK widgets do not have an associated X window. Not having an 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 EventBox emphasizes the event-handling function, the widget can also be used for clipping. (and more, see the example below).

To create a new EventBox widget, use e.g. `(make-instance 'gtk-event-box)` or the function `gtk-event-box-new`. A child widget can then be added to this EventBox with this function `gtk-container-add`.

The following example demonstrates both uses of an EventBox - 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))))
```

7.2 The Alignment widget



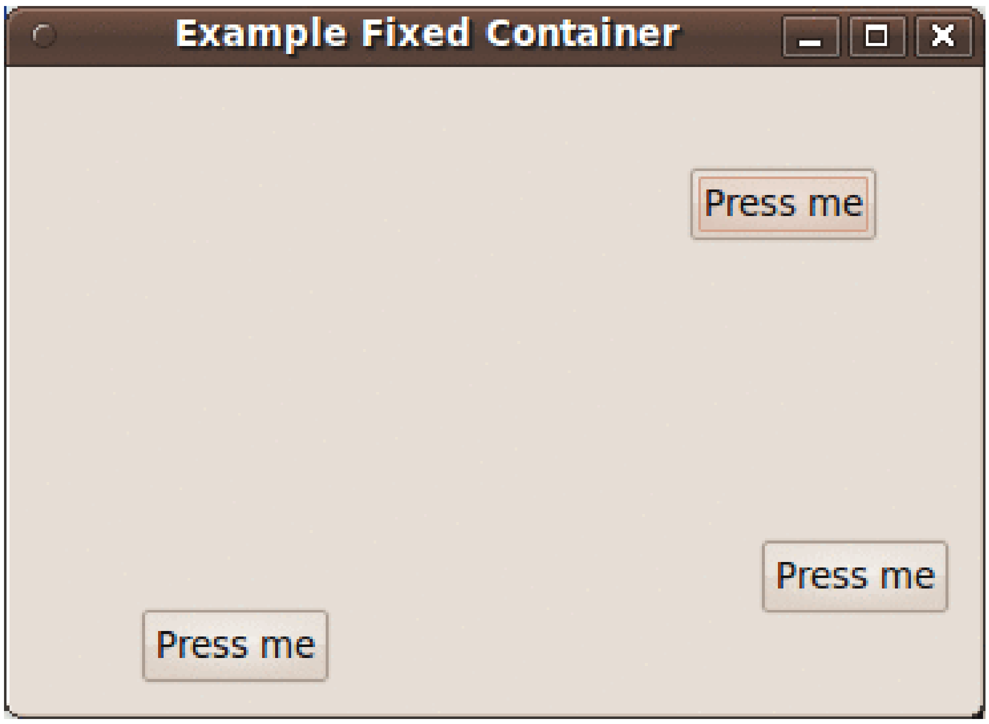
The alignment widget allows to place a widget within its window at a position and size relative to the size of the Alignment widget itself. For example, it can be very useful for centering a widget within the window.

There are the functions `gtk-alignment-new` and `gtk-alignment-set` associated with the Alignment widget. The first function creates a new Alignment widget with the specified parameters. The second function allows the alignment parameters of an exisiting Alignment 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 Alignment widget. The xscale and yscale arguments affect the amount of space allocated to the widget. A child widget can be added to this Alignment 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))))
```

7.3 Fixed Container



The Fixed container allows to place widgets at a fixed position within it's window, relative to it's upper left hand corner. The position of the widgets can be changed dynamically. There are only a few functions associated with the fixed widget like `gtk-fixed-new`, `gtk-fixed-put`, and `gtk-fixed-move`.

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

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

The following example illustrates how to use the 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))))
```

7.4 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))))
```

7.5 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

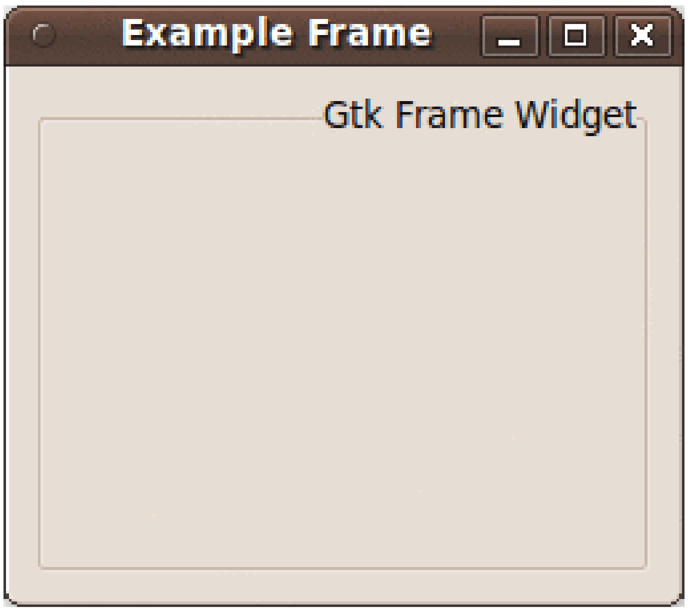
Nil for the label argument will result in no label being displayed. The text of the label can be changed using the function `gtk-frame-set-label`.

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
```

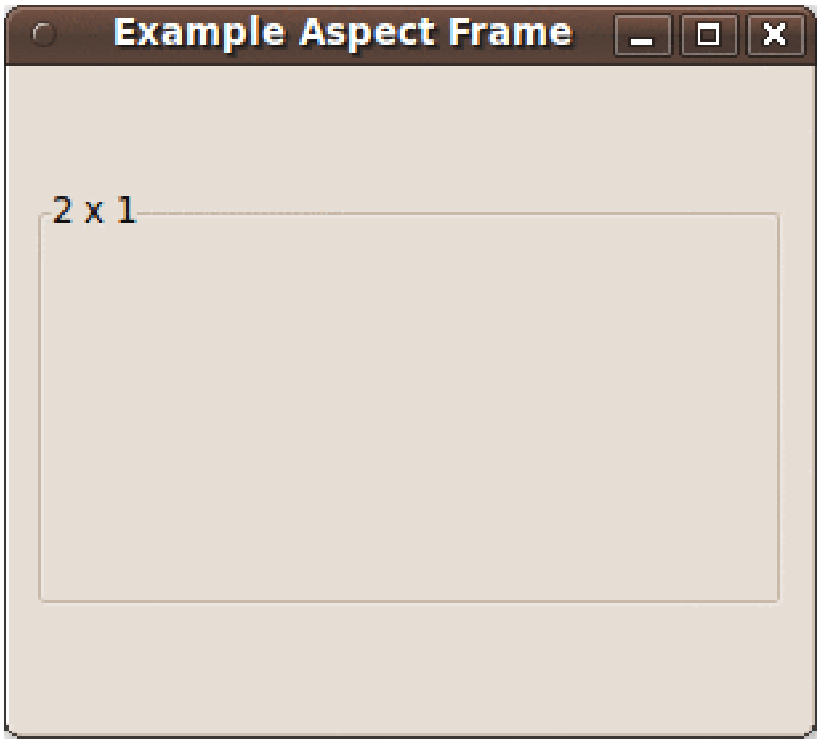
The following code example illustrates the use of the Frame widget.



```
(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))))
```

7.6 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))))
```

8 Menu Widget