

# GTK+ / GNOME

## Group 2

Ma. Angela Arellano  
Joshua Jeremy Gonzalez  
Christian Niño Jacinto  
Jose Antonio Miguel Mella  
Yvette Lourdes Ong  
Ana Janina Pangilinan

**CS 159.3 : Systems Programming**  
Mr. William Yu

## Gnome and GTK+

One might argue that Red Hat Linux, one of the more popular Linux distributions, was wildly successful in getting new users interested in the Linux and Unix operating systems due to its user-friendly interface. After all, a whole new generation of computer users today haven't even seen a command line, and they're unlikely to abandon their lush, graphical point-and-click world for esoteric commands and relentless lines of text. So Red Hat packaged two desktop environments ("graphical interfaces", to put it loosely) with its distributions to make Linux a little less daunting: GNOME and KDE. GNOME, being the default interface and having official backing from Red Hat, became widely used.

## What is GNOME?

GNOME is pronounced "guh-nome", and has to do more with computers than short men wearing pointy hats. GNOME is an acronym for GNU Network Object Model Environment, the sort of phrase that makes you glad acronyms were invented. To put it simply, GNOME is an attempt to make a graphical desktop environment, a visual way to present the Unix operating system.

We say "attempt" because GNOME is constantly under development. Being a part of the GNU project ([www.gnu.org](http://www.gnu.org)), GNOME is free not only to be used but also to be improved upon by anybody who wants to. What started out as a project by a handful of people is now being coded by hundreds of people from all over the world.

## What about GTK+?

Under the hood of GNOME is GTK+, which you can pronounce "guh-tuh-kuh plus" if you want to be a very silly person. It's a toolkit (or "widget set") for creating graphical user interfaces for the X Windows System, much like Swing is used to provide a graphical interface for Java programs. GTK stands for "GIMP toolkit", since it was originally written to develop the GNU Image Manipulation Program (GIMP, originally "General Image Manipulation Program"). Now it's used to develop lots of other applications, and of course, GNOME.

GTK+ is composed of three widget libraries. Widgets are abstract user interface objects used to create concrete instances of UI objects. There are widgets for buttons, for text, for menus, etc. Just like Java's packages, they're also arranged in a hierarchical structure.

However, the user interface widgets are part of only one of the three libraries composing GTK+, that is, GTK. The other two are GLib, and GDK. GLib provides many useful datatypes, such as hashtables and arrays, macros, type conversions, string utilities, and even a lexical scanner. GDK contains the wrappers for low-level windowing functions, such as shape drawing, font manipulation, and window event handling.

Like most software having an acronym that begins with a G, GTK+ is also part of the GNU project.

## History

In the beginning, Spencer Kimball and Peter Mattis developed the General Image Manipulation Program, or GIMP, as a Computer Science project. On January 1996, they released to the public version 0.54; users and developers alike loved it.

GIMP was then dependent on the Motif GUI toolkit, which the developers found severely limiting, codewise and licensewise, so for version 0.60 of GIMP released later in the year, they wrote their own GUI toolkit, the GIMP toolkit (GTK). The public loved it too, and what was intended for creating an image editor eventually became a general purpose GUI toolkit.

February 1997 saw the release of GIMP version 0.99, and with it, GTK+, an overhauled version of GTK which incorporated object-oriented concepts and a signal mechanism. Minor updates and releases for both GIMP and GTK+ followed until June 1997. That's when the developers graduated from school, and the public took over. Today, GTK+ is at version 1.2 with 2.0 on the works.

As for GNOME, it was originally announced on August 1997 by Miguel de Icaza, one of the founders of the GNOME project. He wanted to write a free desktop environment, an alternative to KDE, because even though KDE was under the GNU license, it was dependent on Qt, which wasn't free software then (Qt is now free).

GNOME 0.10 was finished on December that year, and at that time, thanks to the GNU license, more developers have joined the GNOME team. Also on that month, Red Hat officially supported the GNOME project, and on January the next year, created the Red Hat Advanced Development Labs, which funded a team of developers to work on the GNOME project.

On September 1998, the single GNOME tarball was split into 4 for version 0.30: the gnome core, libraries, utilities, and media. By November, the core was already at version 0.99, and code was frozen to fix things up for the official 1.0 release. Finally, on March 1999, it was released at the Linux World Expo. It was also very buggy. It wasn't until October that they released version 1.0.55, which fixed a lot of bugs that users reported. This stable release was shipped with various Linux distributions.

At this time, companies started forming based on GNOME, such as Eazel and Helix Code, and they worked on the infrastructural parts of GNOME and also on the various visual components of the GNOME desktop.

In fact, version 1.2, released on May 2000, was mostly user interface improvements. The developers nicknamed it "Bongo GNOME". Don't ask.

As Linux's popularity increased, big name companies finally took notice on little GNOME. On August 2000, the GNOME Foundation was announced, with various companies joining to develop GNOME, including Sun Microsystems, Hewlett-Packard, and IBM. Sun, being such a nice company, set up the GNOME Accessibility Lab the next month, which would add improvements to GNOME to make it usable by people with disabilities.

GNOME 2.0 has been under development for some time now, and if things went according to schedule, should be released by the end of March 2002.

## Programs with GNOME and GTK+

You can't really have programs that contain GNOME, since GNOME is the desktop environment where other programs will be running on. What you would have would be the various distributions and platforms that support GNOME. GNOME runs on any Unix-like platform that supports X and has GTK+ installed on it.

According to the GNOME FAQ, GNOME runs on recent distributions of Linux. It also runs on FreeBSD, NetBSD, Solaris, OpenBSD, IRIX, HP-UX, and AIX. It however does not run on Windows, BeOS, and Mac OS, unless somebody decides to port it.

Now, which programs are made by GTK+? Hundreds. Perhaps thousands. And most of them are free. To name a few, there's *GIMP*, the reason why GTK+ exists in the first place, *Abiword*, a word processing program, *Glade*, a graphical user interface builder, *Dia*, a diagram creator, *Gnucash*, a set of accounting and financial tools, and *Gnumeric*, a spreadsheet program. There are a lot more programs created using GTK+, most of them packaged with GNOME, which is also created using GTK+.

For an extensive list of applications, visit [www.gnome.org/applist/](http://www.gnome.org/applist/) and [www.gtk.org/apps/](http://www.gtk.org/apps/).

## Key Concepts in Building GTK+ Applications

To create GTK+ applications, you'll have to use widgets which make up the toolkit.

There are various ways to build GTK+ applications, and one of the easiest is to use Glade, a visual user interface builder, comparable to Microsoft's Visual Studio. Simply choose the components to add to your interface, and drag and drop to arrange them. Behind the scenes, Glade does all the dirty work and generates the C source code for your interface. Should you prefer it in another language such as C++ or Python, external tools are available to process the XML interface description files by Glade.

Before resorting to shortcuts, however, it's best to learn how to code GTK+ applications by hand. As mentioned above, GTK+ applications are written in C. If you're unfamiliar with that, there are bindings to other programming languages, including C++. To use GTK+, you have to include `gtk/gtk.h`, that is `"#include <gtk/gtk.h>"` to declare all that needs to be used by the toolkit. Also, all GTK+ apps need the line `"gtk_init (&argc, &argv)"` which sets up the applications, loading default visual and color maps, signal handlers, and also checks command line arguments passed to the app.

Applications should also have `"gtk_main ()"`, usually found after the user interface has been set up. It puts GTK to sleep, and waits for events (usually user driven) to occur; for example, a button press.

## An Example Window

Creating components should be relatively familiar to those who've taken up object-oriented programming such as Java or C++. To create an object, simply declare the object type (GUI components are usually `"GtkWidget"`), followed by a pointer to that object. Then, use the appropriate method to specify and initialize which GUI component it is. For example, to create a window, we type:

```
GtkWidget *window;  
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

`gtk_window_new` is a method that creates a new window, accepting window type as an argument. To make it visible, we type:

```
gtk_widget_show (window)
```

which makes the component visible. We use this method for all components we want to show.

Here are a few more useful methods for window creation:

To set the window title:

```
void gtk_window_set_title (GtkWindow *window, const gchar *title);+
```

To set the size:

```
void gtk_window_set_default_size (GtkWindow *window, gint width, gint height);+
```

To set its position:

```
void gtk_window_set_position (GtkWindow *window, GtkWindowPosition position);
```

Of course, your window would be pretty boring if there was nothing in it. Here's a method for adding other widgets into your window:

```
void gtk_container_add (GtkContainer *container, GtkWidget *widget);
```

Notice that it takes a `GtkContainer` as an argument, but our window is a, well, `GtkWindow`. To remedy that, you can type cast the window to a container, with this macro: `GTK_CONTAINER ()`, so an example would be:

```
gtk_container_add (GTK_CONTAINER (window), button);
```

That one adds a button to the window.

Here's is a sample code that shows button widgets:

```
/* example-start buttons buttons.c */
#include <gtk/gtk.h>

/* Create a new hbox with an image and a label packed into it
 * and return the box. */

GtkWidget *xpm_label_box( GtkWidget *parent,
                          gchar   *xpm_filename,
                          gchar   *label_text )
{
    GtkWidget *box1;
    GtkWidget *label;
    GtkWidget *pixmapwid;
    GdkPixmap *pixmap;
    GdkBitmap *mask;
    GtkStyle *style;

    /* Create box for xpm and label */
    box1 = gtk_hbox_new (FALSE, 0);
```

---

<sup>+</sup> `gint` and `gchar` are simply typedefs of integers and characters respectively.



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

/* Create a new button */
button = gtk_button_new ();

/* Connect the "clicked" signal of the button to our callback */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (callback), (gpointer) "cool
button");

/* This calls our box creating function */
box1 = xpm_label_box(window, "info.xpm", "cool button");

/* Pack and show all our widgets */
gtk_widget_show(box1);

gtk_container_add (GTK_CONTAINER (button), box1);

gtk_widget_show(button);

gtk_container_add (GTK_CONTAINER (window), button);

gtk_widget_show (window);

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

return(0);
}
/* example-end */
```

The same compiler that compiles non-GTK+ C programs can also handle the above GTK+ source code, if you add a few options:

```
gcc -Wall -g gtkcode.c -o gtkcode `gtk-config --cflags --libs`
```

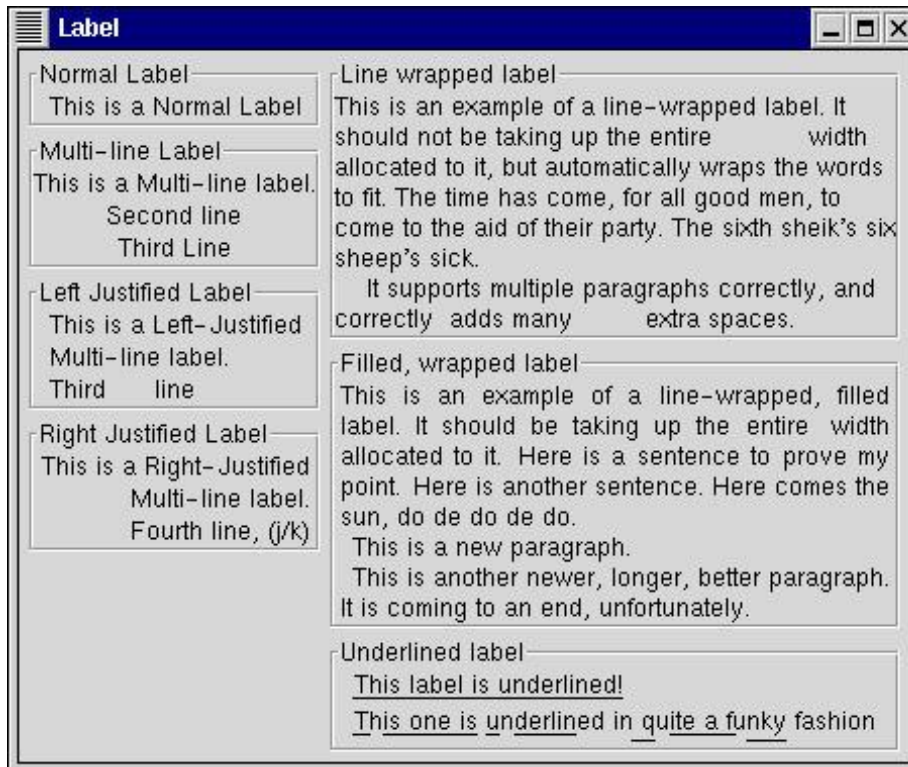
gtk-config is a program included with GTK that sets up the compiler switches needed to compile GTK+ programs. The `--cflags` option causes it to output a list of include directories for the compiler to look in, and the `--libs` outputs a list of libraries for the compiler to link with. Take note, the type of single quotes used is important here, since it has special meaning to the shell.

## Other Widgets

Here are other widgets which you might add to a window, along with some of their methods. The method names pretty much explain themselves.

*GtkLabel* - used for text labels:

```
GtkWidget* gtk_label_new (const gchar *str);
void gtk_label_set_justify (GtkLabel *label, GtkJustification
                           jtype);
void gtk_label_set_line_wrap (GtkLabel *label, gboolean wrap);
void gtk_label_set_text (GtkLabel *label, const gchar *str);
```



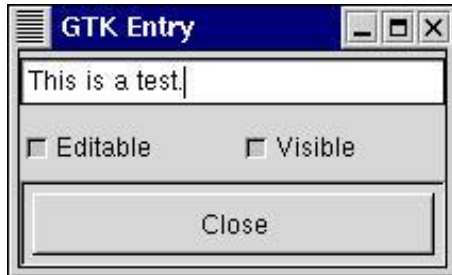
*GtkButton* - it's a button! :

```
GtkWidget* gtk_button_new (void);
GtkWidget* gtk_button_new_with_label (const gchar *label);
```



*GtkEntry* - a single line text field:

```
GtkWidget* gtk_entry_new (void);
GtkWidget* gtk_entry_new_with_max_length (guint16 max);
void gtk_entry_set_text (GtkEntry *entry, const gchar *text);
gchar* gtk_entry_get_text (GtkEntry *entry);
```



*GtkImage* - displays a graphical image:

```
GtkWidget* gtk_image_new (GdkImage *val, GdkBitmap *mask);
```

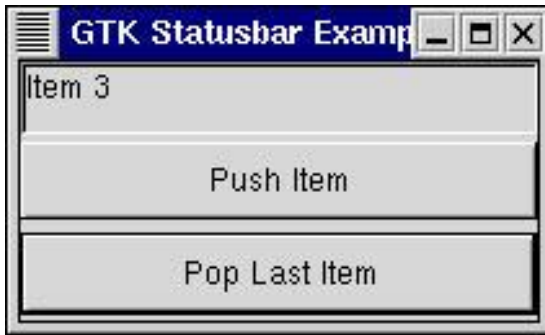
*GtkProgressBar* -- a widget which indicates progress visually.

```
GtkWidget* gtk_progress_bar_new (void);
void gtk_progress_bar_pulse (GtkProgressBar *pbar);
```



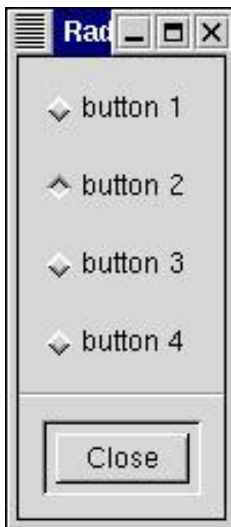
*GtkStatusbar* - report messages of minor importance to the user.

```
GtkWidget* gtk_statusbar_new (void);
guint gtk_statusbar_get_context_id (GtkStatusbar *statusbar,
const gchar *context_description);
guint gtk_statusbar_push (GtkStatusbar *statusbar, guint context_id,
const gchar *text);
void gtk_statusbar_pop (GtkStatusbar *statusbar, guint context_id);
void gtk_statusbar_remove (GtkStatusbar *statusbar, guint context_id,
guint message_id);
```



*GtkRadioButton* - for a group of radio buttons

```
GtkWidget* gtk_radio_button_new (GSLList *group);
GSLList* gtk_radio_button_group (GtkRadioButton *radio_button);
void gtk_radio_button_set_group (GtkRadioButton *radio_button, GSLList *group);
```



*GtkCheckButton* - create widgets with a discrete toggle button.

```
GtkWidget* gtk_check_button_new (void);
GtkWidget* gtk_check_button_new_with_label (const gchar *label);
GtkWidget* gtk_check_button_new_with_mnemonic (const gchar *label);
```

*GtkToggleButton* - create buttons which retain their state.

```
GtkWidget* gtk_toggle_button_new (void);
GtkWidget* gtk_toggle_button_new_with_label (const gchar *label);
GtkWidget* gtk_toggle_button_new_with_mnemonic (const gchar *label);
gboolean gtk_toggle_button_get_mode (GtkToggleButton *toggle_button);
void gtk_toggle_button_toggled (GtkToggleButton *toggle_button);
gboolean gtk_toggle_button_get_inconsistent (GtkToggleButton *toggle_button);
```

*GtkMenuShell* - an abstract base class to derive *GtkMenu* and *GtkMenuBar* subclasses

The *GtkMenuShell* is a container that holds *GtkMenuItem* widgets

### *GtkMenuBar* - a standard menu bar

```

GtkWidget* gtk_menu_bar_new (void);
void gtk_menu_bar_append (GtkMenuBar *menu_bar, GtkWidget *child);
void gtk_menu_bar_prepend (GtkMenuBar *menu_bar, GtkWidget *child);
void gtk_menu_bar_insert (GtkMenuBar *menu_bar, GtkWidget *child, gint
                          position);

```

### *GtkMenu* - a widget for drop-down menus

```

GtkWidget* gtk_menu_new (void);

void gtk_menu_append (GtkMenu *menu, GtkWidget *child);
void gtk_menu_prepend (GtkMenu *menu, GtkWidget *child);
void gtk_menu_insert (GtkMenu *menu, GtkWidget *child, gint position);
void gtk_menu_popup (GtkMenu *menu, GtkWidget *parent_menu_shell,
                    GtkWidget *parent_menu_item, GtkMenuPositionFunc
                    func, gpointer data, guint button, guint32 activate_time);

```



### *GtkMenuItem* - the components of any menu

```

GtkWidget* gtk_menu_item_new (void);
GtkWidget* gtk_menu_item_new_with_label (const gchar *label);
void gtk_menu_item_set_submenu (GtkMenuItem *menu_item, GtkWidget
                                *submenu);
void gtk_menu_item_remove_submenu (GtkMenuItem *menu_item);
void gtk_menu_item_select (GtkMenuItem *menu_item);
void gtk_menu_item_deselect (GtkMenuItem *menu_item);
void gtk_menu_item_activate (GtkMenuItem *menu_item);

```

### *GtkCheckMenuItem* - menu items with checkboxes

```

GtkWidget* gtk_check_menu_item_new (void);
GtkWidget* gtk_check_menu_item_new_with_label (const gchar *label);
void gtk_check_menu_item_toggled (GtkCheckMenuItem *check_menu_item);

```

### *GtkCombo* - a text entry field with a dropdown list

```

GtkWidget* gtk_combo_new (void);
void gtk_combo_set_popdown_strings (GtkCombo *combo, GList *strings);
void gtk_combo_set_value_in_list (GtkCombo *combo, gboolean val,
                                  gboolean ok_if_empty);
void gtk_combo_set_use_arrows (GtkCombo *combo, gboolean val);

```

These commonly used components are merely a small part of all the available widgets for GTK+. For an exhaustive list of widgets and detailed function explanations, you may view the API documentation, available for download at [www.gtk.org/api/](http://www.gtk.org/api/).

Just remember to add your widgets to a container, and don't forget show them. It would help if you show your container last, that way, all your widgets will be shown at the same time, instead of one after the other.

Of course, randomly packing all those widgets into a container will result in a very messy user interface. Not only that, you'll notice that nothing happens when you click on your user interface. To help with your problems, GTK+ provides layout widgets to organize your GUI, and event handlers to make them interactive.

## Basic Layouts

Of course, randomly packing all those widgets into a container will result in a very messy user interface. Good thing GTK+ provides ways to layout all those widgets. Here are some container functions found in GTK+:

### *Fixed Container*

The Fixed container allows you to place widgets at a fixed position within its window, relative to its upper left hand corner. The position of the widgets can be changed dynamically.

There are only three functions associated with the fixed widget:

```
GtkWidget* gtk_fixed_new( void );

void gtk_fixed_put( GtkFixed *fixed,
                  GtkWidget *widget,
                  gint16  x,
                  gint16  y );

void gtk_fixed_move( GtkFixed *fixed,
                   GtkWidget *widget,
                   gint16  x,
                   gint16  y );
```

The function `gtk_fixed_new` allows you to create a new Fixed container, while `gtk_fixed_put` places 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.

The following example illustrates how to use the Fixed Container.

```
/* example-start fixed fixed.c */
```

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

gint x=50;
gint y=50;

/* This callback function moves the button to a new position
 * in the Fixed container. */
void move_button( GtkWidget *widget,
                  GtkWidget *fixed )
{
    x = (x+30)%300;
    y = (y+50)%300;
    gtk_fixed_move( GTK_FIXED(fixed), widget, x, y);
}

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

    /* Initialise GTK */
    gtk_init(&argc, &argv);

    /* Create a new window */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Fixed Container");

    /* Here we connect the "destroy" event to a signal handler */
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                       GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

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

    /* Create a Fixed Container */
    fixed = gtk_fixed_new();
    gtk_container_add(GTK_CONTAINER(window), fixed);
    gtk_widget_show(fixed);

    for (i = 1 ; i <= 3 ; i++) {
        /* Creates a new button with the label "Press me" */
        button = gtk_button_new_with_label ("Press me");

        /* When the button receives the "clicked" signal, it will call the
         * function move_button() passing it the Fixed Container as its
         * argument. */
        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (move_button), fixed);

        /* This packs the button into the fixed containers window. */
        gtk_fixed_put (GTK_FIXED (fixed), button, i*50, i*50);
    }
}
```

```

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

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

/* Enter the event loop */
gtk_main ();

return(0);
}
/* example-end */

```

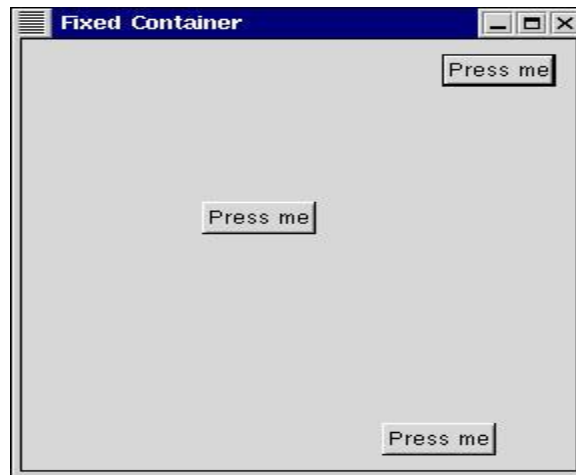


Fig 1.1 Example Screenshot

## 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:

```

GtkWidget *gtk_layout_new( GtkAdjustment *hadjustment,
                           GtkAdjustment *vadjustment );

```

Where you can optionally specify the Adjustment objects that the Layout widget will use for its scrolling.

You can also add and move widgets in the Layout container using the following two functions:

```

void gtk_layout_put( GtkLayout *layout,

```



## Frames

Frames can be used to enclose one or a group of widgets with a box which can optionally be labeled. The position of the label and the style of the box can be altered to suit. A Frame can be created with the following function:

```
GtkWidget *gtk_frame_new( const gchar *label );
```

The label is by default placed in the upper left hand corner of the frame. A value of NULL for the label argument will result in no label being displayed. The text of the label can be changed using the next function.

```
void gtk_frame_set_label( GtkFrame *frame,
                        const gchar *label );
```

The position of the label can be changed using this function:

```
void gtk_frame_set_label_align( GtkFrame *frame,
                              gfloat xalign,
                              gfloat yalign );
```

Where *xalign* and *yalign* take values between 0.0 and 1.0. *xalign* indicates the position of the label along the top horizontal of the frame, and *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 next function alters the style of the box that is used to outline the frame.

```
void gtk_frame_set_shadow_type( GtkFrame *frame,
                               GtkShadowType type);
```

The type argument can take one of the following values:

```
GTK_SHADOW_NONE
GTK_SHADOW_IN
GTK_SHADOW_OUT
GTK_SHADOW_ETCHED_IN (the default)
GTK_SHADOW_ETCHED_OUT
```

The following is an example code which illustrates the use of the Frame widget.

```
/* example-start frame frame.c */

#include <gtk/gtk.h>

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

    /* Initialise GTK */
    gtk_init(&argc, &argv);
```

```
/* Create a new window */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "Frame Example");

/* Here we connect the "destroy" event to a signal handler */
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                   GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

gtk_widget_set_usize(window, 300, 300);
/* Sets the border width of the window. */
gtk_container_set_border_width (GTK_CONTAINER (window), 10);

/* Create a Frame */
frame = gtk_frame_new(NULL);
gtk_container_add(GTK_CONTAINER(window), frame);

/* Set the frame's label */
gtk_frame_set_label( GTK_FRAME(frame), "GTK Frame Widget" );

/* Align the label at the right of the frame */
gtk_frame_set_label_align( GTK_FRAME(frame), 1.0, 0.0);

/* Set the style of the frame */
gtk_frame_set_shadow_type(           GTK_FRAME(frame),
                                   GTK_SHADOW_ETCHED_OUT);

gtk_widget_show(frame);

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

/* Enter the event loop */
gtk_main ();

return(0);
}/* example-end */
```

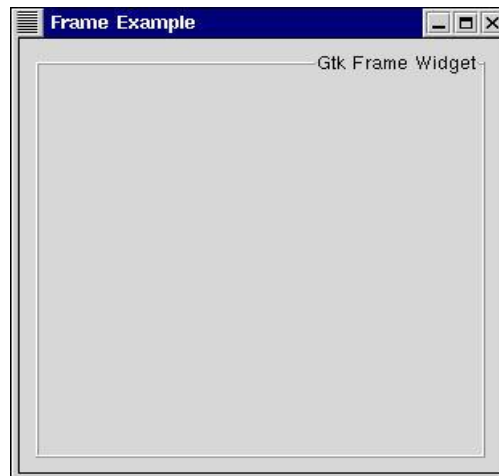


Fig 3.1 Example Screenshot

## Paned Window Widgets

The paned window widgets are useful when you want to divide an area into two parts, with the relative size of the two parts controlled by the user. A groove is drawn between the two portions with a handle that the user can drag to change the ratio. The division can either be horizontal (HPaned) or vertical (VPaned).

To create a new paned window, call one of:

```
GtkWidget *gtk_hpaned_new (void);
```

```
GtkWidget *gtk_vpaned_new (void);
```

After creating the paned window widget, you need to add child widgets to its two halves. To do this, use the functions:

```
void gtk_paned_add1 (GtkPaned *paned, GtkWidget *child);
```

```
void gtk_paned_add2 (GtkPaned *paned, GtkWidget *child);
```

`gtk_paned_add1()` adds the child widget to the left or top half of the paned window.

`gtk_paned_add2()` adds the child widget to the right or bottom half of the paned window.

As an example, we will create part of the user interface of an imaginary email program. A window is divided into two portions vertically, with the top portion being a list of email messages and the bottom portion the text of the email message. Most of the program is pretty straightforward. A couple of points to note: text can't be added to a Text widget until it is realized. This could be done by calling `gtk_widget_realize()`, but as a demonstration of an alternate technique, we connect a handler to the "realize" signal to add the text. Also, we need to add the `GTK_SHRINK` option to some of the items in the table containing the text window and its scrollbars, so that when the bottom portion is made smaller, the correct portions shrink instead of being pushed off the bottom of the window.

```
/* example-start paned paned.c */

#define GTK_ENABLE_BROKEN
#include <stdio.h>
#include <gtk/gtk.h>

/* Create the list of "messages" */
GtkWidget *create_list( void )
{

    GtkWidget *scrolled_window;
    GtkWidget *list;
    GtkWidget *list_item;

    int i;
```

```

char buffer[16];

/* Create a new scrolled window, with scrollbars only if needed */
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
                               (scrolled_window),
                               GTK_POLICY_AUTOMATIC,
                               GTK_POLICY_AUTOMATIC);

/* Create a new list and put it in the scrolled window */
list = gtk_list_new ();
gtk_scrolled_window_add_with_viewport (
    GTK_SCROLLED_WINDOW (scrolled_window), list);
gtk_widget_show (list);

/* Add some messages to the window */
for (i=0; i<10; i++) {

    sprintf(buffer, "Message #%d", i);
    list_item = gtk_list_item_new_with_label (buffer);
    gtk_container_add (GTK_CONTAINER(list), list_item);
    gtk_widget_show (list_item);

}

return scrolled_window;
}

/* Add some text to our text widget - this is a callback that is invoked
when our window is realized. We could also force our window to be
realized with gtk_widget_realize, but it would have to be part of
a hierarchy first */

void realize_text( GtkWidget *text,
                 gpointer data )
{
    gtk_text_freeze (GTK_TEXT (text));
    gtk_text_insert (GTK_TEXT (text), NULL, &text->style->black, NULL,
        "From: pathfinder@nasa.gov\n"
        "To: mom@nasa.gov\n"
        "Subject: Made it!\n"
        "\n"
        "We just got in this morning. The weather has been\n"
        "great - clear but cold, and there are lots of fun sights.\n"
        "Sojourner says hi. See you soon.\n"
        "-Path\n", -1);

    gtk_text_thaw (GTK_TEXT (text));
}

/* Create a scrolled text area that displays a "message" */
GtkWidget *create_text( void )
{
    GtkWidget *table;
    GtkWidget *text;

```

```

GtkWidget *hscrollbar;
GtkWidget *vscrollbar;

/* Create a table to hold the text widget and scrollbars */
table = gtk_table_new (2, 2, FALSE);

/* Put a text widget in the upper left hand corner. Note the use of
 * GTK_SHRINK in the y direction */
text = gtk_text_new (NULL, NULL);
gtk_table_attach (GTK_TABLE (table), text, 0, 1, 0, 1,
                 GTK_FILL | GTK_EXPAND,
                 GTK_FILL | GTK_EXPAND | GTK_SHRINK, 0, 0);
gtk_widget_show (text);

/* Put a HScrollbar in the lower left hand corner */
hscrollbar = gtk_hscrollbar_new (GTK_TEXT (text)->hadj);
gtk_table_attach (GTK_TABLE (table), hscrollbar, 0, 1, 1, 2,
                 GTK_EXPAND | GTK_FILL, GTK_FILL, 0, 0);
gtk_widget_show (hscrollbar);

/* And a VScrollbar in the upper right */
vscrollbar = gtk_vscrollbar_new (GTK_TEXT (text)->vadj);
gtk_table_attach (GTK_TABLE (table), vscrollbar, 1, 2, 0, 1,
                 GTK_FILL, GTK_EXPAND | GTK_FILL | GTK_SHRINK, 0, 0);
gtk_widget_show (vscrollbar);

/* Add a handler to put a message in the text widget when it is realized */
gtk_signal_connect (GTK_OBJECT (text), "realize",
                  GTK_SIGNAL_FUNC (realize_text), NULL);

return table;
}

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

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Paned Windows");
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    gtk_widget_set_usize (GTK_WIDGET(window), 450, 400);

    /* create a vpaned widget and add it to our toplevel window */

    vpaned = gtk_vpaned_new ();
    gtk_container_add (GTK_CONTAINER(window), vpaned);
    gtk_widget_show (vpaned);

```

```
/* Now create the contents of the two halves of the window */

list = create_list ();
gtk_paned_add1 (GTK_PANED(vpaned), list);
gtk_widget_show (list);

text = create_text ();
gtk_paned_add2 (GTK_PANED(vpaned), text);
gtk_widget_show (text);
gtk_widget_show (window);
gtk_main ();
return 0;
}
/* example-end */
```



Fig 4.1 Example Screenshot

## Scrolled Windows

Scrolled windows are used to create a scrollable area with another widget inside it. You may insert any type of widget into a scrolled window, and it will be accessible regardless of the size by using the scrollbars.

The following function is used to create a new scrolled window.

```
GtkWidget *gtk_scrolled_window_new( GtkAdjustment *hadjustment,
                                   GtkAdjustment *vadjustment );
```

Where the first argument is the adjustment for the horizontal direction, and the second, the adjustment for the vertical direction. These are almost always set to NULL.

```
void gtk_scrolled_window_set_policy(GtkScrolledWindow
                                   *scrolled_window, GtkPolicyType
                                   hscrollbar_policy, GtkPolicyType
                                   vscrollbar_policy );
```

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

The policy may be one of `GTK_POLICY_AUTOMATIC` or `GTK_POLICY_ALWAYS`. `GTK_POLICY_AUTOMATIC` will automatically decide whether you need scrollbars, whereas `GTK_POLICY_ALWAYS` will always leave the scrollbars there.

You can then place your object into the scrolled window using the following function.

```
void gtk_scrolled_window_add_with_viewport( GtkScrolledWindow *scrolled_window,
                                           GtkWidget *child);
```

Here is a simple example that packs a table with 100 toggle buttons into a scrolled window.

```
/* example-start scrolledwin scrolledwin.c */
```

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

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

int main( int argc,
         char *argv[] )
{
    static GtkWidget *window;
    GtkWidget *scrolled_window;
    GtkWidget *table;
```

```

GtkWidget *button;
char buffer[32];
int i, j;

gtk_init (&argc, &argv);

/* Create a new dialog window for the scrolled window to be
 * packed into. */
window = gtk_dialog_new ();
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                   (GtkSignalFunc) destroy, NULL);
gtk_window_set_title (GTK_WINDOW (window), "GtkScrolledWindow example");
gtk_container_set_border_width (GTK_CONTAINER (window), 0);
gtk_widget_set_usize(window, 300, 300);

/* create a new scrolled window. */
scrolled_window = gtk_scrolled_window_new (NULL, NULL);

gtk_container_set_border_width (GTK_CONTAINER (scrolled_window), 10);

/* the policy is one of GTK_POLICY_AUTOMATIC, or GTK_POLICY_ALWAYS.
 * GTK_POLICY_AUTOMATIC will automatically decide whether you need
 * scrollbars, whereas GTK_POLICY_ALWAYS will always leave the scrollbars
 * there. The first one is the horizontal scrollbar, the second,
 * the vertical. */
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                               GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
/* The dialog window is created with a vbox packed into it. */
gtk_box_pack_start (GTK_BOX (GTK_DIALOG(window)->vbox), scrolled_window,
                   TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);

/* create a table of 10 by 10 squares. */
table = gtk_table_new (10, 10, FALSE);

/* set the spacing to 10 on x and 10 on y */
gtk_table_set_row_spacings (GTK_TABLE (table), 10);
gtk_table_set_col_spacings (GTK_TABLE (table), 10);

/* pack the table into the scrolled window */
gtk_scrolled_window_add_with_viewport (
    GTK_SCROLLED_WINDOW (scrolled_window), table);
gtk_widget_show (table);

/* this simply creates a grid of toggle buttons on the table
 * to demonstrate the scrolled window. */
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++) {
        sprintf (buffer, "button (%d,%d)\n", i, j);
        button = gtk_toggle_button_new_with_label (buffer);
        gtk_table_attach_defaults (GTK_TABLE (table), button,
                                   i, i+1, j, j+1);
        gtk_widget_show (button);
    }

```

```
/* Add a "close" button to the bottom of the dialog */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                          (GtkSignalFunc) gtk_widget_destroy,
                          GTK_OBJECT (window));

/* this makes it so the button is the default. */

GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area), button,
TRUE, TRUE, 0);

/* This grabs this button to be the default button. Simply hitting
 * the "Enter" key will cause this button to activate. */
gtk_widget_grab_default (button);
gtk_widget_show (button);

gtk_widget_show (window);

gtk_main();

return(0);
}
/* example-end */
```

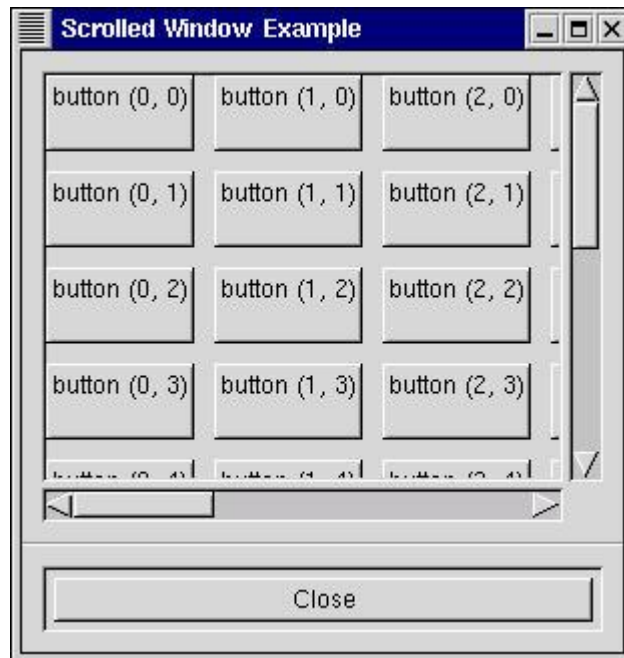


Fig 5.1 Example Screenshot

Try using the `gtk_widget_set_usize()` call to set the default size of the window or other widgets.

## Button Boxes

Button Boxes are a convenient way to quickly layout a group of buttons. They come in both horizontal and vertical flavours. You create a new Button Box with one of the following calls, which create a horizontal or vertical box, respectively:

```
GtkWidget *gtk_hbutton_box_new( void );
```

```
GtkWidget *gtk_vbutton_box_new( void );
```

The only attributes pertaining to button boxes effect how the buttons are laid out. You can change the spacing between the buttons with:

```
void gtk_hbutton_box_set_spacing_default( gint spacing );
```

```
void gtk_vbutton_box_set_spacing_default( gint spacing );
```

Similarly, the current spacing values can be queried using:

```
gint gtk_hbutton_box_get_spacing_default( void );
```

```
gint gtk_vbutton_box_get_spacing_default( void );
```

The second attribute that we can access effects the layout of the buttons within the box. It is set using one of:

```
void gtk_hbutton_box_set_layout_default( GtkButtonBoxStyle  
layout );
```

```
void gtk_vbutton_box_set_layout_default( GtkButtonBoxStyle  
layout );
```

The layout argument can take one of the following values:

```
GTK_BUTTONBOX_DEFAULT_STYLE  
GTK_BUTTONBOX_SPREAD  
GTK_BUTTONBOX_EDGE  
GTK_BUTTONBOX_START  
GTK_BUTTONBOX_END
```

The current layout setting can be retrieved using:

```
GtkButtonBoxStyle gtk_hbutton_box_get_layout_default( void );
```

```
GtkButtonBoxStyle gtk_vbutton_box_get_layout_default( void );
```

Buttons are added to a Button Box using the usual function:

```
gtk_container_add( GTK_CONTAINER(button_box), child_widget );
```

Here's an example that illustrates all the different layout settings for Button Boxes.

```
/* example-start buttonbox buttonbox.c */  
#include <gtk/gtk.h>
```

```

/* Create a Button Box with the specified parameters */
GtkWidget *create_bbox( gint horizontal,
                        char *title,
                        gint spacing,
                        gint child_w,
                        gint child_h,
                        gint layout )
{
    GtkWidget *frame;
    GtkWidget *bbox;
    GtkWidget *button;

    frame = gtk_frame_new (title);

    if (horizontal)
        bbox = gtk_hbutton_box_new ();
    else
        bbox = gtk_vbutton_box_new ();

    gtk_container_set_border_width (GTK_CONTAINER (bbox), 5);
    gtk_container_add (GTK_CONTAINER (frame), bbox);

    /* Set the appearance of the Button Box */
    gtk_button_box_set_layout (GTK_BUTTON_BOX (bbox), layout);
    gtk_button_box_set_spacing (GTK_BUTTON_BOX (bbox), spacing);
    gtk_button_box_set_child_size (GTK_BUTTON_BOX (bbox), child_w, child_h);

    button = gtk_button_new_with_label ("OK");
    gtk_container_add (GTK_CONTAINER (bbox), button);

    button = gtk_button_new_with_label ("Cancel");
    gtk_container_add (GTK_CONTAINER (bbox), button);

    button = gtk_button_new_with_label ("Help");
    gtk_container_add (GTK_CONTAINER (bbox), button);

    return(frame);
}

int main( int argc,
          char *argv[] )
{
    static GtkWidget* window = NULL;
    GtkWidget *main_vbox;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *frame_horz;
    GtkWidget *frame_vert;

    /* Initialize GTK */
    gtk_init( &argc, &argv );

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Button Boxes");

```

```
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                  GTK_SIGNAL_FUNC(gtk_main_quit),
                  NULL);

gtk_container_set_border_width (GTK_CONTAINER (window), 10);

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

frame_horz = gtk_frame_new ("Horizontal Button Boxes");
gtk_box_pack_start (GTK_BOX (main_vbox), frame_horz, TRUE, TRUE, 10);

vbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
gtk_container_add (GTK_CONTAINER (frame_horz), vbox);

gtk_box_pack_start (GTK_BOX (vbox),
                  create_bbox (TRUE, "Spread (spacing 40)", 40, 85, 20,
                              GTK_BUTTONBOX_SPREAD),
                  TRUE, TRUE, 0);

gtk_box_pack_start (GTK_BOX (vbox),
                  create_bbox (TRUE, "Edge (spacing 30)", 30, 85, 20,
                              GTK_BUTTONBOX_EDGE),
                  TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (vbox),
                  create_bbox (TRUE, "Start (spacing 20)", 20, 85, 20,
                              GTK_BUTTONBOX_START),
                  TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (vbox),
                  create_bbox (TRUE, "End (spacing 10)", 10, 85, 20,
                              GTK_BUTTONBOX_END),
                  TRUE, TRUE, 5);

frame_vert = gtk_frame_new ("Vertical Button Boxes");
gtk_box_pack_start (GTK_BOX (main_vbox), frame_vert, TRUE, TRUE, 10);

hbox = gtk_hbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (hbox), 10);
gtk_container_add (GTK_CONTAINER (frame_vert), hbox);

gtk_box_pack_start (GTK_BOX (hbox),
                  create_bbox (FALSE, "Spread (spacing 5)", 5, 85, 20,
                              GTK_BUTTONBOX_SPREAD),
                  TRUE, TRUE, 0);

gtk_box_pack_start (GTK_BOX (hbox),
                  create_bbox (FALSE, "Edge (spacing 30)", 30, 85, 20,
                              GTK_BUTTONBOX_EDGE),
                  TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (hbox),
                  create_bbox (FALSE, "Start (spacing 20)", 20, 85, 20,
                              GTK_BUTTONBOX_START),
                  TRUE, TRUE, 5);
```

```

gtk_box_pack_start(GTK_BOX(hbox),
    create_bbox(FALSE, "End (spacing 20)", 20, 85, 20, GTK_BUTTONBOX_END),
    TRUE, TRUE, 5);

gtk_widget_show_all(window);

/* Enter the event loop */
gtk_main();

return(0);
}
/* example-end*/

```

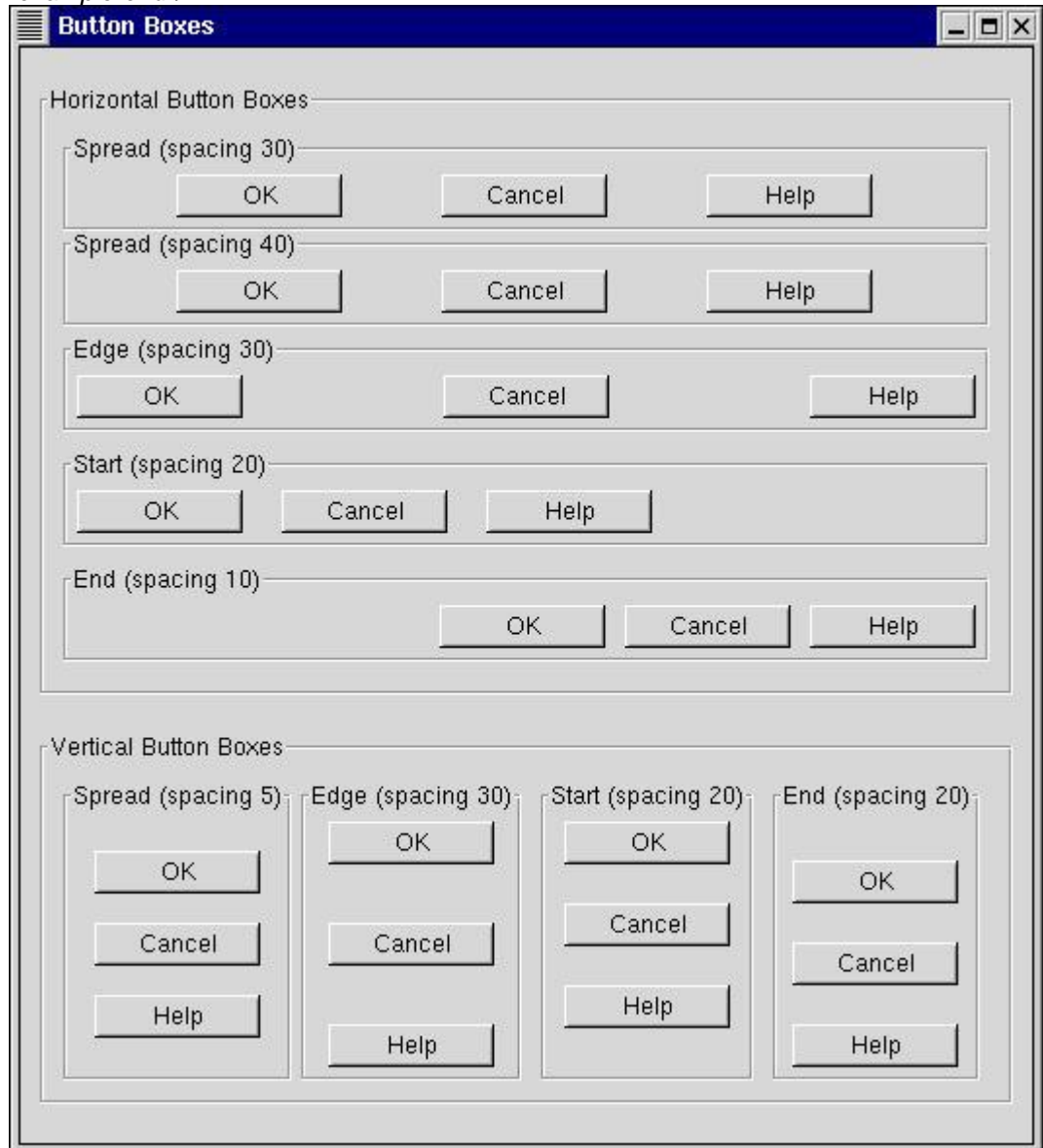


Fig. 6.1 Example Screenshot



```

const char *tooltip_private_text,
GtkWidget *icon,
GtkSignalFunc callback,
gpointer user_data,
gint position );

```

To simplify adding spaces between toolbar items, you may use the following functions:

```

void gtk_toolbar_append_space( GtkWidget *toolbar );

void gtk_toolbar_prepend_space( GtkWidget *toolbar );

void gtk_toolbar_insert_space( GtkWidget *toolbar,
                               gint position );

```

While the size of the added space can be set globally for a whole toolbar with the function:

```

void gtk_toolbar_set_space_size( GtkWidget *toolbar,
                                 gint space_size );

```

If it's required, the orientation of a toolbar and its style can be changed "on the fly" using the following functions:

```

void gtk_toolbar_set_orientation( GtkWidget *toolbar,
                                  GtkOrientation orientation );

void gtk_toolbar_set_style( GtkWidget *toolbar,
                            GtkToolbarStyle style );

void gtk_toolbar_set_tooltips( GtkWidget *toolbar,
                               gint enable );

```

Where orientation is one of `GTK_ORIENTATION_HORIZONTAL` or `GTK_ORIENTATION_VERTICAL`. The style is used to set appearance of the toolbar items by using one of `GTK_TOOLBAR_ICONS`, `GTK_TOOLBAR_TEXT`, or `GTK_TOOLBAR_BOTH`.

To show some other things that can be done with a toolbar, let's take the following program:

```

#include <gtk/gtk.h>
#include "gtk.xpm"

/* This function is connected to the Close button or
 * closing the window from the WM */
gint delete_event (GtkWidget *widget, GdkEvent *event, gpointer data)
{
    gtk_main_quit ();
    return(FALSE);}

```

We include a nice XPM picture to serve as an icon for all of the buttons.

```

GtkWidget* close_button; /* This button will emit signal to close
 * application */

```

```

GtkWidget* tooltips_button; /* to enable/disable tooltips */
GtkWidget* text_button,
      * icon_button,
      * both_button; /* radio buttons for toolbar style */
GtkWidget* entry; /* a text entry to show packing any widget into
      * toolbar */

```

Not all of the above widgets are needed here, but to make things everything is put together:

```

/* when one of the buttons is toggled, we just
 * check which one is active and set the style of the toolbar
 * accordingly
 * ATTENTION: our toolbar is passed as data to callback ! */
void radio_event (GtkWidget *widget, gpointer data)
{
    if (GTK_TOGGLE_BUTTON (text_button)->active)
        gtk_toolbar_set_style(GTK_TOOLBAR ( data ), GTK_TOOLBAR_TEXT);
    else if (GTK_TOGGLE_BUTTON (icon_button)->active)
        gtk_toolbar_set_style(GTK_TOOLBAR ( data ), GTK_TOOLBAR_ICONS);
    else if (GTK_TOGGLE_BUTTON (both_button)->active)
        gtk_toolbar_set_style(GTK_TOOLBAR ( data ), GTK_TOOLBAR_BOTH);
}

/* even easier, just check given toggle button and enable/disable
 * tooltips */
void toggle_event (GtkWidget *widget, gpointer data)
{
    gtk_toolbar_set_tooltips (GTK_TOOLBAR ( data ),
        GTK_TOGGLE_BUTTON (widget)->active);
}

```

The above are just two callback functions that will be called when one of the buttons on a toolbar is pressed.

```

int main (int argc, char *argv[])
{
    /* Here is our main window (a dialog) and a handle for the handlebox */
    GtkWidget* dialog;
    GtkWidget* handlebox;

    /* Ok, we need a toolbar, an icon with a mask (one for all of
     the buttons) and an icon widget to put this icon in (but
     we'll create a separate widget for each button) */
    GtkWidget * toolbar;
    GdkPixmap * icon;
    GdkBitmap * mask;
    GtkWidget * iconw;

    /* this is called in all GTK application. */
    gtk_init (&argc, &argv);
    /* create a new window with a given title, and nice size */
    dialog = gtk_dialog_new ();
    gtk_window_set_title ( GTK_WINDOW ( dialog ) , "GTKToolbar Tutorial");
    gtk_widget_set_usize( GTK_WIDGET ( dialog ) , 600 , 300 );
    GTK_WINDOW ( dialog ) ->allow_shrink = TRUE;
}

```

```

/* typically we quit if someone tries to close us */
gtk_signal_connect ( GTK_OBJECT ( dialog ), "delete_event",
                    GTK_SIGNAL_FUNC ( delete_event ), NULL);

/* we need to realize the window because we use pixmaps for
 * items on the toolbar in the context of it */
gtk_widget_realize ( dialog );

/* to make it nice we'll put the toolbar into the handle box,
 * so that it can be detached from the main window */
handlebox = gtk_handle_box_new ();
gtk_box_pack_start ( GTK_BOX ( GTK_DIALOG(dialog)->vbox ),
                    handlebox, FALSE, FALSE, 5 );

```

A handle box is just another box that can be used to pack widgets in to. The difference between it and typical boxes is that it can be detached from a parent window (or, in fact, the handle box remains in the parent, but it is reduced to a very small rectangle, while all of its contents are reparented to a new freely floating window). It is usually nice to have a detachable toolbar, so these two widgets occur together quite often.

```

/* toolbar will be horizontal, with both icons and text, and
 * with 5pxl spaces between items and finally,
 * we'll also put it into our handlebox */
toolbar = gtk_toolbar_new ( GTK_ORIENTATION_HORIZONTAL,
                           GTK_TOOLBAR_BOTH );
gtk_container_set_border_width ( GTK_CONTAINER ( toolbar ), 5 );
gtk_toolbar_set_space_size ( GTK_TOOLBAR ( toolbar ), 5 );
gtk_container_add ( GTK_CONTAINER ( handlebox ), toolbar );

/* now we create icon with mask: we'll reuse it to create
 * icon widgets for toolbar items */
icon = gdk_pixmap_create_from_xpm_d ( dialog->window, &mask,
                                     &dialog->style->white, gtk_xpm );

```

The above code is just a straightforward initialization of the toolbar widget and creation of a GDK pixmap with its mask

```

/* our first item is <close> button */
iconw = gtk_pixmap_new ( icon, mask ); /* icon widget */
close_button =
    gtk_toolbar_append_item ( GTK_TOOLBAR ( toolbar ), /* our toolbar */
                             "Close", /* button label */
                             "Closes this app", /* this button's tooltip */
                             "Private", /* tooltip private info */
                             iconw, /* icon widget */
                             GTK_SIGNAL_FUNC ( delete_event ), /* a signal */
                             NULL );
gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ); /* space after item */

```

In the above code on the other hand, the simplest case can be seen: adding a button to toolbar. Just before appending a new item, a pixmap widget is constructed to serve as an icon for this item; this step will have to be repeated for each new item. Just after the item we also add a space, so the following items will not touch each other. As

you see `gtk_toolbar_append_item` returns a pointer to our newly created button widget, so that we can work with it in the normal way.

```
/* now, let's make our radio buttons group... */
iconw = gtk_pixmap_new ( icon, mask );
icon_button = gtk_toolbar_append_element(
    GTK_TOOLBAR(toolbar),
    GTK_TOOLBAR_CHILD_RADIOBUTTON, /* a type of element */
    NULL, /* pointer to widget */
    "Icon", /* label */
    "Only icons in toolbar", /* tooltip */
    "Private", /* tooltip private string */
    iconw, /* icon */
    GTK_SIGNAL_FUNC (radio_event), /* signal */
    toolbar); /* data for signal */
gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );
```

Here we begin creating a radio buttons group. To do this we use `gtk_toolbar_append_element`. In fact, using this function one can also +add simple items or even spaces (type = `GTK_TOOLBAR_CHILD_SPACE` or `+GTK_TOOLBAR_CHILD_BUTTON`). In the above case we start creating a radio group. In creating other radio buttons for this group a pointer to the previous button in the group is required, so that a list of buttons can be easily constructed.

```
/* following radio buttons refer to previous ones */
iconw = gtk_pixmap_new ( icon, mask );
text_button =
    gtk_toolbar_append_element(GTK_TOOLBAR(toolbar),
        GTK_TOOLBAR_CHILD_RADIOBUTTON,
        icon_button,
        "Text",
        "Only texts in toolbar",
        "Private",
        iconw,
        GTK_SIGNAL_FUNC (radio_event),
        toolbar);
gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );

iconw = gtk_pixmap_new ( icon, mask );
both_button =
    gtk_toolbar_append_element(GTK_TOOLBAR(toolbar),
        GTK_TOOLBAR_CHILD_RADIOBUTTON,
        text_button,
        "Both",
        "Icons and text in toolbar",
        "Private",
        iconw,
        GTK_SIGNAL_FUNC (radio_event),
        toolbar);
gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(both_button),TRUE);
```

In the end we have to set the state of one of the buttons manually (otherwise they all stay in active state, preventing us from switching between them).

```

/* here we have just a simple toggle button */
iconw = gtk_pixmap_new ( icon, mask );
tooltips_button =
  gtk_toolbar_append_element(GTK_TOOLBAR(toolbar),
    GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
    NULL,
    "Tooltips",
    "Toolbar with or without tips",
    "Private",
    iconw,
    GTK_SIGNAL_FUNC (toggle_event),
    toolbar);
gtk_toolbar_append_space ( GTK_TOOLBAR ( toolbar ) );
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(tooltips_button),TRUE);

```

A toggle button can be created in the obvious way (if one knows how to create radio buttons already).

```

/* to pack a widget into toolbar, we only have to
 * create it and append it with an appropriate tooltip */
entry = gtk_entry_new ();
gtk_toolbar_append_widget( GTK_TOOLBAR (toolbar),
    entry,
    "This is just an entry",
    "Private" );

/* well, it isn't created within the toolbar, so we must still show it */
gtk_widget_show ( entry );

```

As you see, adding any kind of widget to a toolbar is simple. The one thing you have to remember is that this widget must be shown manually (contrary to other items which will be shown together with the toolbar).

```

/* that's it ! let's show everything. */
gtk_widget_show ( toolbar );
gtk_widget_show (handlebox);
gtk_widget_show ( dialog );

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

return 0;
}

/* XPM */
static char * gtk_xpm[] = {
"32 39 5 1",
".   c none",
"+   c black",
"@   c #3070E0",
"#   c #F05050",
"$   c #35E035",
". . . . .+ . . . . .",
". . . . .+++++ . . . . .",
". . . . .+++++@@+ . . . . .",
". . . . .+++++@@@@@+ . . . . .",
". . . . .+++++@@@@@@@@@+ . . . . .",
". . . . .+++++@@@@@@@@@@@@@+ . . . . .",
". . . . .+++++@@@@@@@@@@@@@@@@@+ . . . . .",

```



## Notebooks

The Notebook Widget is a collection of "pages" that overlap each other, each page contains different information with only one page visible at a time. This widget has become more common lately in GUI programming, and it is a good way to show blocks of similar information that warrant separation in their display.

The first function call you will need to know is used to create a new notebook widget.

```
GtkWidget *gtk_notebook_new( void );
```

Once the notebook has been created, there are a number of functions that operate on the notebook widget.

First, is how to position the page indicators. These page indicators or "tabs" as they are referred to, can be positioned in four ways: top, bottom, left, or right.

```
void gtk_notebook_set_tab_pos( GtkWidget *notebook,  
                               GtkPositionType pos );
```

GtkPositionType will be one of the following, which are pretty self explanatory:

```
GTK_POS_LEFT  
GTK_POS_RIGHT  
GTK_POS_TOP  
GTK_POS_BOTTOM
```

GTK\_POS\_TOP is the default.

Next we will look at how to add pages to the notebook. There are three ways to add pages to the Notebook.

```
void gtk_notebook_append_page( GtkWidget *notebook,  
                               GtkWidget *child,  
                               GtkWidget *tab_label );
```

```
void gtk_notebook_prepend_page( GtkWidget *notebook,  
                                GtkWidget *child,  
                                GtkWidget *tab_label );
```

```
void gtk_notebook_insert_page( GtkWidget *notebook,  
                               GtkWidget *child,  
                               GtkWidget *tab_label,  
                               gint position );
```

The first two functions add pages to the notebook by inserting them from the back of the notebook (append), or the front of the notebook (prepend). child is the widget that is placed within the notebook page, and tab\_label is the label for the page being added. The child widget must be created separately, and is typically a set of options setup within one of the other container widgets, such as a table.

The final function for adding a page to the notebook however, contains all of the properties of the previous two, but it allows you to specify what position you want the page to be in the notebook. The parameters are the same as `_append_` and `_prepend_` except it contains an extra parameter, `position`. This parameter is used to specify what place this page will be inserted into the first page having position zero.

To remove a page from the notebook, this function takes the page specified by `page_num` and removes it from the widget pointed to by `notebook`.

```
void gtk_notebook_remove_page( GtkNotebook *notebook,
                              gint      page_num );
```

To find out what the current page is in a notebook use the function:

```
gint gtk_notebook_get_current_page( GtkNotebook *notebook );
```

These next two functions are simple calls to move the notebook page forward or backward. Just like a turning the pages of an ordinary book, simply provide the respective function call with the notebook widget you wish to operate on, `gtk_notebook_next_page` or `gtk_notebook_prev_page`. But if the Notebook is currently on the last page, and `gtk_notebook_next_page` is called, the notebook will wrap back to the first page. Likewise, if the Notebook is on the first page, and `gtk_notebook_prev_page` is called, the notebook will wrap to the last page.

```
void gtk_notebook_next_page( GtkNoteBook *notebook );
```

```
void gtk_notebook_prev_page( GtkNoteBook *notebook );
```

This next function sets the "active" page. If you wish the notebook to be opened to page 5 for example, you would use this function. Without using this function, the notebook defaults to the first page.

```
void gtk_notebook_set_page( GtkNotebook *notebook,
                           gint      page_num );
```

The next two functions add or remove the notebook page tabs and the notebook border respectively.

```
void gtk_notebook_set_show_tabs( GtkNotebook *notebook,
                                 gboolean  show_tabs);
```

```
void gtk_notebook_set_show_border( GtkNotebook *notebook,
                                   gboolean  show_border );
```

The next function is useful when the you have a large number of pages, and the tabs don't fit on the page. It allows the tabs to be scrolled through using two arrow buttons.

```
void gtk_notebook_set_scrollable( GtkNotebook *notebook,
                                  gboolean  scrollable );
```

`show_tabs`, `show_border` and `scrollable` can be either TRUE or FALSE.

Now let's look at an example, it is expanded from the testgtk.c code that comes with the GTK distribution. This small program creates a window with a notebook and six buttons. The notebook contains 11 pages, added in three different ways, appended, inserted, and prepended. The buttons allow you rotate the tab positions, add/remove the tabs and border, remove a page, change pages in both a forward and backward manner, and exit the program.

```

/* example-start notebook notebook.c */

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

/* This function rotates the position of the tabs */
void rotate_book( GtkWidget *button,
                 GtkWidget *notebook )
{
    gtk_notebook_set_tab_pos (notebook, (notebook->tab_pos + 1) % 4);
}

/* Add/Remove the page tabs and the borders */
void tabsborder_book( GtkWidget *button,
                    GtkWidget *notebook )
{
    gint tval = FALSE;
    gint bval = FALSE;
    if (notebook->show_tabs == 0)
        tval = TRUE;
    if (notebook->show_border == 0)
        bval = TRUE;

    gtk_notebook_set_show_tabs (notebook, tval);
    gtk_notebook_set_show_border (notebook, bval);
}

/* Remove a page from the notebook */
void remove_book( GtkWidget *button,
                 GtkWidget *notebook )
{
    gint page;

    page = gtk_notebook_get_current_page(notebook);
    gtk_notebook_remove_page (notebook, page);
    /* Need to refresh the widget --
       This forces the widget to redraw itself. */
    gtk_widget_draw(GTK_WIDGET(notebook), NULL);
}

gint delete( GtkWidget *widget,
            GtkWidget *event,
            gpointer data )
{
    gtk_main_quit();
    return(FALSE);
}

```

```
int main( int argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;
    GtkWidget *notebook;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *checkboxbutton;
    int i;
    char bufferf[32];
    char bufferl[32];

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                       GTK_SIGNAL_FUNC (delete), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    table = gtk_table_new(3,6,FALSE);
    gtk_container_add (GTK_CONTAINER (window), table);

    /* Create a new notebook, place the position of the tabs */
    notebook = gtk_notebook_new ();
    gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_TOP);
    gtk_table_attach_defaults(GTK_TABLE(table), notebook, 0,6,0,1);
    gtk_widget_show(notebook);

    /* Let's append a bunch of pages to the notebook */
    for (i=0; i < 5; i++) {
        sprintf(bufferf, "Append Frame %d", i+1);
        sprintf(bufferl, "Page %d", i+1);

        frame = gtk_frame_new (bufferf);
        gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
        gtk_widget_set_usize (frame, 100, 75);
        gtk_widget_show (frame);

        label = gtk_label_new (bufferf);
        gtk_container_add (GTK_CONTAINER (frame), label);
        gtk_widget_show (label);

        label = gtk_label_new (bufferl);
        gtk_notebook_append_page (GTK_NOTEBOOK (notebook), frame,
label);
    }

    /* Now let's add a page to a specific spot */
    checkboxbutton = gtk_check_button_new_with_label ("Check me please!");
    gtk_widget_set_usize(checkboxbutton, 100, 75);
    gtk_widget_show (checkboxbutton);
}
```

```

label = gtk_label_new ("Add page");
gtk_notebook_insert_page (GTK_NOTEBOOK (notebook), checkbox, label, 2);

/* Now finally let's prepend pages to the notebook */
for (i=0; i < 5; i++) {
    sprintf(bufferf, "Prepend Frame %d", i+1);
    sprintf(bufferl, "PPage %d", i+1);

    frame = gtk_frame_new (bufferf);
    gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
    gtk_widget_set_usize (frame, 100, 75);
    gtk_widget_show (frame);

    label = gtk_label_new (bufferf);
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_widget_show (label);

    label = gtk_label_new (bufferl);
    gtk_notebook_prepend_page (GTK_NOTEBOOK(notebook), frame,
label);
}

/* Set what page to start at (page 4) */
gtk_notebook_set_page (GTK_NOTEBOOK(notebook), 3);

/* Create a bunch of buttons */
button = gtk_button_new_with_label ("close");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                          GTK_SIGNAL_FUNC (delete), NULL);
gtk_table_attach_defaults(GTK_TABLE(table), button, 0, 1, 1, 2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("next page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                          (GtkSignalFunc) gtk_notebook_next_page,
                          GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 1, 2, 1, 2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("prev page");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                          (GtkSignalFunc) gtk_notebook_prev_page,
                          GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 2, 3, 1, 2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("tab position");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                  (GtkSignalFunc) rotate_book,
                  GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 3, 4, 1, 2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("tabs/border on/off");

```

```

gtk_signal_connect (GTK_OBJECT (button), "clicked",
                  (GtkSignalFunc) tabsborder_book,
                  GTK_OBJECT (notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 4,5,1,2);
gtk_widget_show(button);

button = gtk_button_new_with_label ("remove page");
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                  (GtkSignalFunc) remove_book,
                  GTK_OBJECT(notebook));
gtk_table_attach_defaults(GTK_TABLE(table), button, 5,6,1,2);
gtk_widget_show(button);

gtk_widget_show(table);
gtk_widget_show(window);

gtk_main ();

return(0);
}
/* example-end */

```

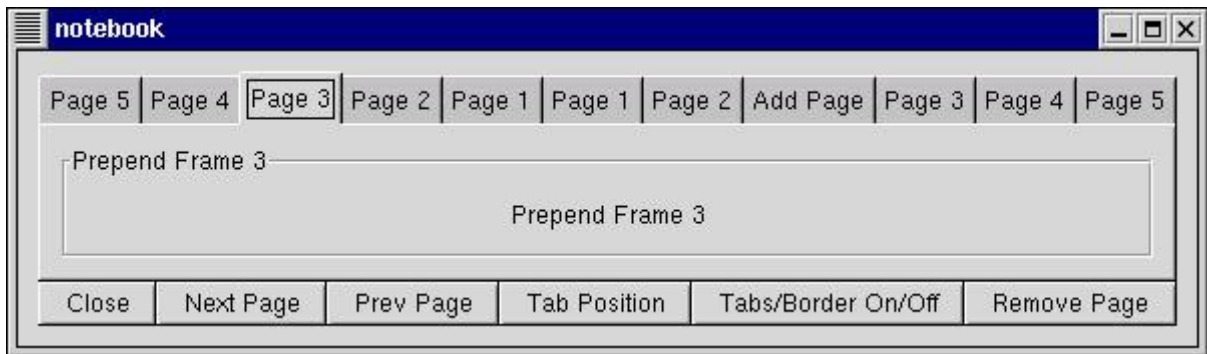


Fig 8.1 Example Screenshot

## Handling Events

### *Signals and Callbacks*

GTK is an event driven toolkit, which means it will sleep in `gtk_main` 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, we set up a signal handler to catch these signals and call the appropriate function. This is done by using a function such as:

```
gint gtk_signal_connect( GtkObject *object,
                        gchar *name,
                        GtkSignalFunc 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 you wish to catch. The third is the function you wish to be called when it is caught, and the fourth, the data you wish to have passed to this function.

The function specified in the third argument is called a "callback function", and should generally be of the form

```
void callback_func( GtkWidget *widget,
                   gpointer callback_data );
```

where the first argument will be a pointer to the widget that emitted the signal, and the second a pointer to the data given as the last argument to the `gtk_signal_connect()` function 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. For example, the CList "select\_row" signal provides both row and column parameters.

Another call used is:

```
gint gtk_signal_connect_object( GtkObject *object,
                               gchar *name,
                               GtkSignalFunc func,
                               GtkObject *slot_object );
```

`gtk_signal_connect_object()` is the same as `gtk_signal_connect()` except that the callback function only uses one argument, a pointer to a GTK object. So when using this function to connect signals, the callback should be of the form

```
void callback_func( GtkObject *object );
```

where the object is usually a widget. We usually don't setup callbacks for `gtk_signal_connect_object` however. They are usually used to call a GTK function that accepts a single widget or object as an argument.

The purpose of having two functions to connect signals is simply to allow the callbacks to have a different number of arguments. Many functions in the GTK library accept only a single `GtkWidget` pointer as an argument, so you want to use the `gtk_signal_connect_object()` for these, whereas for your functions, you may need to have additional data supplied to the callbacks.

## Events

In addition to the signal mechanism described above, there is a set of *events* that reflect the X event mechanism. Callbacks may also be attached to these events. These events are:

- `event`
- `button_release_event`
- `motion_notify_event`
- `delete_event`
- `destroy_event`
- `expose_event`
- `key_press_event`
- `key_release_event`
- `enter_notify_event`
- `leave_notify_event`
- `configure_event`
- `focus_in_event`
- `focus_out_event`
- `map_event`
- `unmap_event`
- `property_notify_event`
- `selection_clear_event`
- `selection_request_event`
- `selection_notify_event`
- `proximity_in_event`
- `proximity_out_event`
- `drag_begin_event`
- `drag_request_event`
- `drag_end_event`
- `drag_enter_event`
- `drop_leave_event`
- `drop_data_available_event`
- `other_event`

In order to connect a callback function to one of these events you use the function `gtk_signal_connect`, as described above, using one of the above event names as the name parameter. The callback function for events has a slightly different form than that for signals:

```
gint callback_func( GtkWidget *widget,  
                  GdkEvent *event,  
                  gpointer callback_data );
```

`GdkEvent` is a C union structure whose type will depend upon which of the above events has occurred. In order for us to tell which event has been issued each of the possible alternatives has a type member that reflects the event being issued. The other components of the event structure will depend upon the type of the event. Possible values for the type are:

`GDK_NOTHING`

```
GDK_DELETE
GDK_DESTROY
GDK_EXPOSE
GDK_MOTION_NOTIFY
GDK_BUTTON_PRESS
GDK_2BUTTON_PRESS
GDK_3BUTTON_PRESS
GDK_BUTTON_RELEASE
GDK_KEY_PRESS
GDK_KEY_RELEASE
GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY
GDK_FOCUS_CHANGE
GDK_CONFIGURE
GDK_MAP
GDK_UNMAP
GDK_PROPERTY_NOTIFY
GDK_SELECTION_CLEAR
GDK_SELECTION_REQUEST
GDK_SELECTION_NOTIFY
GDK_PROXIMITY_IN
GDK_PROXIMITY_OUT
GDK_DRAG_BEGIN
GDK_DRAG_REQUEST
GDK_DROP_ENTER
GDK_DROP_LEAVE
GDK_DROP_DATA_AVAIL
GDK_CLIENT_EVENT
GDK_VISIBILITY_NOTIFY
GDK_NO_EXPOSE
GDK_OTHER_EVENT      /* Deprecated, use filters instead */
```

So, to connect a callback function to one of these events we would use something like:

```
gtk_signal_connect( GTK_OBJECT(button), "button_press_event",
                   GTK_SIGNAL_FUNC(button_press_callback),
                   NULL);
```

This assumes that `button` is a `Button` widget. Now, when the mouse is over the button and a mouse button is pressed, the function `button_press_callback` will be called. This function may be declared as:

```
static gint button_press_callback( GtkWidget *widget,
                                   GdkEventButton *event,
                                   gpointer data );
```

Note that we can declare the second argument as type `GdkEventButton` as we know what type of event will occur for this function to be called.

The value returned from this function indicates whether the event should be propagated further by the GTK event handling mechanism. Returning `TRUE` indicates that the event has been handled, and that it should not propagate further. Returning `FALSE` continues the normal event handling.

## Example HelloWorld ala GTK

This example program is found in GTK Source Distribution:

```
/* example-start helloworld helloworld.c */

#include <gtk/gtk.h>

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

gint 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 */
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. */
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                       GTK_SIGNAL_FUNC (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. */
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                   GTK_SIGNAL_FUNC (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. */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (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. */
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                          GTK_SIGNAL_FUNC (gtk_widget_destroy),
                          GTK_OBJECT (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);
}
/* example-end */
```

Taken from:

<http://www.gtk.org>

<http://www.gnome.org>

<http://www.gimp.org/~sjburgess/gimp-history.html>

<http://primates.ximian.com/~miguel/gnome-history.html>