

Part I. GTK+ Overview: GTK+ 3 Reference Manual

GTK+ is a library for creating graphical user interfaces. It works on many UNIX-like platforms, Windows, and OS X. GTK+ is released under the GNU Library General Public License (GNU LGPL), which allows for flexible licensing of client applications. GTK+ has a C-based object-oriented architecture that allows for maximum flexibility. Bindings for many other languages have been written, including C++, Objective-C, Guile/Scheme, Perl, Python, TOM, Ada95, Free Pascal, and Eiffel.

GTK+ depends on the following libraries:

GLib	A general-purpose utility library, not specific to graphical user interfaces. GLib provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on.
GObject	A library that provides a type system, a collection of fundamental types including an object type, a signal system.
GIO	A modern, easy-to-use VFS API including abstractions for files, drives, volumes, stream IO, as well as network programming and DBus communication.
cairo	Cairo is a 2D graphics library with support for multiple output devices.
Pango	Pango is a library for internationalized text handling. It centers around the PangoLayout object, representing a paragraph of text. Pango provides the engine for GtkTextView, GtkLabel, GtkEntry, and other widgets that display text.
ATK	ATK is the Accessibility Toolkit. It provides a set of generic interfaces allowing accessibility technologies to interact with a graphical user interface. For example, a screen reader uses ATK to discover the text in an interface and read it to blind users. GTK+ widgets have built-in support for accessibility using the ATK framework.
GdkPixbuf	This is a small library which allows you to create GdkPixbuf ("pixel buffer") objects from image data or image files. Use a GdkPixbuf in combination with GtkImage to display images.
GDK	GDK is the abstraction layer that allows GTK+ to support multiple windowing systems. GDK provides window system facilities on X11, Windows, and OS X.
GTK+	The GTK+ library itself contains <i>widgets</i> , that is, GUI components such as GtkButton or GtkTextView.

Table of Contents

[Getting Started with GTK+](#)

[Basics](#)

[Packing](#)

[Building user interfaces](#)

[Building applications](#)

[A trivial application](#)

[Populating the window](#)

[Opening files](#)

[An application menu](#)

[A preference dialog](#)

[Adding a search bar](#)

[Adding a side bar](#)

[Properties](#)

[Header bar](#)

[Custom Drawing](#)

[Mailing lists and bug reports](#) — Getting help with GTK+

[Common Questions](#) — Find answers to common questions in the GTK+ manual

[The GTK+ Drawing Model](#) — The GTK+ drawing model in detail

[The GTK+ Input and Event Handling Model](#) — GTK+ input and event handling in detail

Getting Started with GTK+

[Basics](#)

[Packing](#)

[Building user interfaces](#)

[Building applications](#)

[A trivial application](#)

[Populating the window](#)

[Opening files](#)

[An application menu](#)

[A preference dialog](#)

[Adding a search bar](#)

[Adding a side bar](#)

[Properties](#)

[Header bar](#)

[Custom Drawing](#)

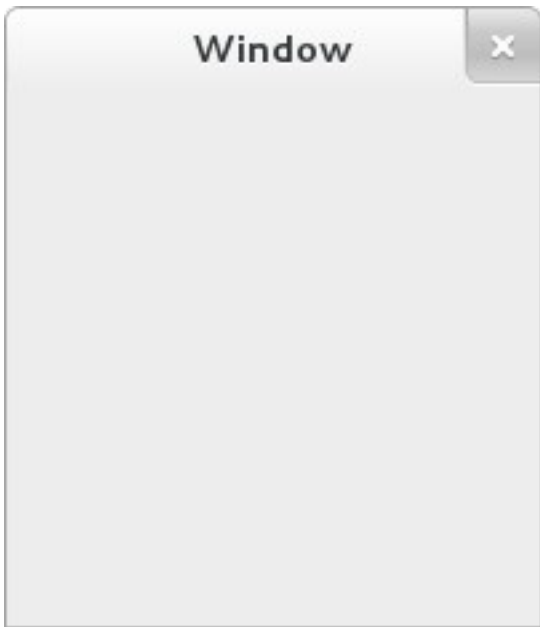
GTK+ is a [widget toolkit](#). Each user interface created by GTK+ consists of widgets. This is implemented in C using GObject, an object-oriented framework for C. Widgets are organized in a hierarchy. The window widget is the main container. The user interface is then built by adding buttons, drop-down menus, input fields, and other widgets to the window. If you are creating complex user interfaces it is recommended to use [GtkBuilder](#) and its GTK-specific markup description language, instead of assembling the interface manually. You can also use a visual user interface editor, like [Glade](#).

GTK+ is event-driven. The toolkit listens for events such as a click on a button, and passes the event to your application.

This chapter contains some tutorial information to get you started with GTK+ programming. It assumes that you have GTK+, its dependencies and a C compiler installed and ready to use. If you need to build GTK+ itself first, refer to the [Compiling the GTK+ libraries](#) section in this reference.

Basics

To begin our introduction to GTK, we'll start with a simple signal-based Gtk application. This program will create an empty 200 × 200 pixel window.



Create a new file with the following content named `example-0.c`.

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

static void
activate (GtkApplication* app,
          gpointer        user_data)
{
    GtkWidget *window;

    window = gtk_application_window_new (app);
    gtk_window_set_title (GTK_WINDOW (window), "Window");
    gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);
    gtk_widget_show_all (window);
}

int
main (int   argc,
      char **argv)
{
    GtkApplication *app;
    int status;

    app = gtk_application_new ("org.gtk.example", G_APPLICATION_FLAGS_NONE);
    g_signal_connect (app, "activate", G_CALLBACK (activate), NULL);
    status = g_application_run (G_APPLICATION (app), argc, argv);
    g_object_unref (app);

    return status;
}
```

You can compile the program above with GCC using:

```
gcc `pkg-config --cflags gtk+-3.0` -o example-0 example-0.c `pkg-config --libs gtk+-3.0`
```

For more information on how to compile a GTK+ application, please refer to the [Compiling GTK+ Applications](#) section in this reference.

All GTK+ applications will, of course, include `gtk/gtk.h`, which declares functions, types and macros required by GTK+ applications.

Even if GTK+ installs multiple header files, only the top-level `gtk/gtk.h` header can be directly included by third party code. The compiler will abort with an error if any other header is directly included.

In a GTK+ application, the purpose of the `main()` function is to create a [GtkApplication](#) object and run it. In this example a [GtkApplication](#) pointer named `app` is called and then initialized using [gtk_application_new\(\)](#).

When creating a [GtkApplication](#) you need to pick an application identifier (a name) and input to [gtk_application_new\(\)](#) as parameter. For this example `org.gtk.example` is used but for choosing an identifier for your application see [this guide](#). Lastly [gtk_application_new\(\)](#) takes a `GApplicationFlags` as input for your application, if your application would have special needs.

Next the [activate signal](#) is connected to the `activate()` function above the `main()` functions. The `activate` signal will be sent when your application is launched with `g_application_run()` on the line below. The `gtk_application_run()` also takes as arguments the pointers to the command line arguments counter and string array; this allows GTK+ to parse specific command line arguments that control the behavior of GTK+ itself. The parsed arguments will be removed from the array, leaving the unrecognized ones for your application to parse.

Within `g_application_run` the `activate()` signal is sent and we then proceed into the `activate()` function of the application. Inside the `activate()` function we want to construct our GTK window, so that a window is shown when the application is launched. The call to [gtk_application_window_new\(\)](#) will create a new [GtkWindow](#) and store it inside the window pointer. The window will have a frame, a title bar, and window controls depending on the platform.

A window title is set using [gtk_window_set_title\(\)](#). This function takes a `GtkWindow*` pointer and a string as input. As our window pointer is a `GtkWidget` pointer, we need to cast it to `GtkWindow*`. But instead of casting `window` via `(GtkWindow*)`, `window` can be cast using the macro `GTK_WINDOW()`. `GTK_WINDOW()` will check if the pointer is an instance of the `GtkWindow` class, before casting, and emit a warning if the check fails. More information about this convention can be found [here](#).

Finally the window size is set using `gtk_window_set_default_size` and the window is then shown by GTK via [gtk_widget_show_all\(\)](#).

When you exit the window, by for example pressing the X, the `g_application_run()` in the main loop returns with a number which is saved inside an integer named "status". Afterwards, the [GtkApplication](#) object is freed from memory with `g_object_unref()`. Finally the status integer is returned and the GTK application exits.

While the program is running, GTK+ is receiving *events*. These are typically input events caused by the user interacting with your program, but also things like messages from the window manager or other applications. GTK+ processes these and as a result, *signals* may be emitted on your widgets. Connecting handlers for these signals is how you normally make your program do something in response to user input.

The following example is slightly more complex, and tries to showcase some of the capabilities of GTK+.

In the long tradition of programming languages and libraries, it is called *Hello, World*.



Example 1. Hello World in GTK+

Create a new file with the following content named example-1.c.

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

static void
print_hello (GtkWidget *widget,
             gpointer   data)
{
    g_print ("Hello World\n");
}

static void
activate (GtkApplication *app,
          gpointer        user_data)
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *button_box;

    window = gtk_application_window_new (app);
    gtk_window_set_title (GTK_WINDOW (window), "Window");
    gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);

    button_box = gtk_button_box_new (GTK_ORIENTATION_HORIZONTAL);
    gtk_container_add (GTK_CONTAINER (window), button_box);

    button = gtk_button_new_with_label ("Hello World");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);
    g_signal_connect_swapped (button, "clicked", G_CALLBACK (gtk_widget_destroy), window);
    gtk_container_add (GTK_CONTAINER (button_box), button);

    gtk_widget_show_all (window);
}

int
main (int   argc,
      char **argv)
{
    GtkApplication *app;
    int status;

    app = gtk_application_new ("org.gtk.example", G_APPLICATION_FLAGS_NONE);
    g_signal_connect (app, "activate", G_CALLBACK (activate), NULL);
    status = g_application_run (G_APPLICATION (app), argc, argv);
    g_object_unref (app);

    return status;
}
```

You can compile the program above with GCC using:

```
gcc `pkg-config --cflags gtk+-3.0` -o example-1 example-1.c `pkg-config --libs gtk+-3.0`
```

As seen above, example-1.c builds further upon example-0.c by adding a button to our window, with the label "Hello World". Two new GtkWidget pointers are declared to accomplish this, `button` and `button_box`. The `button_box` variable is created to store a [GtkButtonBox](#) which is GTK+'s way of controlling the size and layout of buttons. The [GtkButtonBox](#) is created and assigned to `gtk_button_box_new()` which takes a [GtkOrientation](#) enum as parameter. The buttons which this box will contain can either be stored horizontally or vertically but this does not matter in this particular case as we are dealing with only one button. After initializing `button_box` with horizontal orientation, the code adds the `button_box` widget to the window widget using [gtk_container_add\(\)](#).

Next the `button` variable is initialized in similar manner. [gtk_button_new_with_label\(\)](#) is called which returns a `GtkButton` to be stored inside `button`. Afterwards `button` is added to our `button_box`. Using [g_signal_connect](#) the button is connected to a function in our app called `print_hello()`, so that when the button is clicked, GTK will call this function. As the `print_hello()` function does not use any data as input, `NULL` is passed to it. `print_hello()` calls `g_print()` with the string "Hello World" which will print Hello World in a terminal if the GTK application was started from one.

After connecting `print_hello()`, another signal is connected to the "clicked" state of the button using [g_signal_connect_swapped\(\)](#). This function is similar to a [g_signal_connect\(\)](#) with the difference lying in how the callback function is treated. [g_signal_connect_swapped\(\)](#) allow you to specify what the callback function should take as parameter by letting you pass it as data. In this case the function being called back is [gtk_widget_destroy\(\)](#) and the window pointer is passed to it. This has the effect that when the button is clicked, the whole GTK window is destroyed. In contrast if a normal [g_signal_connect\(\)](#) were used to connect the "clicked" signal with [gtk_widget_destroy\(\)](#), then the button itself would have been destroyed, not the window. More information about creating buttons can be found [here](#).

The rest of the code in example-1.c is identical to example-0.c. Next section will elaborate further on how to add several GtkWidget to your GTK application.

Packing

When creating an application, you'll want to put more than one widget inside a window. When you want to put more than one widget into a window, it becomes important to control how each widget is positioned and sized. This is where packing comes in.

GTK+ comes with a large variety of *layout containers* whose purpose it is to control the layout of the child widgets that are added to them. See [Layout Containers](#) for an overview.

The following example shows how the `GtkGrid` container lets you arrange several buttons:



Example 2. Packing buttons

Create a new file with the following content named example-2.c.

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

static void
print_hello (GtkWidget *widget,
             gpointer   data)
{
    g_print ("Hello World\n");
}

static void
activate (GtkApplication *app,
          gpointer        user_data)
{
    GtkWidget *window;
    GtkWidget *grid;
    GtkWidget *button;

    /* create a new window, and set its title */
    window = gtk_application_window_new (app);
    gtk_window_set_title (GTK_WINDOW (window), "Window");
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* Here we construct the container that is going pack our buttons */
    grid = gtk_grid_new ();

    /* Pack the container in the window */
    gtk_container_add (GTK_CONTAINER (window), grid);

    button = gtk_button_new_with_label ("Button 1");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    /* Place the first button in the grid cell (0, 0), and make it fill
     * just 1 cell horizontally and vertically (ie no spanning)
     */
    gtk_grid_attach (GTK_GRID (grid), button, 0, 0, 1, 1);

    button = gtk_button_new_with_label ("Button 2");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    /* Place the second button in the grid cell (1, 0), and make it fill
     * just 1 cell horizontally and vertically (ie no spanning)
     */
    gtk_grid_attach (GTK_GRID (grid), button, 1, 0, 1, 1);

    button = gtk_button_new_with_label ("Quit");
    g_signal_connect_swapped (button, "clicked", G_CALLBACK (gtk_widget_destroy), window);

    /* Place the Quit button in the grid cell (0, 1), and make it
     * span 2 columns.
     */
    gtk_grid_attach (GTK_GRID (grid), button, 0, 1, 2, 1);

    /* Now that we are done packing our widgets, we show them all
     * in one go, by calling gtk_widget_show_all() on the window.
     * This call recursively calls gtk_widget_show() on all widgets
     * that are contained in the window, directly or indirectly.
     */
}
```



```
    gtk_widget_show_all (window);  
}  
  
int  
main (int    argc,  
      char **argv)  
{  
    GtkApplication *app;  
    int status;  
  
    app = gtk_application_new ("org.gtk.example", G_APPLICATION_FLAGS_NONE);  
    g_signal_connect (app, "activate", G_CALLBACK (activate), NULL);  
    status = g_application_run (G_APPLICATION (app), argc, argv);  
    g_object_unref (app);  
  
    return status;  
}
```

You can compile the program above with GCC using:

```
gcc `pkg-config --cflags gtk+-3.0` -o example-2 example-2.c `pkg-config --libs gtk+-3.0`
```

Building user interfaces

When constructing a more complicated user interface, with dozens or hundreds of widgets, doing all the setup work in C code is cumbersome, and making changes becomes next to impossible.

Thankfully, GTK+ supports the separation of user interface layout from your business logic, by using UI descriptions in an XML format that can be parsed by the [GtkBuilder](#) class.

Example 3. Packing buttons with GtkBuilder

Create a new file with the following content named example-3.c.

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

static void
print_hello (GtkWidget *widget,
             gpointer data)
{
    g_print ("Hello World\n");
}

int
main (int argc,
      char *argv[])
{
    GtkBuilder *builder;
    GObject *window;
    GObject *button;
    GError *error = NULL;

    gtk_init (&argc, &argv);

    /* Construct a GtkBuilder instance and load our UI description */
    builder = gtk_builder_new ();
    if (gtk_builder_add_from_file (builder, "builder.ui", &error) == 0)
    {
        g_printerr ("Error loading file: %s\n", error->message);
        g_clear_error (&error);
        return 1;
    }

    /* Connect signal handlers to the constructed widgets. */
    window = gtk_builder_get_object (builder, "window");
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);

    button = gtk_builder_get_object (builder, "button1");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    button = gtk_builder_get_object (builder, "button2");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    button = gtk_builder_get_object (builder, "quit");
    g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);

    gtk_main ();

    return 0;
}
```

Create a new file with the following content named builder.ui.

```
<interface>
  <object id="window" class="GtkWindow">
    <property name="visible">True</property>
    <property name="title">Grid</property>
    <property name="border-width">10</property>
    <child>
      <object id="grid" class="GtkGrid">
        <property name="visible">True</property>
        <child>
          <object id="button1" class="GtkButton">
            <property name="visible">True</property>
            <property name="label">Button 1</property>
          </object>
          <packing>
            <property name="left-attach">0</property>
            <property name="top-attach">0</property>
          </packing>
        </child>
        <child>
          <object id="button2" class="GtkButton">
            <property name="visible">True</property>
            <property name="label">Button 2</property>
          </object>
          <packing>
            <property name="left-attach">1</property>
            <property name="top-attach">0</property>
          </packing>
        </child>
        <child>
          <object id="quit" class="GtkButton">
            <property name="visible">True</property>
            <property name="label">Quit</property>
          </object>
          <packing>
            <property name="left-attach">0</property>
            <property name="top-attach">1</property>
            <property name="width">2</property>
          </packing>
        </child>
      </object>
    <packing>
    </packing>
  </child>
</object>
</interface>
```

You can compile the program above with GCC using:

```
gcc `pkg-config --cflags gtk+-3.0` -o example-3 example-3.c `pkg-config --libs gtk+-3.0`
```

Note that GtkBuilder can also be used to construct objects that are not widgets, such as tree models, adjustments, etc. That is the reason the method we use here is called [gtk_builder_get_object\(\)](#) and returns a `GObject*` instead of a `GtkWidget*`.

Normally, you would pass a full path to [gtk_builder_add_from_file\(\)](#) to make the execution of your program independent of the current directory.

A common location to install UI descriptions and similar data is

`/usr/share/apname`.

It is also possible to embed the UI description in the source code as a string and use `gtk_builder_add_from_string()` to load it. But keeping the UI description in a separate file has several advantages: It is then possible to make minor adjustments to the UI without recompiling your program, and, more importantly, graphical UI editors such as [glade](#) can load the file and allow you to create and modify your UI by point-and-click.

Building applications

An application consists of a number of files:

The binary	This gets installed in <code>/usr/bin</code> .
A desktop file	The desktop file provides important information about the application to the desktop shell, such as its name, icon, D-Bus name, commandline to launch it, etc. It is installed in <code>/usr/share/applications</code> .
An icon	The icon gets installed in <code>/usr/share/icons/hicolor/48x48/apps</code> , where it will be found regardless of the current theme.
A settings schema	If the application uses GSettings, it will install its schema in <code>/usr/share/glib-2.0/schemas</code> , so that tools like <code>dconf-editor</code> can find it.
Other resources	Other files, such as GtkBuilder ui files, are best loaded from resources stored in the application binary itself. This eliminates the need for most of the files that would traditionally be installed in an application-specific location in <code>/usr/share</code> .

GTK+ includes application support that is built on top of `GApplication`. In this tutorial we'll build a simple application by starting from scratch, adding more and more pieces over time. Along the way, we'll learn about [GtkApplication](#), templates, resources, application menus, settings, [GtkHeaderBar](#), [GtkStack](#), [GtkSearchBar](#), [GtkListBox](#), and more.

The full, buildable sources for these examples can be found in the `examples/` directory of the GTK+ source distribution, or [online](#) in the GTK+ git repository. You can build each example separately by using `make` with the `Makefile.example` file. For more information, see the `README` included in the `examples` directory.

A trivial application

When using [GtkApplication](#), the `main()` function can be very simple. We just call `g_application_run()` and give it an instance of our application class.

```
1 #include <gtk/gtk.h>
2
3 #include "exampleapp.h"
4
5 int
6 main (int argc, char *argv[])
7 {
8     return g_application_run (G_APPLICATION (example_app_new ()), argc, argv);
9 }
```

All the application logic is in the application class, which is a subclass of [GtkApplication](#). Our example does not yet have any interesting functionality. All it does is open a window when it is activated without arguments, and open the files it is given, if it is started with arguments.

To handle these two cases, we override the `activate()` vfunc, which gets called when the application is launched without commandline arguments, and the `open()` vfunc, which gets called when the application is

launched with commandline arguments.

To learn more about GApplication entry points, consult the GIO [documentation](#).

```
1 #include <gtk/gtk.h>
2
3 #include "exampleapp.h"
4 #include "exampleappwin.h"
5
6 struct _ExampleApp
7 {
8     GtkApplication parent;
9 };
10
11 G_DEFINE_TYPE(ExampleApp, example_app, GTK_TYPE_APPLICATION);
12
13 static void
14 example_app_init (ExampleApp *app)
15 {
16 }
17
18 static void
19 example_app_activate (GApplication *app)
20 {
21     ExampleAppWindow *win;
22
23     win = example_app_window_new (EXAMPLE_APP (app));
24     gtk_window_present (GTK_WINDOW (win));
25 }
26
27 static void
28 example_app_open (GApplication *app,
29                 GFile **files,
30                 gint n_files,
31                 const gchar *hint)
32 {
33     GList *windows;
34     ExampleAppWindow *win;
35     int i;
36
37     windows = gtk_application_get_windows (GTK_APPLICATION (app));
38     if (windows)
39         win = EXAMPLE_APP_WINDOW (windows->data);
40     else
41         win = example_app_window_new (EXAMPLE_APP (app));
42
43     for (i = 0; i < n_files; i++)
44         example_app_window_open (win, files[i]);
45
46     gtk_window_present (GTK_WINDOW (win));
47 }
48
49 static void
50 example_app_class_init (ExampleAppClass *class)
51 {
52     G_APPLICATION_CLASS (class)->activate = example_app_activate;
53     G_APPLICATION_CLASS (class)->open = example_app_open;
54 }
55
56 ExampleApp *
57 example_app_new (void)
58 {
59     return g_object_new (EXAMPLE_APP_TYPE,
```

```
60         "application-id", "org.gtk.exampleapp",
61         "flags", G_APPLICATION_HANDLES_OPEN,
62         NULL);
63 }
```

Another important class that is part of the application support in GTK+ is [GtkApplicationWindow](#). It is typically subclassed as well. Our subclass does not do anything yet, so we will just get an empty window.

```
1  #include <gtk/gtk.h>
2
3  #include "exampleapp.h"
4  #include "exampleappwin.h"
5
6  struct _ExampleAppWindow
7  {
8      GtkApplicationWindow parent;
9  };
10
11 G_DEFINE_TYPE(ExampleAppWindow, example_app_window, GTK_TYPE_APPLICATION_WINDOW);
12
13 static void
14 example_app_window_init (ExampleAppWindow *app)
15 {
16 }
17
18 static void
19 example_app_window_class_init (ExampleAppWindowClass *class)
20 {
21 }
22
23 ExampleAppWindow *
24 example_app_window_new (ExampleApp *app)
25 {
26     return g_object_new (EXAMPLE_APP_WINDOW_TYPE, "application", app, NULL);
27 }
28
29 void
30 example_app_window_open (ExampleAppWindow *win,
31                         GFile *file)
32 {
33 }
```

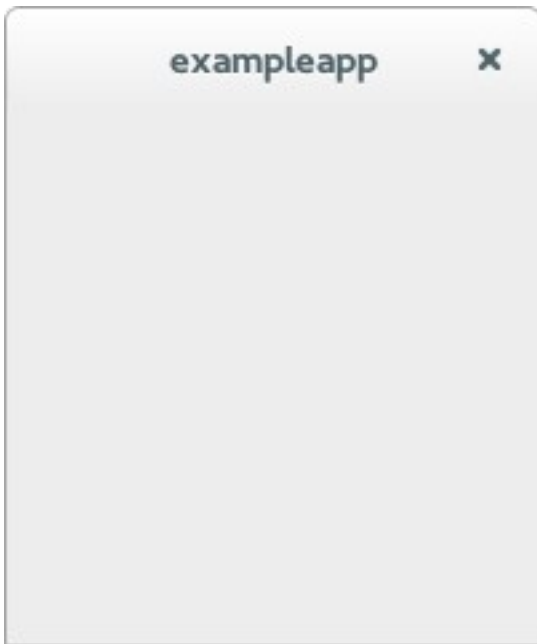
As part of the initial setup of our application, we also create an icon and a desktop file.



```
1 [Desktop Entry]
2 Type=Application
3 Name=Example
4 Icon=exampleapp
5 StartupNotify=true
6 Exec=@bindir@/exampleapp
```

Note that @bindir@ needs to be replaced with the actual path to the binary before this desktop file can be used.

Here is what we've achieved so far:



This does not look very impressive yet, but our application is already presenting itself on the session bus, it has single-instance semantics, and it accepts files as commandline arguments.

Populating the window

In this step, we use a [GtkBuilder](#) template to associate a [GtkBuilder](#) ui file with our application window class.

Our simple ui file puts a [GtkHeaderBar](#) on top of a [GtkStack](#) widget. The header bar contains a [GtkStackSwitcher](#), which is a standalone widget to show a row of 'tabs' for the pages of a [GtkStack](#).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <interface>
3   <!-- interface-requires gtk+ 3.8 -->
4   <template class="ExampleAppWindow" parent="GtkApplicationWindow">
5     <property name="title" translatable="yes">Example Application</property>
6     <property name="default-width">600</property>
7     <property name="default-height">400</property>
8     <child>
9       <object class="GtkBox" id="content_box">
10        <property name="visible">True</property>
11        <property name="orientation">vertical</property>
12        <child>
```

```

13     <object class="GtkHeaderBar" id="header">
14         <property name="visible">True</property>
15         <child type="title">
16             <object class="GtkStackSwitcher" id="tabs">
17                 <property name="visible">True</property>
18                 <property name="stack">stack</property>
19             </object>
20         </child>
21     </object>
22 </child>
23 <child>
24     <object class="GtkStack" id="stack">
25         <property name="visible">True</property>
26     </object>
27 </child>
28 </object>
29 </child>
30 </template>
31 </interface>

```

To make use of this file in our application, we revisit our [GtkApplicationWindow](#) subclass, and call [gtk_widget_class_set_template_from_resource\(\)](#) from the class init function to set the ui file as template for this class. We also add a call to [gtk_widget_init_template\(\)](#) in the instance init function to instantiate the template for each instance of our class.

...

```

static void
example_app_window_init (ExampleAppWindow *win)
{
    gtk_widget_init_template (GTK_WIDGET (win));
}

static void
example_app_window_class_init (ExampleAppWindowClass *class)
{
    gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
                                                "/org/gtk/exampleapp/window.ui");
}

```

...

[\(full source\)](#)

You may have noticed that we used the `_from_resource()` variant of the function that sets a template. Now we need to use [GLib's resource functionality](#) to include the ui file in the binary. This is commonly done by listing all resources in a `.gresource.xml` file, such as this:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <gresources>
3     <gresource prefix="/org/gtk/exampleapp">
4         <file preprocess="xml-stripblanks">window.ui</file>
5     </gresource>
6 </gresources>

```

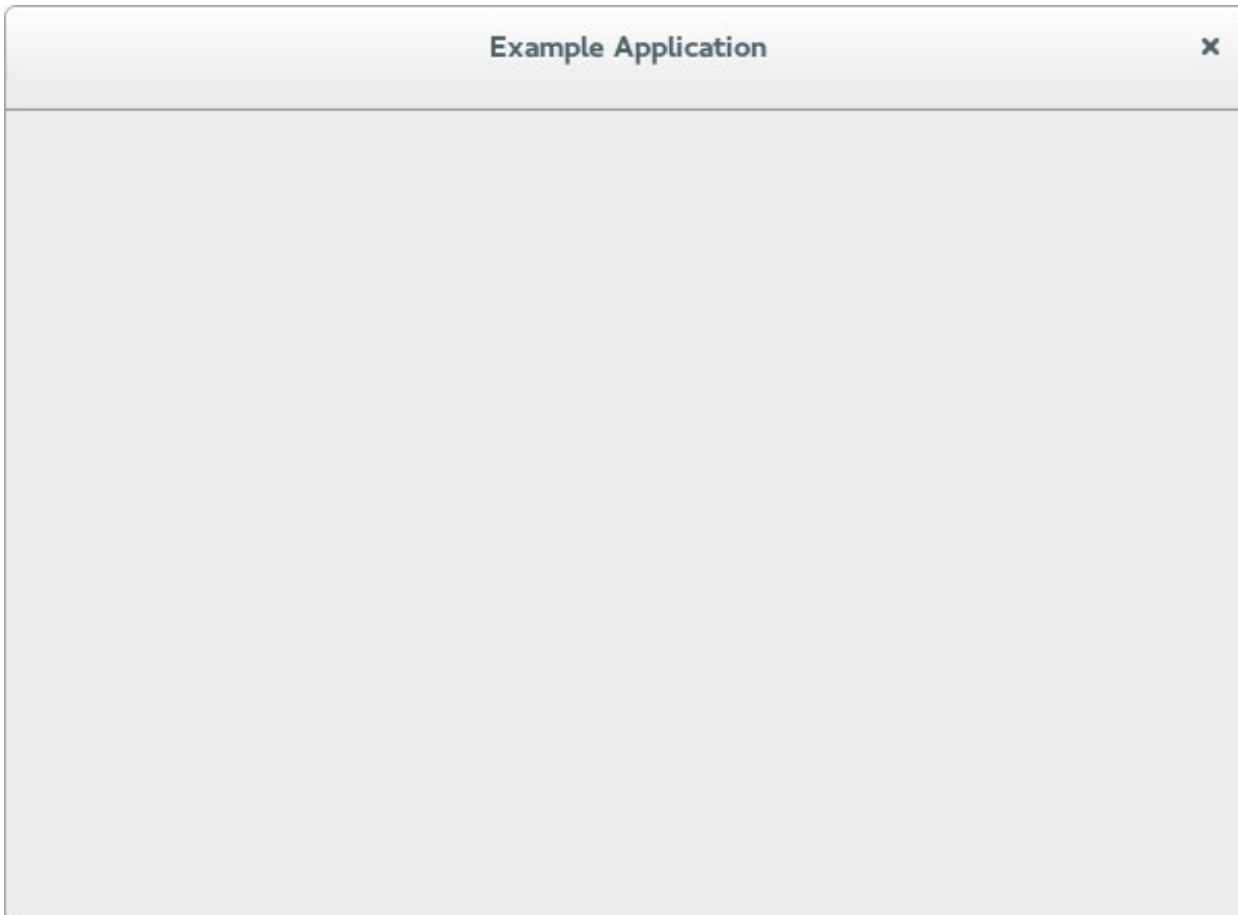
This file has to be converted into a C source file that will be compiled and linked into the application together with the other source files. To do so, we use the `glib-compile-resources` utility:

```

glib-compile-resources exampleapp.gresource.xml --target=resources.c --generate-source

```

Our application now looks like this:



Opening files

In this step, we make our application show the content of all the files that it is given on the commandline.

To this end, we add a private struct to our application window subclass and keep a reference to the [GtkStack](#) there. The [gtk_widget_class_bind_template_child_private\(\)](#) function arranges things so that after instantiating the template, the stack member of the private struct will point to the widget of the same name from the template.

```
...

struct _ExampleAppWindowPrivate
{
    GtkWidget *stack;
};

G_DEFINE_TYPE_WITH_PRIVATE(ExampleAppWindow, example_app_window,
GTK_TYPE_APPLICATION_WINDOW);

...

static void
example_app_window_class_init (ExampleAppWindowClass *class)
{
    gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
                                                "/org/gtk/exampleapp/window.ui");
    gtk_widget_class_bind_template_child_private (GTK_WIDGET_CLASS (class),
ExampleAppWindow, stack);
}

...
```

[\(full source\)](#)

Now we revisit the `example_app_window_open()` function that is called for each commandline argument, and construct a `GtkTextView` that we then add as a page to the stack:

```
...

void
example_app_window_open (ExampleAppWindow *win,
                        GFile             *file)
{
    ExampleAppWindowPrivate *priv;
    gchar *basename;
    GtkWidget *scrolled, *view;
    gchar *contents;
    gsize length;

    priv = example_app_window_get_instance_private (win);
    basename = g_file_get_basename (file);

    scrolled = gtk_scrolled_window_new (NULL, NULL);
    gtk_widget_show (scrolled);
    gtk_widget_set_hexpand (scrolled, TRUE);
    gtk_widget_set_vexpand (scrolled, TRUE);
    view = gtk_text_view_new ();
    gtk_text_view_set_editable (GTK_TEXT_VIEW (view), FALSE);
    gtk_text_view_set_cursor_visible (GTK_TEXT_VIEW (view), FALSE);
    gtk_widget_show (view);
    gtk_container_add (GTK_CONTAINER (scrolled), view);
    gtk_stack_add_titled (GTK_STACK (priv->stack), scrolled, basename, basename);

    if (g_file_load_contents (file, NULL, &contents, &length, NULL, NULL))
    {
        GtkTextBuffer *buffer;

        buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (view));
        gtk_text_buffer_set_text (buffer, contents, length);
        g_free (contents);
    }

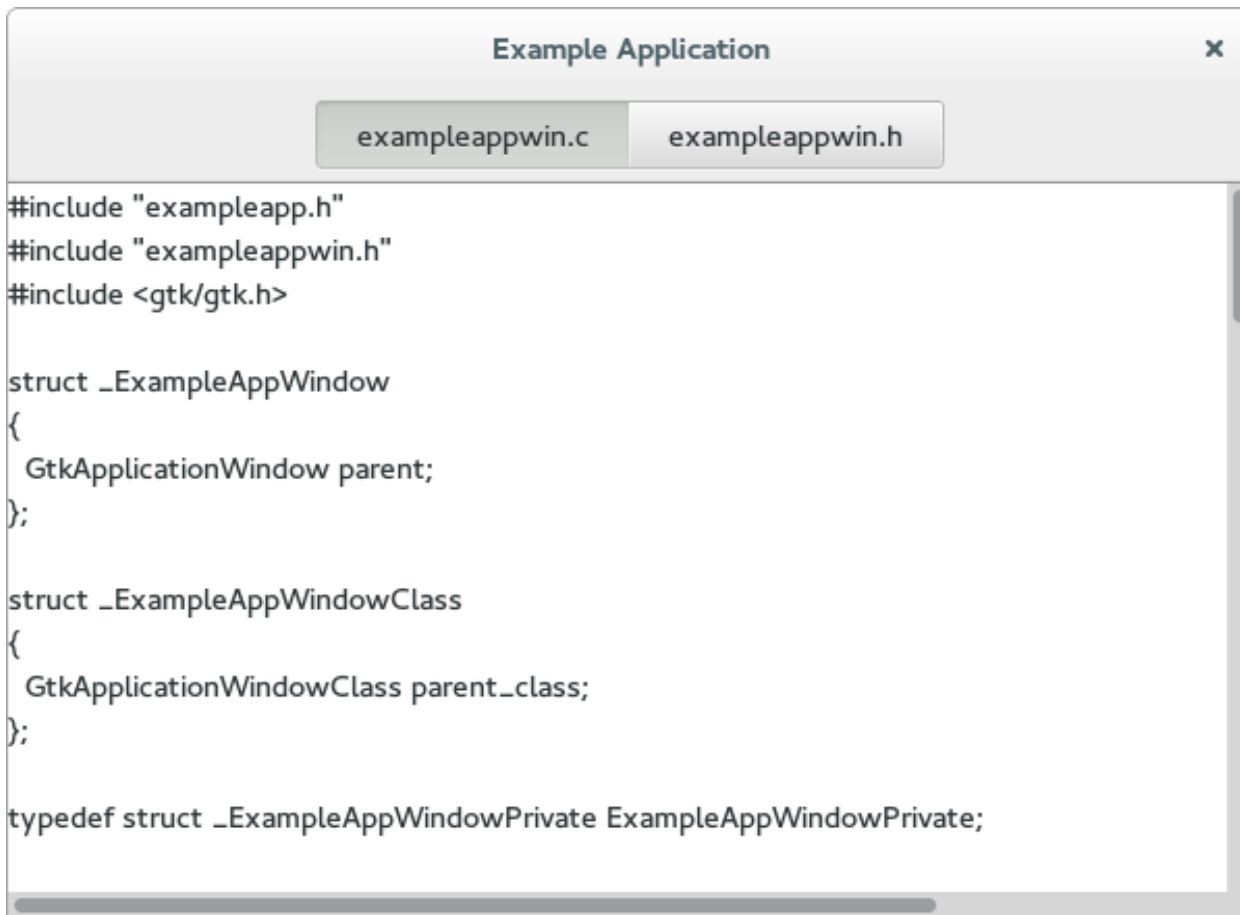
    g_free (basename);
}

...
```

[\(full source\)](#)

Note that we did not have to touch the stack switcher at all. It gets all its information from the stack that it belongs to. Here, we are passing the label to show for each file as the last argument to the [gtk_stack_add_titled\(\)](#) function.

Our application is beginning to take shape:



```
Example Application x
exampleappwin.c exampleappwin.h
#include "exampleapp.h"
#include "exampleappwin.h"
#include <gtk/gtk.h>

struct _ExampleAppWindow
{
    GtkApplicationWindow parent;
};

struct _ExampleAppWindowClass
{
    GtkApplicationWindowClass parent_class;
};

typedef struct _ExampleAppWindowPrivate ExampleAppWindowPrivate;
```

An application menu

An application menu is shown by GNOME shell at the top of the screen. It is meant to collect infrequently used actions that affect the whole application.

Just like the window template, we specify our application menu in a ui file, and add it as a resource to our binary.

```
1 <?xml version="1.0"?>
2 <interface>
3   <!-- interface-requires gtk+ 3.0 -->
4   <menu id="appmenu">
5     <section>
6       <item>
7         <attribute name="label" translatable="yes">_Preferences</attribute>
8         <attribute name="action">app.preferences</attribute>
9       </item>
10    </section>
11    <section>
12      <item>
13        <attribute name="label" translatable="yes">_Quit</attribute>
14        <attribute name="action">app.quit</attribute>
15      </item>
16    </section>
17  </menu>
18 </interface>
```

To associate the app menu with the application, we have to call [gtk_application_set_app_menu\(\)](#). Since app menus work by activating GActions, we also have to add a suitable set of actions to our application.

Both of these tasks are best done in the `startup()` vfunc, which is guaranteed to be called once for each

primary application instance:

```
...

static void
preferences_activated (GSimpleAction *action,
                      GVariant      *parameter,
                      gpointer       app)
{
}

static void
quit_activated (GSimpleAction *action,
               GVariant      *parameter,
               gpointer       app)
{
    g_application_quit (G_APPLICATION (app));
}

static GActionEntry app_entries[] =
{
    { "preferences", preferences_activated, NULL, NULL, NULL },
    { "quit", quit_activated, NULL, NULL, NULL }
};

static void
example_app_startup (GApplication *app)
{
    GtkBuilder *builder;
    GMenuModel *app_menu;
    const gchar *quit_accels[2] = { "<Ctrl>Q", NULL };

    G_APPLICATION_CLASS (example_app_parent_class)->startup (app);

    g_action_map_add_action_entries (G_ACTION_MAP (app),
                                    app_entries, G_N_ELEMENTS (app_entries),
                                    app);
    gtk_application_set_accels_for_action (GTK_APPLICATION (app),
                                           "app.quit",
                                           quit_accels);

    builder = gtk_builder_new_from_resource ("/org/gtk/exampleapp/app-menu.ui");
    app_menu = G_MENU_MODEL (gtk_builder_get_object (builder, "appmenu"));
    gtk_application_set_app_menu (GTK_APPLICATION (app), app_menu);
    g_object_unref (builder);
}

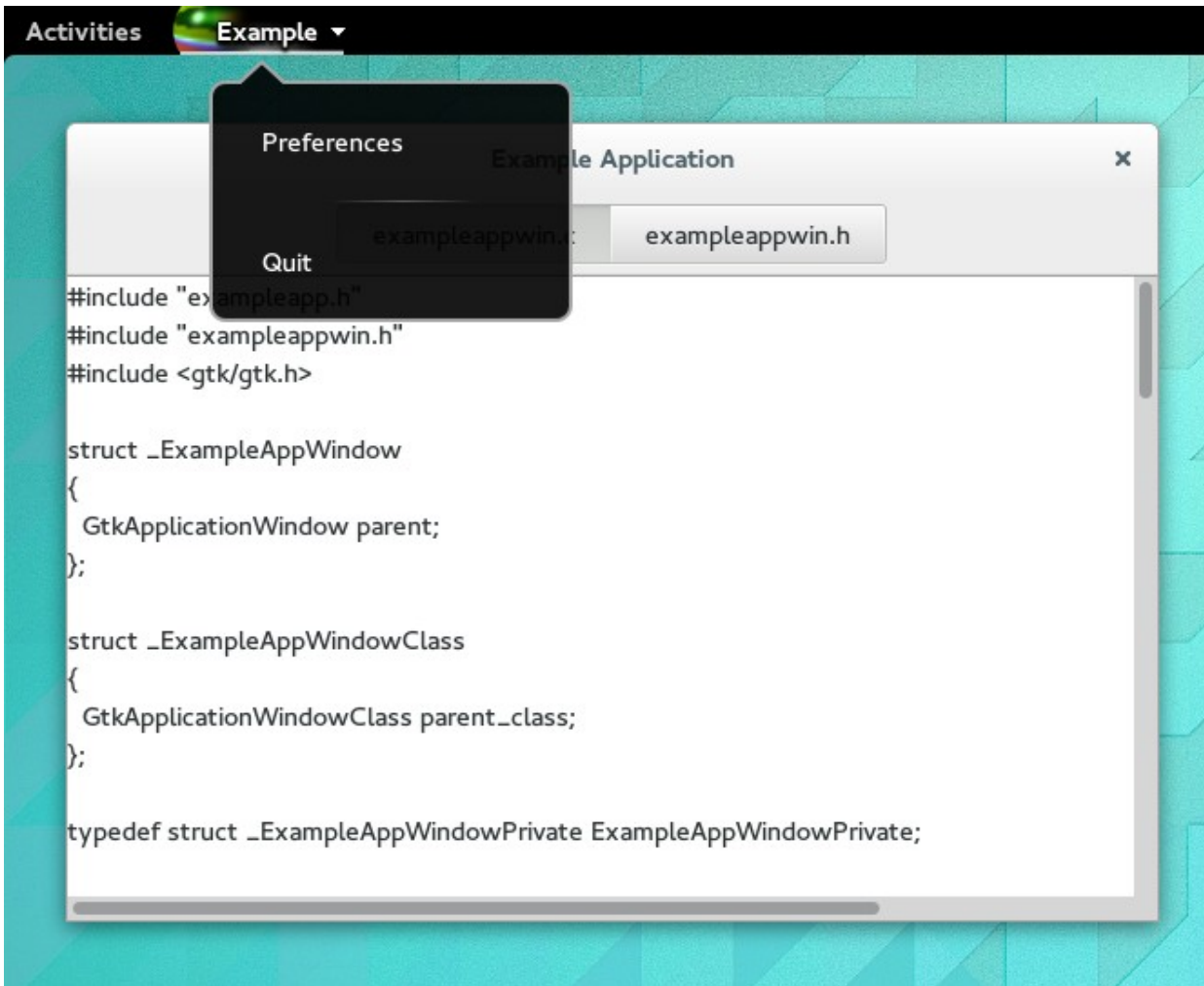
static void
example_app_class_init (ExampleAppClass *class)
{
    G_APPLICATION_CLASS (class)->startup = example_app_startup;
    ...
}

...
```

[\(full source\)](#)

Our preferences menu item does not do anything yet, but the Quit menu item is fully functional. Note that it can also be activated by the usual Ctrl-Q shortcut. The shortcut was added with [gtk_application_set_accels_for_action\(\)](#).

The application menu looks like this:



A preference dialog

A typical application will have a some preferences that should be remembered from one run to the next. Even for our simple example application, we may want to change the font that is used for the content.

We are going to use GSettings to store our preferences. GSettings requires a schema that describes our settings:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <schemalist>
3   <schema path="/org/gtk/exampleapp/" id="org.gtk.exampleapp">
4     <key name="font" type="s">
5       <default>'Monospace 12'</default>
6       <summary>Font</summary>
7       <description>The font to be used for content.</description>
8     </key>
9     <key name="transition" type="s">
10      <choices>
11        <choice value='none' />
12        <choice value='crossfade' />
13        <choice value='slide-left-right' />
14      </choices>
15      <default>'none'</default>
```

```

16     <summary>Transition</summary>
17     <description>The transition to use when switching tabs.</description>
18 </key>
19 </schema>
20 </schemalist>

```

Before we can make use of this schema in our application, we need to compile it into the binary form that GSettings expects. GIO provides [macros](#) to do this in autotools-based projects.

Next, we need to connect our settings to the widgets that they are supposed to control. One convenient way to do this is to use GSettings bind functionality to bind settings keys to object properties, as we do here for the transition setting.

```

...

static void
example_app_window_init (ExampleAppWindow *win)
{
    ExampleAppWindowPrivate *priv;

    priv = example_app_window_get_instance_private (win);
    gtk_widget_init_template (GTK_WIDGET (win));
    priv->settings = g_settings_new ("org.gtk.exampleapp");

    g_settings_bind (priv->settings, "transition",
                    priv->stack, "transition-type",
                    G_SETTINGS_BIND_DEFAULT);
}

```

...

[\(full source\)](#)

The code to connect the font setting is a little more involved, since there is no simple object property that it corresponds to, so we are not going to go into that here.

At this point, the application will already react if you change one of the settings, e.g. using the gsettings commandline tool. Of course, we expect the application to provide a preference dialog for these. So lets do that now. Our preference dialog will be a subclass of GtkDialog, and we'll use the same techniques that we've already seen: templates, private structs, settings bindings.

Lets start with the template.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <interface>
3 <!-- interface-requires gtk+ 3.8 -->
4 <template class="ExampleAppPrefs" parent="GtkDialog">
5 <property name="title" translatable="yes">Preferences</property>
6 <property name="resizable">False</property>
7 <property name="modal">True</property>
8 <child internal-child="vbox">
9 <object class="GtkBox" id="vbox">
10 <child>
11 <object class="GtkGrid" id="grid">
12 <property name="visible">True</property>
13 <property name="margin">6</property>
14 <property name="row-spacing">12</property>
15 <property name="column-spacing">6</property>
16 <child>
17 <object class="GtkLabel" id="fontlabel">
18 <property name="visible">True</property>
19 <property name="label">_Font:</property>
20 <property name="use-underline">True</property>

```

```

21         <property name="mnemonic-widget">font</property>
22         <property name="xalign">1</property>
23     </object>
24     <packing>
25         <property name="left-attach">0</property>
26         <property name="top-attach">0</property>
27     </packing>
28 </child>
29 <child>
30     <object class="GtkFontButton" id="font">
31         <property name="visible">True</property>
32     </object>
33     <packing>
34         <property name="left-attach">1</property>
35         <property name="top-attach">0</property>
36     </packing>
37 </child>
38 <child>
39     <object class="GtkLabel" id="transitionlabel">
40         <property name="visible">True</property>
41         <property name="label">_Transition:</property>
42         <property name="use-underline">True</property>
43         <property name="mnemonic-widget">transition</property>
44         <property name="xalign">1</property>
45     </object>
46     <packing>
47         <property name="left-attach">0</property>
48         <property name="top-attach">1</property>
49     </packing>
50 </child>
51 <child>
52     <object class="GtkComboBoxText" id="transition">
53         <property name="visible">True</property>
54         <items>
55             <item translatable="yes" id="none">None</item>
56             <item translatable="yes" id="crossfade">Fade</item>
57             <item translatable="yes" id="slide-left-right">Slide</item>
58         </items>
59     </object>
60     <packing>
61         <property name="left-attach">1</property>
62         <property name="top-attach">1</property>
63     </packing>
64 </child>
65 </object>
66 </child>
67 </object>
68 </child>
69 </template>
70 </interface>

```

Next comes the dialog subclass.

```

1 #include <gtk/gtk.h>
2
3 #include "exampleapp.h"
4 #include "exampleappwin.h"
5 #include "exampleappprefs.h"
6
7 struct _ExampleAppPrefs
8 {
9     GtkWidget parent;

```

```

10 };
11
12 typedef struct _ExampleAppPrefsPrivate ExampleAppPrefsPrivate;
13
14 struct _ExampleAppPrefsPrivate
15 {
16     GSettings *settings;
17     GtkWidget *font;
18     GtkWidget *transition;
19 };
20
21 G_DEFINE_TYPE_WITH_PRIVATE(ExampleAppPrefs, example_app_prefs, GTK_TYPE_DIALOG)
22
23 static void
24 example_app_prefs_init (ExampleAppPrefs *prefs)
25 {
26     ExampleAppPrefsPrivate *priv;
27
28     priv = example_app_prefs_get_instance_private (prefs);
29     gtk_widget_init_template (GTK_WIDGET (prefs));
30     priv->settings = g_settings_new ("org.gtk.exampleapp");
31
32     g_settings_bind (priv->settings, "font",
33                     priv->font, "font",
34                     G_SETTINGS_BIND_DEFAULT);
35     g_settings_bind (priv->settings, "transition",
36                     priv->transition, "active-id",
37                     G_SETTINGS_BIND_DEFAULT);
38 }
39
40 static void
41 example_app_prefs_dispose (GObject *object)
42 {
43     ExampleAppPrefsPrivate *priv;
44
45     priv = example_app_prefs_get_instance_private (EXAMPLE_APP_PREFS (object));
46     g_clear_object (&priv->settings);
47
48     G_OBJECT_CLASS (example_app_prefs_parent_class)->dispose (object);
49 }
50
51 static void
52 example_app_prefs_class_init (ExampleAppPrefsClass *class)
53 {
54     G_OBJECT_CLASS (class)->dispose = example_app_prefs_dispose;
55
56     gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
57                                                 "/org/gtk/exampleapp/prefs.ui");
58     gtk_widget_class_bind_template_child_private (GTK_WIDGET_CLASS (class),
59 ExampleAppPrefs, font);
60     gtk_widget_class_bind_template_child_private (GTK_WIDGET_CLASS (class),
61 ExampleAppPrefs, transition);
62 }
63
64 ExampleAppPrefs *
65 example_app_prefs_new (ExampleAppWindow *win)
66 {
67     return g_object_new (EXAMPLE_APP_PREFS_TYPE, "transient-for", win, "use-header-bar",
68 TRUE, NULL);
69 }

```

Now we revisit the `preferences_activated()` function in our application class, and make it open a new preference dialog.


```

static void
preferences_activated (GSimpleAction *action,
                      GVariant      *parameter,
                      gpointer       app)
{
    ExampleAppPrefs *prefs;
    GtkWidget *win;

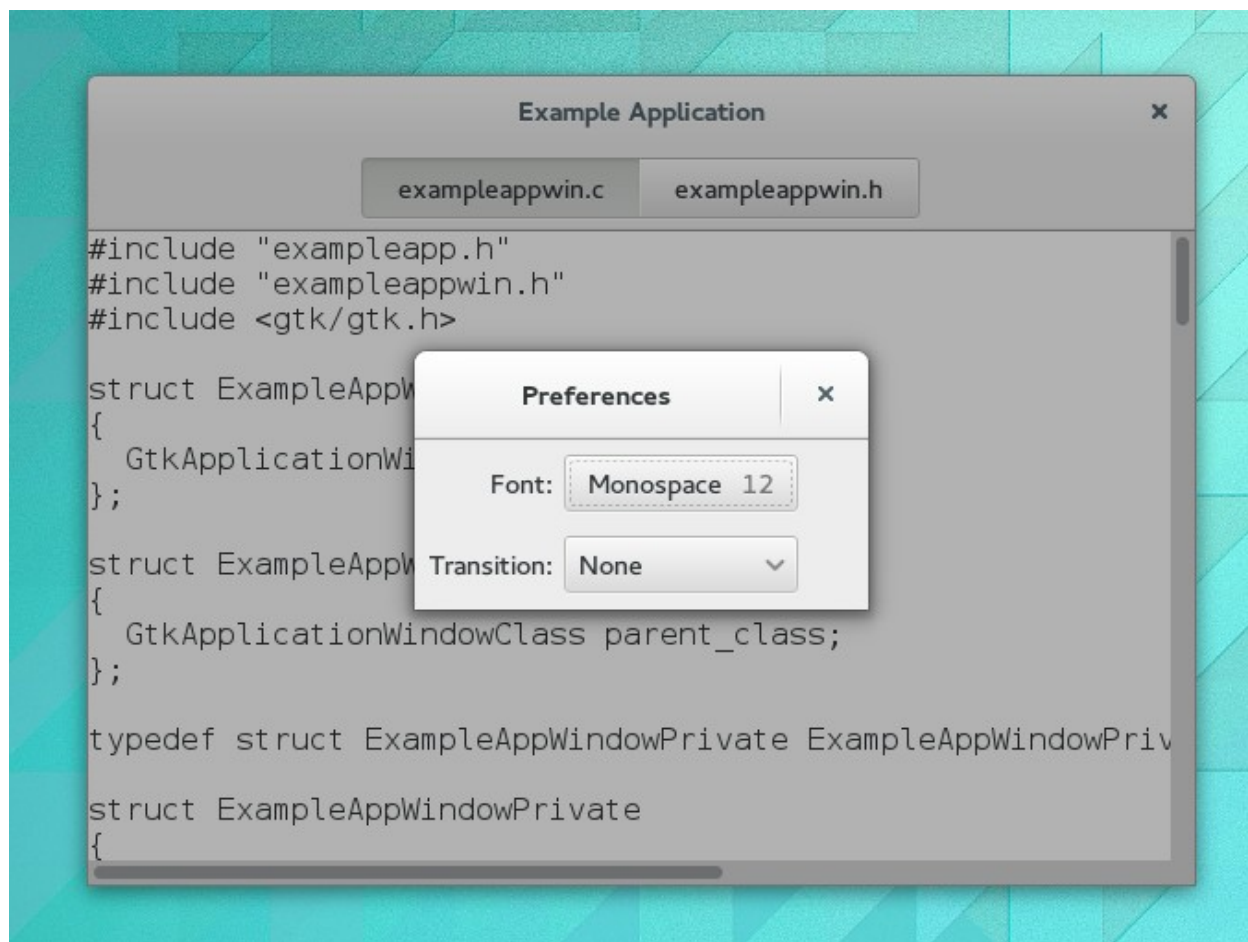
    win = gtk_application_get_active_window (GTK_APPLICATION (app));
    prefs = example_app_prefs_new (EXAMPLE_APP_WINDOW (win));
    gtk_window_present (GTK_WINDOW (prefs));
}

...

```

[\(full source\)](#)

After all this work, our application can now show a preference dialog like this:



Adding a search bar

We continue to flesh out the functionality of our application. For now, we add search. GTK+ supports this with [GtkSearchEntry](#) and [GtkSearchBar](#). The search bar is a widget that can slide in from the top to present a search entry.

We add a toggle button to the header bar, which can be used to slide out the search bar below the header bar.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <interface>
3   <!-- interface-requires gtk+ 3.8 -->
4   <template class="ExampleAppWindow" parent="GtkApplicationWindow">
5     <property name="title" translatable="yes">Example Application</property>
6     <property name="default-width">600</property>
7     <property name="default-height">400</property>
8     <child>
9       <object class="GtkBox" id="content_box">
10        <property name="visible">True</property>
11        <property name="orientation">vertical</property>
12        <child>
13          <object class="GtkHeaderBar" id="header">
14            <property name="visible">True</property>
15            <child type="title">
16              <object class="GtkStackSwitcher" id="tabs">
17                <property name="visible">True</property>
18                <property name="stack">stack</property>
19              </object>
20            </child>
21            <child>
22              <object class="GtkToggleButton" id="search">
23                <property name="visible">True</property>
24                <property name="sensitive">False</property>
25                <style>
26                  <class name="image-button"/>
27                </style>
28                <child>
29                  <object class="GtkImage" id="search-icon">
30                    <property name="visible">True</property>
31                    <property name="icon-name">edit-find-symbolic</property>
32                    <property name="icon-size">1</property>
33                  </object>
34                </child>
35              </object>
36              <packing>
37                <property name="pack-type">end</property>
38              </packing>
39            </child>
40          </object>
41        </child>
42        <child>
43          <object class="GtkSearchBar" id="searchbar">
44            <property name="visible">True</property>
45            <child>
46              <object class="GtkSearchEntry" id="searchentry">
47                <signal name="search-changed" handler="search_text_changed"/>
48                <property name="visible">True</property>
49              </object>
50            </child>
51          </object>
52        </child>
53        <child>
54          <object class="GtkStack" id="stack">
```

```

55         <signal name="notify::visible-child" handler="visible_child_changed"/>
56         <property name="visible">True</property>
57     </object>
58 </child>
59 </object>
60 </child>
61 </template>
62 </interface>

```

Implementing the search needs quite a few code changes that we are not going to completely go over here. The central piece of the search implementation is a signal handler that listens for text changes in the search entry.

```

...

static void
search_text_changed (GtkEntry      *entry,
                    ExampleAppWindow *win)
{
    ExampleAppWindowPrivate *priv;
    const gchar *text;
    GtkWidget *tab;
    GtkWidget *view;
    GtkTextBuffer *buffer;
    GtkTextIter start, match_start, match_end;

    text = gtk_entry_get_text (entry);

    if (text[0] == '\0')
        return;

    priv = example_app_window_get_instance_private (win);

    tab = gtk_stack_get_visible_child (GTK_STACK (priv->stack));
    view = gtk_bin_get_child (GTK_BIN (tab));
    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (view));

    /* Very simple-minded search implementation */
    gtk_text_buffer_get_start_iter (buffer, &start);
    if (gtk_text_iter_forward_search (&start, text, GTK_TEXT_SEARCH_CASE_INSENSITIVE,
                                     &match_start, &match_end, NULL))
    {
        gtk_text_buffer_select_range (buffer, &match_start, &match_end);
        gtk_text_view_scroll_to_iter (GTK_TEXT_VIEW (view), &match_start,
                                     0.0, FALSE, 0.0, 0.0);
    }
}

static void
example_app_window_init (ExampleAppWindow *win)
{
    ...

    gtk_widget_class_bind_template_callback (GTK_WIDGET_CLASS (class), search_text_changed);

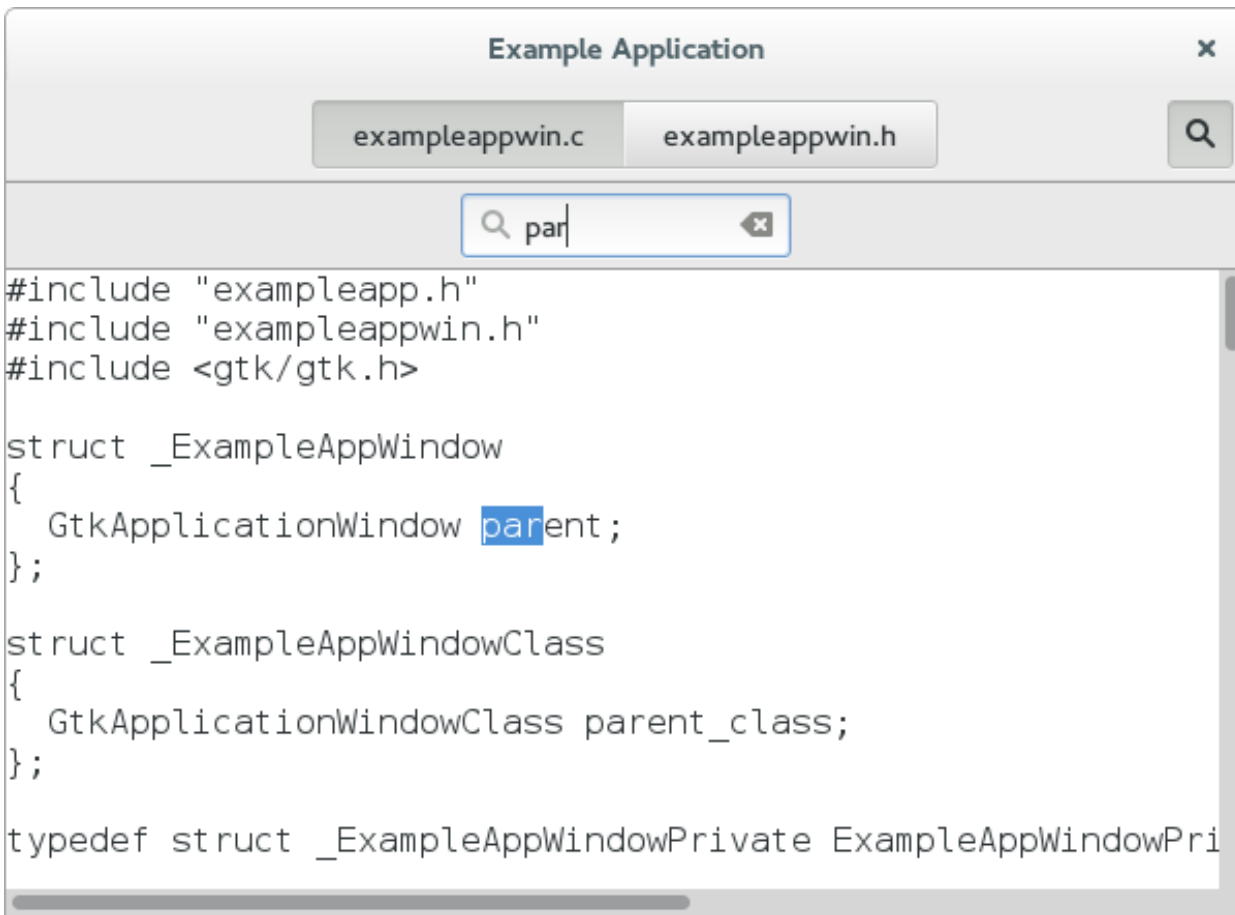
    ...
}

...

```

[\(full source\)](#)

With the search bar, our application now looks like this:



Adding a side bar

As another piece of functionality, we are adding a sidebar, which demonstrates [GtkMenuButton](#), [GtkRevealer](#) and [GtkListBox](#).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <interface>
3   <!-- interface-requires gtk+ 3.8 -->
4   <template class="ExampleAppWindow" parent="GtkApplicationWindow">
5     <property name="title" translatable="yes">Example Application</property>
6     <property name="default-width">600</property>
7     <property name="default-height">400</property>
8     <child>
9       <object class="GtkBox" id="content_box">
10        <property name="visible">True</property>
11        <property name="orientation">vertical</property>
12        <child>
13          <object class="GtkHeaderBar" id="header">
14            <property name="visible">True</property>
15            <child type="title">
16              <object class="GtkStackSwitcher" id="tabs">
17                <property name="visible">True</property>
18                <property name="stack">stack</property>
19              </object>
20            </child>
21          </child>
22          <object class="GtkToggleButton" id="search">
23            <property name="visible">True</property>
24            <property name="sensitive">False</property>
25          </style>
```

```
26         <class name="image-button"/>
27     </style>
28     <child>
29         <object class="GtkImage" id="search-icon">
30             <property name="visible">True</property>
31             <property name="icon-name">edit-find-symbolic</property>
32             <property name="icon-size">1</property>
33         </object>
34     </child>
35 </object>
36 <packing>
37     <property name="pack-type">end</property>
38 </packing>
39 </child>
40 <child>
41     <object class="GtkMenuButton" id="gears">
42         <property name="visible">True</property>
43         <property name="direction">none</property>
44         <property name="use-popover">True</property>
45         <style>
46             <class name="image-button"/>
47         </style>
48     </object>
49     <packing>
50         <property name="pack-type">end</property>
51     </packing>
52 </child>
53 </object>
54 </child>
55 <child>
56     <object class="GtkSearchBar" id="searchbar">
57         <property name="visible">True</property>
58         <child>
59             <object class="GtkSearchEntry" id="searchentry">
60                 <signal name="search-changed" handler="search_text_changed"/>
61                 <property name="visible">True</property>
62             </object>
63         </child>
64     </object>
65 </child>
66 <child>
67     <object class="GtkBox" id="hbox">
68         <property name="visible">True</property>
69         <child>
70             <object class="GtkRevealer" id="sidebar">
71                 <property name="visible">True</property>
72                 <property name="transition-type">slide-right</property>
73                 <child>
74                     <object class="GtkScrolledWindow" id="sidebar-sw">
75                         <property name="visible">True</property>
76                         <property name="hscrollbar-policy">never</property>
77                         <property name="vscrollbar-policy">automatic</property>
78                         <child>
79                             <object class="GtkListBox" id="words">
80                                 <property name="visible">True</property>
81                                 <property name="selection-mode">none</property>
82                             </object>
83                         </child>
84                     </object>
85                 </child>
86             </object>
87         </child>
88     </child>
```

```

89         <object class="GtkStack" id="stack">
90             <signal name="notify::visible-child" handler="visible_child_changed"/>
91             <property name="visible">True</property>
92         </object>
93     </child>
94 </object>
95 </child>
96 </object>
97 </child>
98 </template>
99 </interface>

```

The code to populate the sidebar with buttons for the words found in each file is a little too involved to go into here. But we'll look at the code to add the gears menu.

As expected by now, the gears menu is specified in a GtkBuilder ui file.

```

1  <?xml version="1.0"?>
2  <interface>
3      <!-- interface-requires gtk+ 3.0 -->
4      <menu id="menu">
5          <section>
6              <item>
7                  <attribute name="label" translatable="yes">_Words</attribute>
8                  <attribute name="action">win.show-words</attribute>
9              </item>
10         </section>
11     </menu>
12 </interface>

```

To connect the menuitem to the show-words setting, we use a GAction corresponding to the given GSettings key.

```

...

static void
example_app_window_init (ExampleAppWindow *win)
{
    ...

    builder = gtk_builder_new_from_resource ("/org/gtk/exampleapp/gears-menu.ui");
    menu = G_MENU_MODEL (gtk_builder_get_object (builder, "menu"));
    gtk_menu_button_set_menu_model (GTK_MENU_BUTTON (priv->gears), menu);
    g_object_unref (builder);

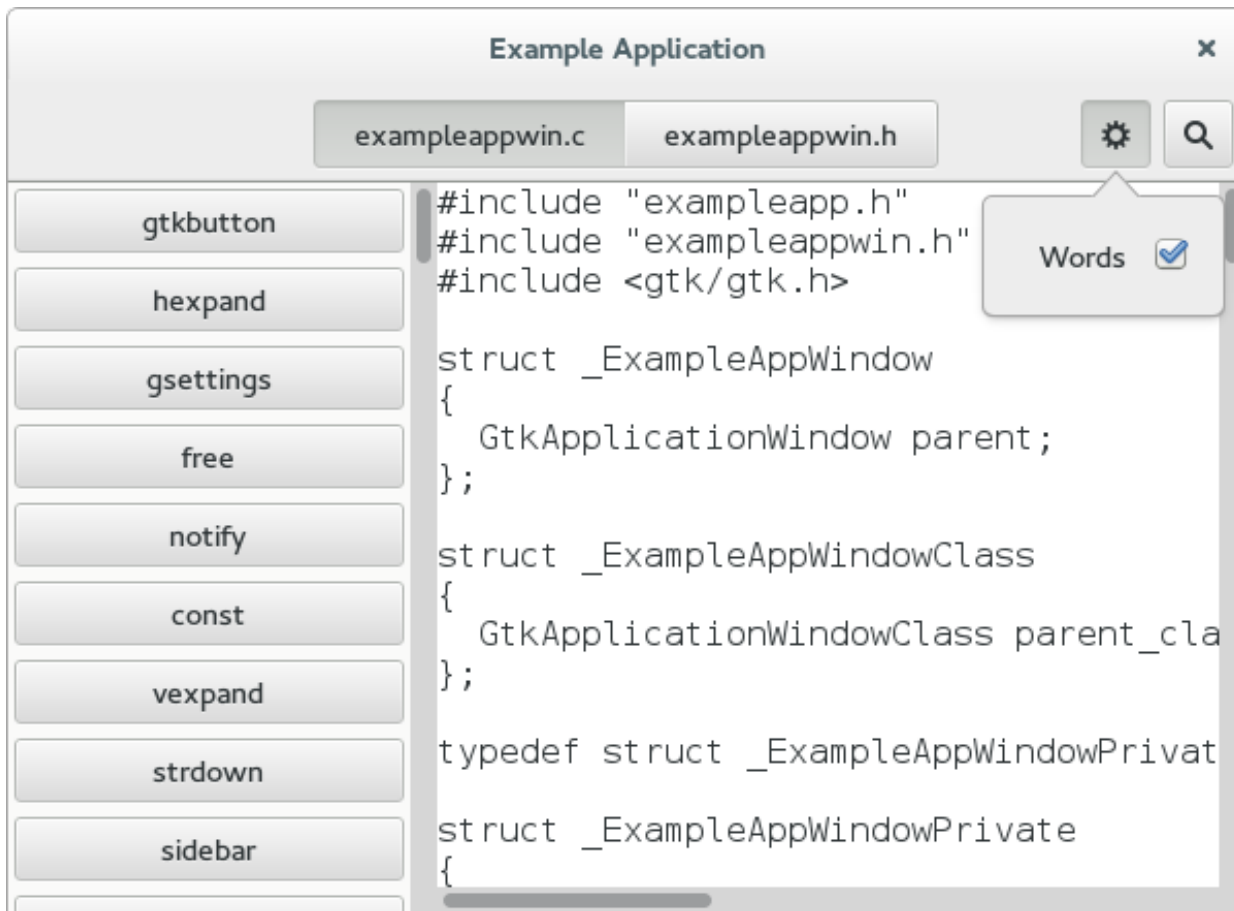
    action = g_settings_create_action (priv->settings, "show-words");
    g_action_map_add_action (G_ACTION_MAP (win), action);
    g_object_unref (action);
}

...

```

[\(full source\)](#)

What our application looks like now:



Properties

Widgets and other objects have many useful properties.

Here we show some ways to use them in new and flexible ways, by wrapping them in actions with `GPropertyAction` or by binding them with `GBinding`.

To set this up, we add two labels to the header bar in our window template, named `lines_label` and `lines`, and bind them to struct members in the private struct, as we've seen a couple of times by now.

We add a new "Lines" menu item to the gears menu, which triggers the show-lines action:

```
1 <?xml version="1.0"?>
2 <interface>
3   <!-- interface-requires gtk+ 3.0 -->
4   <menu id="menu">
5     <section>
6       <item>
7         <attribute name="label" translatable="yes">_Words</attribute>
8         <attribute name="action">win.show-words</attribute>
9       </item>
10      <item>
11        <attribute name="label" translatable="yes">_Lines</attribute>
12        <attribute name="action">win.show-lines</attribute>
13      </item>
14    </section>
15  </menu>
16 </interface>
```

To make this menu item do something, we create a property action for the visible property of the `lines` label, and add it to the actions of the window. The effect of this is that the visibility of the label gets toggled every time the action is activated.

Since we want both labels to appear and disappear together, we bind the visible property of the `lines_label` widget to the same property of the `lines` widget.

```
...
static void
example_app_window_init (ExampleAppWindow *win)
{
    ...

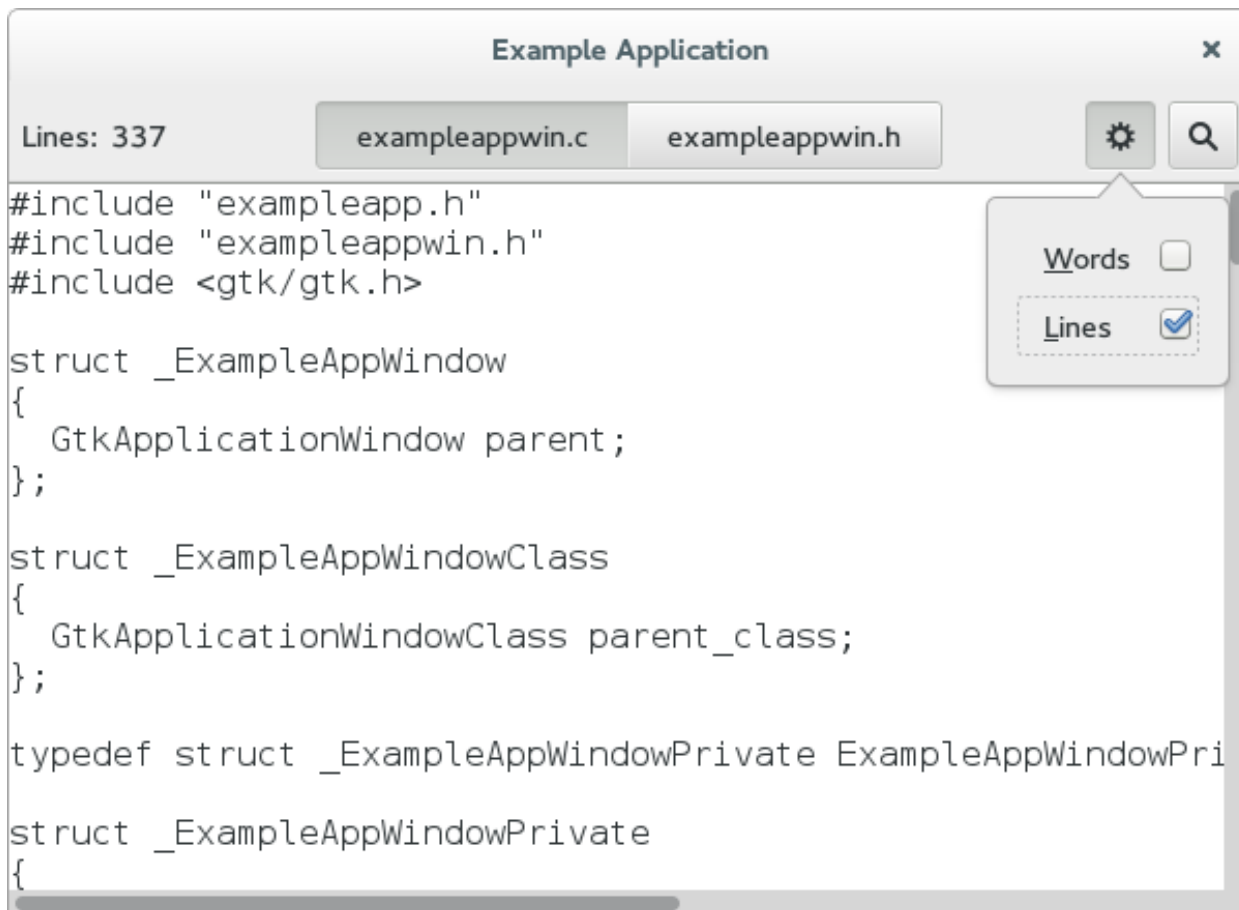
    action = (GAction*) g_property_action_new ("show-lines", priv->lines, "visible");
    g_action_map_add_action (G_ACTION_MAP (win), action);
    g_object_unref (action);

    g_object_bind_property (priv->lines, "visible",
                           priv->lines_label, "visible",
                           G_BINDING_DEFAULT);
}
...
```

[\(full source\)](#)

We also need a function that counts the lines of the currently active tab, and updates the `lines` label. See the [full source](#) if you are interested in the details.

This brings our example application to this appearance:



Header bar

Our application already uses a `GtkHeaderBar`, but so far it still gets a 'normal' window titlebar on top of that. This is a bit redundant, and we will now tell GTK+ to use the header bar as replacement for the titlebar. To do so, we move it around to be a direct child of the window, and set its type to be titlebar.

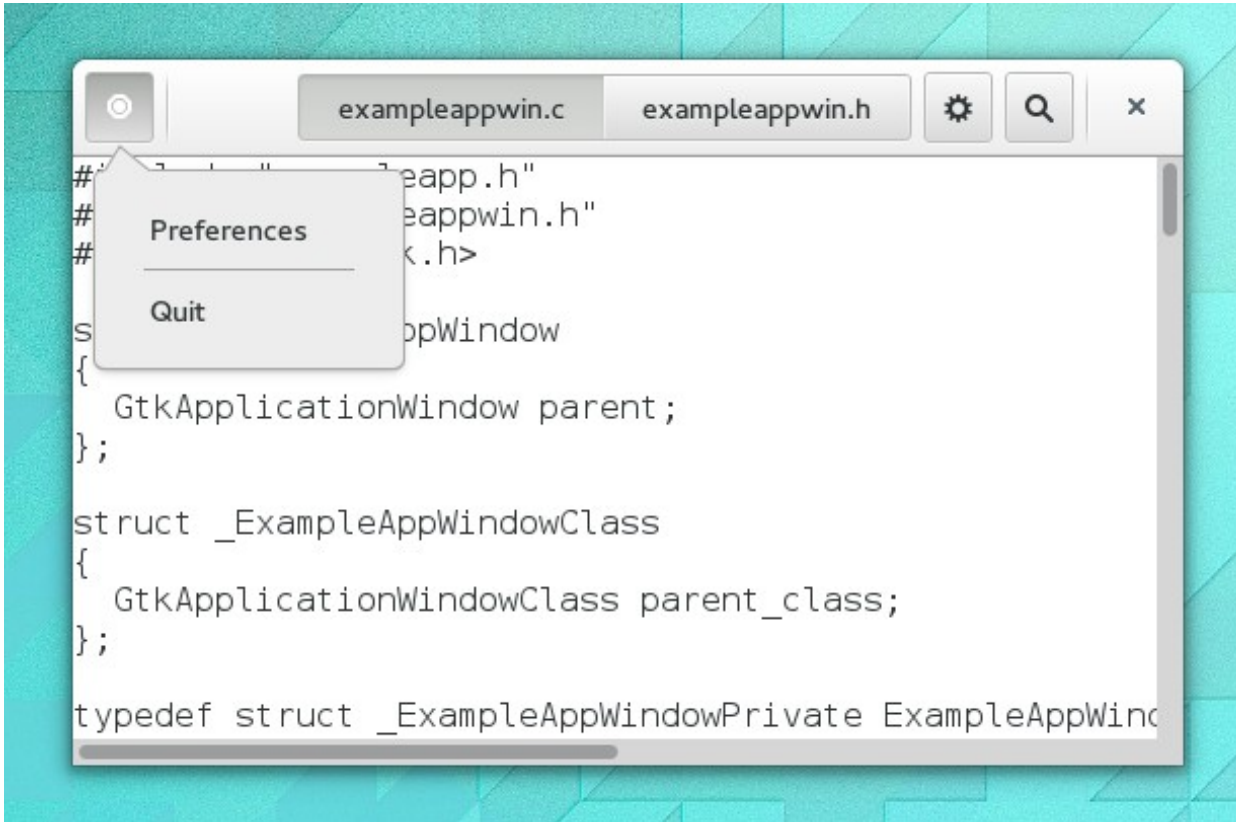
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
3    <!-- interface-requires gtk+ 3.8 -->
4    <template class="ExampleAppWindow" parent="GtkApplicationWindow">
5      <property name="title" translatable="yes">Example Application</property>
6      <property name="default-width">600</property>
7      <property name="default-height">400</property>
8      <child type="titlebar">
9        <object class="GtkHeaderBar" id="header">
10         <property name="visible">True</property>
11         <property name="show-close-button">True</property>
12         <child>
13           <object class="GtkLabel" id="lines_label">
14             <property name="visible">False</property>
15             <property name="label" translatable="yes">Lines:</property>
16           </object>
17           <packing>
18             <property name="pack-type">start</property>
19           </packing>
20         </child>
21         <child>
22           <object class="GtkLabel" id="lines">
23             <property name="visible">False</property>
24           </object>
25           <packing>
26             <property name="pack-type">start</property>
27           </packing>
28         </child>
29         <child type="title">
30           <object class="GtkStackSwitcher" id="tabs">
31             <property name="visible">True</property>
32             <property name="stack">stack</property>
33           </object>
34         </child>
35         <child>
36           <object class="GtkToggleButton" id="search">
37             <property name="visible">True</property>
38             <property name="sensitive">False</property>
39             <style>
40               <class name="image-button"/>
41             </style>
42             <child>
43               <object class="GtkImage" id="search-icon">
44                 <property name="visible">True</property>
45                 <property name="icon-name">edit-find-symbolic</property>
46                 <property name="icon-size">1</property>
47               </object>
48             </child>
49           </object>
50           <packing>
51             <property name="pack-type">end</property>
52           </packing>
53         </child>
54         <child>
55           <object class="GtkMenuButton" id="gears">
```

```

56         <property name="visible">True</property>
57         <property name="direction">none</property>
58         <property name="use-popover">True</property>
59         <style>
60             <class name="image-button"/>
61         </style>
62     </object>
63     <packing>
64         <property name="pack-type">end</property>
65     </packing>
66 </child>
67 </object>
68 </child>
69 <child>
70     <object class="GtkBox" id="content_box">
71         <property name="visible">True</property>
72         <property name="orientation">vertical</property>
73         <child>
74             <object class="GtkSearchBar" id="searchbar">
75                 <property name="visible">True</property>
76                 <child>
77                     <object class="GtkSearchEntry" id="searchentry">
78                         <signal name="search-changed" handler="search_text_changed"/>
79                         <property name="visible">True</property>
80                     </object>
81                 </child>
82             </object>
83         </child>
84         <child>
85             <object class="GtkBox" id="hbox">
86                 <property name="visible">True</property>
87                 <child>
88                     <object class="GtkRevealer" id="sidebar">
89                         <property name="visible">True</property>
90                         <property name="transition-type">slide-right</property>
91                         <child>
92                             <object class="GtkScrolledWindow" id="sidebar-sw">
93                                 <property name="visible">True</property>
94                                 <property name="hscrollbar-policy">never</property>
95                                 <property name="vscrollbar-policy">automatic</property>
96                                 <child>
97                                     <object class="GtkListBox" id="words">
98                                         <property name="visible">True</property>
99                                         <property name="selection-mode">none</property>
100                                    </object>
101                                </child>
102                            </object>
103                        </child>
104                    </object>
105                </child>
106            <child>
107                <object class="GtkStack" id="stack">
108                    <signal name="notify::visible-child" handler="visible_child_changed"/>
109                    <property name="visible">True</property>
110                </object>
111            </child>
112        </object>
113    </child>
114 </object>
115 </child>
116 </template>
117 </interface>

```

A small extra bonus of using a header bar is that we get a fallback application menu for free. Here is how the application now looks, if this fallback is used.



If we set up the window icon for our window, the menu button will use that instead of the generic placeholder icon you see here.

Custom Drawing

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

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

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



Example 4. Drawing in response to input

Create a new file with the following content named example-4.c.

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

/* Surface to store current scribbles */
static cairo_surface_t *surface = NULL;

static void
clear_surface (void)
{
    cairo_t *cr;

    cr = cairo_create (surface);

    cairo_set_source_rgb (cr, 1, 1, 1);
    cairo_paint (cr);

    cairo_destroy (cr);
}

/* Create a new surface of the appropriate size to store our scribbles */
static gboolean
configure_event_cb (GtkWidget      *widget,
                   GdkEventConfigure *event,
                   gpointer          data)
{
    if (surface)
        cairo_surface_destroy (surface);

    surface = gdk_window_create_similar_surface (gtk_widget_get_window (widget),
                                                CAIRO_CONTENT_COLOR,
                                                gtk_widget_get_allocated_width (widget),
                                                gtk_widget_get_allocated_height (widget));

    /* Initialize the surface to white */
    clear_surface ();

    /* We've handled the configure event, no need for further processing. */
    return TRUE;
}

/* Redraw the screen from the surface. Note that the ::draw
 * signal receives a ready-to-be-used cairo_t that is already
 * clipped to only draw the exposed areas of the widget
 */
```

```

static gboolean
draw_cb (GtkWidget *widget,
        cairo_t   *cr,
        gpointer   data)
{
    cairo_set_source_surface (cr, surface, 0, 0);
    cairo_paint (cr);

    return FALSE;
}

/* Draw a rectangle on the surface at the given position */
static void
draw_brush (GtkWidget *widget,
           gdouble    x,
           gdouble    y)
{
    cairo_t *cr;

    /* Paint to the surface, where we store our state */
    cr = cairo_create (surface);

    cairo_rectangle (cr, x - 3, y - 3, 6, 6);
    cairo_fill (cr);

    cairo_destroy (cr);

    /* Now invalidate the affected region of the drawing area. */
    gtk_widget_queue_draw_area (widget, x - 3, y - 3, 6, 6);
}

/* Handle button press events by either drawing a rectangle
 * or clearing the surface, depending on which button was pressed.
 * The ::button-press signal handler receives a GdkEventButton
 * struct which contains this information.
 */
static gboolean
button_press_event_cb (GtkWidget      *widget,
                      GdkEventButton *event,
                      gpointer         data)
{
    /* paranoia check, in case we haven't gotten a configure event */
    if (surface == NULL)
        return FALSE;

    if (event->button == GDK_BUTTON_PRIMARY)
    {
        draw_brush (widget, event->x, event->y);
    }
    else if (event->button == GDK_BUTTON_SECONDARY)
    {
        clear_surface ();
        gtk_widget_queue_draw (widget);
    }

    /* We've handled the event, stop processing */
    return TRUE;
}

```

```

/* Handle motion events by continuing to draw if button 1 is
 * still held down. The ::motion-notify signal handler receives
 * a GdkEventMotion struct which contains this information.
 */
static gboolean
motion_notify_event_cb (GtkWidget      *widget,
                       GdkEventMotion *event,
                       gpointer        data)
{
    /* paranoia check, in case we haven't gotten a configure event */
    if (surface == NULL)
        return FALSE;

    if (event->state & GDK_BUTTON1_MASK)
        draw_brush (widget, event->x, event->y);

    /* We've handled it, stop processing */
    return TRUE;
}

static void
close_window (void)
{
    if (surface)
        cairo_surface_destroy (surface);
}

static void
activate (GtkApplication *app,
          gpointer        user_data)
{
    GtkWidget *window;
    GtkWidget *frame;
    GtkWidget *drawing_area;

    window = gtk_application_window_new (app);
    gtk_window_set_title (GTK_WINDOW (window), "Drawing Area");

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

    gtk_container_set_border_width (GTK_CONTAINER (window), 8);

    frame = gtk_frame_new (NULL);
    gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_IN);
    gtk_container_add (GTK_CONTAINER (window), frame);

    drawing_area = gtk_drawing_area_new ();
    /* set a minimum size */
    gtk_widget_set_size_request (drawing_area, 100, 100);

    gtk_container_add (GTK_CONTAINER (frame), drawing_area);

    /* Signals used to handle the backing surface */
    g_signal_connect (drawing_area, "draw",
                     G_CALLBACK (draw_cb), NULL);
    g_signal_connect (drawing_area, "configure-event",
                     G_CALLBACK (configure_event_cb), NULL);

    /* Event signals */
    g_signal_connect (drawing_area, "motion-notify-event",
                     G_CALLBACK (motion_notify_event_cb), NULL);
    g_signal_connect (drawing_area, "button-press-event",
                     G_CALLBACK (button_press_event_cb), NULL);
}

```

```

/* Ask to receive events the drawing area doesn't normally
 * subscribe to. In particular, we need to ask for the
 * button press and motion notify events that want to handle.
 */
gtk_widget_set_events (drawing_area, gtk_widget_get_events (drawing_area)
                        | GDK_BUTTON_PRESS_MASK
                        | GDK_POINTER_MOTION_MASK);

gtk_widget_show_all (window);
}

int
main (int   argc,
      char **argv)
{
    GtkApplication *app;
    int status;

    app = gtk_application_new ("org.gtk.example", G_APPLICATION_FLAGS_NONE);
    g_signal_connect (app, "activate", G_CALLBACK (activate), NULL);
    status = g_application_run (G_APPLICATION (app), argc, argv);
    g_object_unref (app);

    return status;
}

```

You can compile the program above with GCC using:

```
gcc `pkg-config --cflags gtk+-3.0` -o example-4 example-4.c `pkg-config --libs gtk+-3.0`
```

Mailing lists and bug reports

Mailing lists and bug reports — Getting help with GTK+

Filing a bug report or feature request

If you encounter a bug, misfeature, or missing feature in GTK+, please file a bug report on <https://bugzilla.gnome.org>. We'd also appreciate reports of incomplete or misleading information in the GTK+ documentation; file those against the "docs" component of the "gtk+" product in Bugzilla.

Don't hesitate to file a bug report, even if you think we may know about it already, or aren't sure of the details. Just give us as much information as you have, and if it's already fixed or has already been discussed, we'll add a note to that effect in the report.

The bug tracker should definitely be used for feature requests, it's not only for bugs. We track all GTK+ development in Bugzilla, so it's the way to be sure the GTK+ developers won't forget about an issue.

Submitting Patches

If you develop a bugfix or enhancement for GTK+, please file that in Bugzilla as well. Bugzilla allows you to attach files; please attach a patch generated by the **diff** utility, using the `-u` option to make the patch more readable. All patches must be offered under the terms of the GNU LGPL license, so be sure you are authorized to give us the patch under those terms.

If you want to discuss your patch before or after developing it, mail gtk-devel-list@gnome.org. But be sure to file the Bugzilla report as well; if the patch is only on the list and not in Bugzilla, it's likely to slip through the cracks.

Mailing lists

There are several mailing lists dedicated to GTK+ and related libraries. Discussion of GLib, Pango, and ATK in addition to GTK+ proper is welcome on these lists. You can subscribe or view the archives of these lists on <http://mail.gnome.org>. If you aren't subscribed to the list, any message you post to the list will be held for manual moderation, which might take some days to happen.

gtk-list@gnome.org

gtk-list covers general GTK+ topics; questions about using GTK+ in programs, GTK+ from a user standpoint, announcements of GTK+-related projects such as themes or GTK+ modules would all be on-topic. The bulk of the traffic consists of GTK+ programming questions.

gtk-app-devel-list@gnome.org

gtk-app-devel-list covers writing applications in GTK+. It's narrower in scope than gtk-list, but the two lists overlap quite a bit. gtk-app-devel-list is a good place to ask questions about GTK+ programming.

gtk-devel-list@gnome.org

gtk-devel-list is for discussion of work on GTK+ itself, it is *not* for asking questions about how to use GTK+ in applications. gtk-devel-list is appropriate for discussion of patches, bugs, proposed features, and so on.

gtk-i18n-list@gnome.org

gtk-i18n-list is for discussion of internationalization in GTK+; Pango is the main focus of the list. Questions about the details of using Pango, and discussion of proposed Pango patches or features, are all on topic.

gtk-doc-list@gnome.org

gtk-doc-list is for discussion of the gtk-doc documentation system (used to document GTK+), and for work on the GTK+ documentation.

Common Questions

Common Questions — Find answers to common questions in the GTK+ manual

Questions and Answers

This is an "index" of the reference manual organized by common "How do I..." questions. If you aren't sure which documentation to read for the question you have, this list is a good place to start.

1. General

1.1.

How do I get started with GTK+?

The GTK+ [website](#) offers some [tutorials](#) and other documentation (most of it about GTK+ 2.x, but mostly still applicable). More documentation ranging from whitepapers to online books can be found at the [GNOME developer's site](#). After studying these materials you should be well prepared to come back to this reference manual for details.

1.2.

Where can I get help with GTK+, submit a bug report, or make a feature request?

See the [documentation on this topic](#).

1.3.

How do I port from one GTK+ version to another?

See [Migrating from GTK+ 2.x to GTK+ 3](#). You may also find useful information in the documentation for specific widgets and functions.

If you have a question not covered in the manual, feel free to ask on the mailing lists and please [file a bug report](#) against the documentation.

1.4.

How does memory management work in GTK+? Should I free data returned from functions?

See the documentation for GObject and GInitiallyUnowned. For GObject note specifically `g_object_ref()` and `g_object_unref()`. GInitiallyUnowned is a subclass of GObject so the same points apply, except that it has a "floating" state (explained in its documentation).

For strings returned from functions, they will be declared "const" if they should not be freed. Non-const strings should be freed with `g_free()`. Arrays follow the same rule. If you find an undocumented exception to the rules, please report a bug to <https://bugzilla.gnome.org>.

1.5.

Why does my program leak memory, if I destroy a widget immediately after creating it ?

If GtkFoo isn't a toplevel window, then

```
1 foo = gtk_foo_new ();
```

```
2  gtk_widget_destroy (foo);
```

is a memory leak, because no one assumed the initial floating reference. If you are using a widget and you aren't immediately packing it into a container, then you probably want standard reference counting, not floating reference counting.

To get this, you must acquire a reference to the widget and drop the floating reference (“ref and sink” in GTK+ parlance) after creating it:

```
1  foo = gtk_foo_new ();
2  g_object_ref_sink (foo);
```

When you want to get rid of the widget, you must call [gtk_widget_destroy\(\)](#) to break any external connections to the widget before dropping your reference:

```
1  gtk_widget_destroy (foo);
2  g_object_unref (foo);
```

When you immediately add a widget to a container, it takes care of assuming the initial floating reference and you don't have to worry about reference counting at all ... just call [gtk_widget_destroy\(\)](#) to get rid of the widget.

1.6.

How do I use GTK+ with threads?

This is covered in the GDK threads documentation. See also the GThread documentation for portable threading primitives.

1.7.

How do I internationalize a GTK+ program?

Most people use [GNU gettext](#), already required in order to install GLib. On a UNIX or Linux system with gettext installed, type `info gettext` to read the documentation.

The short checklist on how to use gettext is: call `bindtextdomain()` so gettext can find the files containing your translations, call `textdomain()` to set the default translation domain, call `bind_textdomain_codeset()` to request that all translated strings are returned in UTF-8, then call `gettext()` to look up each string to be translated in the default domain.

`gi18n.h` provides the following shorthand macros for convenience. Conventionally, people define macros as follows for convenience:

```
1  #define _(x)      gettext (x)
2  #define N_(x)    x
3  #define C_(ctx,x) pgettext (ctx, x)
```

You use `N_()` (N stands for no-op) to mark a string for translation in a location where a function call to `gettext()` is not allowed, such as in an array initializer. You eventually have to call `gettext()` on the string to actually fetch the translation. `_()` both marks the string for translation and actually translates it. The `C_()` macro (C stands for context) adds an additional context to the string that is marked for translation, which can help to disambiguate short strings that might need different translations in different parts of your program.

Code using these macros ends up looking like this:

```
1  #include <gi18n.h>
2
3  static const char *global_variable = N_("Translate this string");
4
5  static void
6  make_widgets (void)
7  {
8      GtkWidget *label1;
9      GtkWidget *label2;
10
11     label1 = gtk_label_new (_("Another string to translate"));
```

```
12     label2 = gtk_label_new (_(global_variable));
13     ...
```

Libraries using `gettext` should use `dgettext()` instead of `gettext()`, which allows them to specify the translation domain each time they ask for a translation. Libraries should also avoid calling `textdomain()`, since they will be specifying the domain instead of using the default.

With the convention that the macro `GETTEXT_PACKAGE` is defined to hold your libraries translation domain, `gi18n-lib.h` can be included to provide the following convenience:

```
1 #define _(x) dgettext (GETTEXT_PACKAGE, x)
```

1.8.

How do I use non-ASCII characters in GTK+ programs ?

GTK+ uses [Unicode](#) (more exactly UTF-8) for all text. UTF-8 encodes each Unicode codepoint as a sequence of one to six bytes and has a number of nice properties which make it a good choice for working with Unicode text in C programs:

- ASCII characters are encoded by their familiar ASCII codepoints.
- ASCII characters never appear as part of any other character.
- The zero byte doesn't occur as part of a character, so that UTF-8 strings can be manipulated with the usual C library functions for handling zero-terminated strings.

More information about Unicode and UTF-8 can be found in the [UTF-8 and Unicode FAQ for Unix/Linux](#). GLib provides functions for converting strings between UTF-8 and other encodings, see `g_locale_to_utf8()` and `g_convert()`.

Text coming from external sources (e.g. files or user input), has to be converted to UTF-8 before being handed over to GTK+. The following example writes the content of a ISO-8859-1 encoded text file to `stdout`:

```
1  gchar *text, *utf8_text;
2  gsize length;
3  GError *error = NULL;
4
5  if (g_file_get_contents (filename, &text, &length, NULL))
6  {
7      utf8_text = g_convert (text, length, "UTF-8", "ISO-8859-1",
8                          NULL, NULL, &error);
9      if (error != NULL)
10     {
11         fprintf ("Couldn't convert file %s to UTF-8\n", filename);
12         g_error_free (error);
13     }
14     else
15         g_print (utf8_text);
16 }
17 else
18     fprintf (stderr, "Unable to read file %s\n", filename);
```

For string literals in the source code, there are several alternatives for handling non-ASCII content:

- | | |
|---------------|---|
| direct UTF-8 | If your editor and compiler are capable of handling UTF-8 encoded sources, it is very convenient to simply use UTF-8 for string literals, since it allows you to edit the strings in "wysiwyg". Note that choosing this option may reduce the portability of your code. |
| escaped UTF-8 | Even if your toolchain can't handle UTF-8 directly, you can still encode string literals in UTF-8 by using octal or hexadecimal escapes like <code>\212</code> or <code>\xa8</code> to encode each byte. This is portable, but modifying the escaped strings is not very convenient. Be careful |

when mixing hexadecimal escapes with ordinary text; "\xa8abcd" is a string of length 1 !

runtime conversion If the string literals can be represented in an encoding which your toolchain can handle (e.g. ISO-8859-1), you can write your source files in that encoding and use `g_convert()` to convert the strings to UTF-8 at runtime. Note that this has some runtime overhead, so you may want to move the conversion out of inner loops.

Here is an example showing the three approaches using the copyright sign © which has Unicode and ISO-8859-1 codepoint 169 and is represented in UTF-8 by the two bytes 194, 169, or "\302\251" as a string literal:

```
1 g_print ("direct UTF-8: ©");
2 g_print ("escaped UTF-8: \302\251");
3 text = g_convert ("runtime conversion: ©", -1, "ISO-8859-1", "UTF-8", NULL, NULL,
4 NULL);
5 g_print(text);
   g_free (text);
```

If you are using `gettext()` to localize your application, you need to call `bind_textdomain_codeset()` to ensure that translated strings are returned in UTF-8 encoding.

1.9.

How do I use GTK+ with C++?

There are two ways to approach this. The GTK+ header files use the subset of C that's also valid C++, so you can simply use the normal GTK+ API in a C++ program. Alternatively, you can use a "C++ binding" such as [gtkmm](#) which provides a native C++ API.

When using GTK+ directly, keep in mind that only functions can be connected to signals, not methods. So you will need to use global functions or "static" class functions for signal connections.

Another common issue when using GTK+ directly is that C++ will not implicitly convert an integer to an enumeration. This comes up when using bitfields; in C you can write the following code:

```
1 gdk_window_set_events (gdk_window,
2                       GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK);
```

while in C++ you must write:

```
1 gdk_window_set_events (gdk_window,
2                       (GdkEventMask) GDK_BUTTON_PRESS_MASK |
3                       GDK_BUTTON_RELEASE_MASK);
```

There are very few functions that require this cast, however.

1.10.

How do I use GTK+ with other non-C languages?

See the [list of language bindings](#) on <https://www.gtk.org>.

1.11.

How do I load an image or animation from a file?

To load an image file straight into a display widget, use [gtk_image_new_from_file\(\)](#)^[1]. To load an image for another purpose, use [gdk_pixbuf_new_from_file\(\)](#). To load an animation, use [gdk_pixbuf_animation_new_from_file\(\)](#). [gdk_pixbuf_animation_new_from_file\(\)](#) can also load non-animated images, so use it in combination with [gdk_pixbuf_animation_is_static_image\(\)](#) to load a file of unknown type.

To load an image or animation file asynchronously (without blocking), use [GdkPixbufLoader](#).

1.12.

How do I draw text ?

To draw a piece of text, use a Pango layout and [pango_cairo_show_layout\(\)](#).

```
1 layout = gtk_widget_create_pango_layout (widget, text);
2 fontdesc = pango_font_description_from_string ("Luxi Mono 12");
3 pango_layout_set_font_description (layout, fontdesc);
4 pango_cairo_show_layout (cr, layout);
5 pango_font_description_free (fontdesc);
6 g_object_unref (layout);
```

See also the [Cairo Rendering](#) section of [Pango manual](#).

1.13.

How do I measure the size of a piece of text ?

To obtain the size of a piece of text, use a Pango layout and [pango_layout_get_pixel_size\(\)](#), using code like the following:

```
1 layout = gtk_widget_create_pango_layout (widget, text);
2 fontdesc = pango_font_description_from_string ("Luxi Mono 12");
3 pango_layout_set_font_description (layout, fontdesc);
4 pango_layout_get_pixel_size (layout, &width, &height);
5 pango_font_description_free (fontdesc);
6 g_object_unref (layout);
```

See also the [Layout Objects](#) section of [Pango manual](#).

1.14.

Why are types not registered if I use their GTK_TYPE_BLAH macro ?

The GTK_TYPE_BLAH macros are defined as calls to `gtk_blah_get_type()`, and the `_get_type()` functions are declared as `G_GNUC_CONST` which allows the compiler to optimize the call away if it appears that the value is not being used.

A common workaround for this problem is to store the result in a volatile variable, which keeps the compiler from optimizing the call away.

```
1 volatile GType dummy = GTK_TYPE_BLAH;
```

1.15.

How do I create a transparent toplevel window ?

To make a window transparent, it needs to use a visual which supports that. This is done by getting the RGBA visual of the screen with `gdk_screen_get_rgba_visual()` and setting it on the window. Note that `gdk_screen_get_rgba_visual()` will return `NULL` if transparent windows are not supported on the screen, you should fall back to `gdk_screen_get_system_visual()` in that case. Additionally, note that this will change from screen to screen, so it needs to be repeated whenever the window is moved to a different screen.

```
1 GdkVisual *visual;
2
3 visual = gdk_screen_get_rgba_visual (screen);
4 if (visual == NULL)
5     visual = gdk_screen_get_system_visual (screen);
6
7 gtk_widget_set_visual (GTK_WIDGET (window), visual);
```

To fill the alpha channel on the window simply use cairos RGBA drawing capabilities.

Note that the presence of an RGBA visual is no guarantee that the window will actually appear transparent on screen. On X11, this requires a compositing manager to be running. See [gtk_widget_is_composited\(\)](#) for a

way to find out if the alpha channel will be respected.

2. Which widget should I use...

2.1.

...for lists and trees?

See [tree widget overview](#) — you should use the [GtkTreeView](#) widget. (A list is just a tree with no branches, so the tree widget is used for lists as well).

2.2.

...for multi-line text display or editing?

See [text widget overview](#) — you should use the [GtkTextView](#) widget.

If you only have a small amount of text, [GtkLabel](#) may also be appropriate of course. It can be made selectable with [gtk_label_set_selectable\(\)](#). For a single-line text entry, see [GtkEntry](#).

2.3.

...to display an image or animation?

[GtkImage](#) can display images in just about any format GTK+ understands. You can also use [GtkDrawingArea](#) if you need to do something more complex, such as draw text or graphics over the top of the image.

2.4.

...for presenting a set of mutually-exclusive choices, where Windows would use a combo box?

With GTK+, a [GtkComboBox](#) is the recommended widget to use for this use case. This widget looks like either a combo box or the current option menu, depending on the current theme. If you need an editable text entry, use the [“has-entry”](#) property.

3. GtkWidget

3.1.

How do I change the color of a widget?

See [gtk_widget_override_color\(\)](#) and [gtk_widget_override_background_color\(\)](#). You can also change the appearance of a widget by installing a custom style provider, see [gtk_style_context_add_provider\(\)](#).

To change the background color for widgets such as [GtkLabel](#) that have no background, place them in a [GtkEventBox](#) and set the background of the event box.

3.2.

How do I change the font of a widget?

This has several possible answers, depending on what exactly you want to achieve. One option is [gtk_widget_override_font\(\)](#).

```
1 PangoFontDesc *font_desc = pango_font_description_new ();
2 pango_font_description_set_size (font_desc, 40);
3 gtk_widget_override_font (widget, font);
4 pango_font_description_free (font_desc);
```

If you want to make the text of a label larger, you can use [gtk_label_set_markup\(\)](#):

```
1 gtk_label_set_markup (label, "<big>big text</big>");
```

This is preferred for many apps because it's a relative size to the user's chosen font size. See [g_markup_escape_text\(\)](#) if you are constructing such strings on the fly.

You can also change the font of a widget by putting

```
.my-widget-class {  
    font: Sans 30;  
}
```

in a CSS file, loading it with [gtk_css_provider_load_from_file\(\)](#), and adding the provider with [gtk_style_context_add_provider_for_screen\(\)](#). To associate this style information with your widget, set a style class on its [GtkStyleContext](#) using [gtk_style_context_add_class\(\)](#). The advantage of this approach is that users can then override the font you have chosen. See the [GtkStyleContext](#) documentation for more discussion.

3.3.

How do I disable/ghost/desensitize a widget?

In GTK+ a disabled widget is termed "insensitive." See [gtk_widget_set_sensitive\(\)](#).

4. GtkTextView

4.1.

How do I get the contents of the entire text widget as a string?

See [gtk_text_buffer_get_bounds\(\)](#) and [gtk_text_buffer_get_text\(\)](#) or [gtk_text_iter_get_text\(\)](#).

```
1 GtkTextIter start, end;  
2 GtkTextBuffer *buffer;  
3 char *text;  
4  
5 buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));  
6 gtk_text_buffer_get_bounds (buffer, &start, &end);  
7 text = gtk_text_iter_get_text (&start, &end);  
8 /* use text */  
9 g_free (text);
```

4.2.

How do I make a text widget display its complete contents in a specific font?

If you use [gtk_text_buffer_insert_with_tags\(\)](#) with appropriate tags to select the font, the inserted text will have the desired appearance, but text typed in by the user before or after the tagged block will appear in the default style.

To ensure that all text has the desired appearance, use [gtk_widget_override_font\(\)](#) to change the default font for the widget.

4.3.

How do I make a text view scroll to the end of the buffer automatically ?

A good way to keep a text buffer scrolled to the end is to place a [mark](#) at the end of the buffer, and give it right gravity. The gravity has the effect that text inserted at the mark gets inserted *before*, keeping the mark at the end.

To ensure that the end of the buffer remains visible, use [gtk_text_view_scroll_to_mark\(\)](#) to scroll to the

mark after inserting new text.

The gtk-demo application contains an example of this technique.

5. [GtkTreeView](#)

5.1.

How do I associate some data with a row in the tree?

Remember that the [GtkTreeModel](#) columns don't necessarily have to be displayed. So you can put non-user-visible data in your model just like any other data, and retrieve it with [gtk_tree_model_get\(\)](#). See the [tree widget overview](#).

5.2.

How do I put an image and some text in the same column?

You can pack more than one [GtkCellRenderer](#) into a single [GtkTreeViewColumn](#) using [gtk_tree_view_column_pack_start\(\)](#) or [gtk_tree_view_column_pack_end\(\)](#). So pack both a [GtkCellRendererPixbuf](#) and a [GtkCellRendererText](#) into the column.

5.3.

I can set data easily on my [GtkTreeStore](#)/[GtkListStore](#) models using [gtk_list_store_set\(\)](#) and [gtk_tree_store_set\(\)](#), but can't read it back?

Both the [GtkTreeStore](#) and the [GtkListStore](#) implement the [GtkTreeModel](#) interface. Consequentially, you can use any function this interface implements. The easiest way to read a set of data back is to use [gtk_tree_model_get\(\)](#).

5.4.

How do I change the way that numbers are formatted by [GtkTreeView](#)?

Use [gtk_tree_view_insert_column_with_data_func\(\)](#) or [gtk_tree_view_column_set_cell_data_func\(\)](#) and do the conversion from number to string yourself (with, say, [g_strdup_printf\(\)](#)).

The following example demonstrates this:

```
1  enum
2  {
3      DOUBLE_COLUMN,
4      N_COLUMNS
5  };
6
7  GtkListStore *mycolumns;
8  GtkTreeView *treeview;
9
10 void
11 my_cell_double_to_text (GtkTreeViewColumn *tree_column,
12                        GtkCellRenderer *cell,
13                        GtkTreeModel *tree_model,
14                        GtkTreeIter *iter,
15                        gpointer data)
16 {
17     GtkCellRendererText *cell_text = (GtkCellRendererText *)cell;
18     gdouble d;
```



```

19  gchar *text;
20
21  /* Get the double value from the model. */
22  gtk_tree_model_get (tree_model, iter, (gint)data, &d, -1);
23  /* Now we can format the value ourselves. */
24  text = g_strdup_printf ("%2f", d);
25  g_object_set (cell, "text", text, NULL);
26  g_free (text);
27  }
28
29  void
30  set_up_new_columns (GtkTreeView *myview)
31  {
32  GtkCellRendererText *renderer;
33  GtkTreeViewColumn *column;
34  GtkListStore *mycolumns;
35
36  /* Create the data model and associate it with the given TreeView */
37  mycolumns = gtk_list_store_new (N_COLUMNS, G_TYPE_DOUBLE);
38  gtk_tree_view_set_model (myview, GTK_TREE_MODEL (mycolumns));
39
40  /* Create a GtkCellRendererText */
41  renderer = gtk_cell_renderer_text_new ();
42
43  /* Create a new column that has a title ("Example column"),
44   * uses the above created renderer that will render the double
45   * value into text from the associated model's rows.
46   */
47  column = gtk_tree_view_column_new ();
48  gtk_tree_view_column_set_title (column, "Example column");
49  renderer = gtk_cell_renderer_text_new ();
50  gtk_tree_view_column_pack_start (column, renderer, TRUE);
51
52  /* Append the new column after the GtkTreeView's previous columns. */
53  gtk_tree_view_append_column (GTK_TREE_VIEW (myview), column);
54  /* Since we created the column by hand, we can set it up for our
55   * needs, e.g. set its minimum and maximum width, etc.
56   */
57  /* Set up a custom function that will be called when the column content
58   * is rendered. We use the func_data pointer as an index into our
59   * model. This is convenient when using multi column lists.
60   */
61  gtk_tree_view_column_set_cell_data_func (column, renderer,
62                                           my_cell_double_to_text,
63                                           (gpointer)DOUBLE_COLUMN, NULL);
64  }

```

5.5.

How do I hide the expander arrows in my tree view ?

Set the expander-column property of the tree view to a hidden column. See [gtk tree view set expander column\(\)](#) and [gtk tree view column set visible\(\)](#).

6. Using cairo with GTK+

6.1.

How do I use cairo to draw in GTK+ applications ?

The [“draw”](#) signal gets a ready-to-use cairo context as parameter that you should use.

All drawing in GTK+ is normally done in a draw handler, and GTK+ creates a temporary pixmap for double-buffering the drawing. It is possible to turn off double-buffering, with [gtk_widget_set_double_buffered\(\)](#), but this is not ideal, since it can cause some flickering.

6.2.

Can I improve the performance of my application by using the Glitz or GL backend of cairo ?

No. The GDK X11 backend uses the cairo X backend (and the other GDK backends use their respective native cairo backends). The GTK+ developers believe that the best way to improving the GDK drawing performance is to optimize the cairo X backend and the relevant code paths in the X server that it uses (mostly the Render extension).

6.3.

Can I use cairo to draw on a [GdkPixbuf](#) ?

No, at least not yet. The cairo image surface does not support the pixel format used by [GdkPixbuf](#).

^[1] If the file load fails, [gtk_image_new_from_file\(\)](#) will display no image graphic — to detect a failed load yourself, use [gdk_pixbuf_new_from_file\(\)](#) directly, then [gtk_image_new_from_pixbuf\(\)](#).

The GTK+ Drawing Model

The GTK+ Drawing Model — The GTK+ drawing model in detail

Overview of the drawing model

This chapter describes the GTK+ drawing model in detail. If you are interested in the procedure which GTK+ follows to draw its widgets and windows, you should read this chapter; this will be useful to know if you decide to implement your own widgets. This chapter will also clarify the reasons behind the ways certain things are done in GTK+; for example, why you cannot change the background color of all widgets with the same method.

Windows and events

Programs that run in a windowing system generally create rectangular regions in the screen called *windows*. Traditional windowing systems do not automatically save the graphical content of windows, and instead ask client programs to repaint those windows whenever it is needed. For example, if a window that is stacked below other windows gets raised to the top, then a client program has to repaint the area that was previously obscured. When the windowing system asks a client program to redraw part of a window, it sends an *exposure event* to the program for that window.

Here, "windows" means "rectangular regions with automatic clipping", instead of "oplevel application windows". Most windowing systems support nested windows, where the contents of child windows get clipped by the boundaries of their parents. Although GTK+ and GDK in particular may run on a windowing system with no such notion of nested windows, GDK presents the illusion of being under such a system. A toplevel window may contain many subwindows and sub-subwindows, for example, one for the menu bar, one for the document area, one for each scrollbar, and one for the status bar. In addition, controls that receive user input, such as clickable buttons, are likely to have their own subwindows as well.

In practice, most windows in modern GTK+ application are client-side constructs. Only few windows (in particular toplevel windows) are *native*, which means that they represent a window from the underlying windowing system on which GTK+ is running. For example, on X11 it corresponds to a `Window`; on Win32, it corresponds to a `HANDLE`.

Generally, the drawing cycle begins when GTK+ receives an exposure event from the underlying windowing system: if the user drags a window over another one, the windowing system will tell the underlying window that it needs to repaint itself. The drawing cycle can also be initiated when a widget itself decides that it needs to update its display. For example, when the user types a character in a `GtkEntry` widget, the entry asks GTK+ to queue a redraw operation for itself.

The windowing system generates events for native windows. The GDK interface to the windowing system translates such native events into `GdkEvent` structures and sends them on to the GTK layer. In turn, the GTK layer finds the widget that corresponds to a particular `GdkWindow` and emits the corresponding event signals on that widget.

The following sections describe how GTK+ decides which widgets need to be repainted in response to such events, and how widgets work internally in terms of the resources they use from the windowing system.

The frame clock

All GTK+ applications are mainloop-driven, which means that most of the time the app is idle inside a loop that just waits for something to happen and then calls out to the right place when it does. On top of this GTK+ has a frame clock that gives a “pulse” to the application. This clock beats at a steady rate, which is tied to the framerate of the output (this is synced to the monitor via the window manager/compositor). The clock has several phases:

- Events
- Update
- Layout
- Paint

The phases happens in this order and we will always run each phase through before going back to the start.

The Events phase is a long stretch of time between each redraw where we get input events from the user and other events (like e.g. network I/O). Some events, like mouse motion are compressed so that we only get a single mouse motion event per clock cycle.

Once the Events phase is over we pause all external events and run the redraw loop. First is the Update phase, where all animations are run to calculate the new state based on the estimated time the next frame will be visible (available via the frame clock). This often involves geometry changes which drives the next phase, Layout. If there are any changes in widget size requirements we calculate a new layout for the widget hierarchy (i.e. we assign sizes and positions). Then we go to the Paint phase where we redraw the regions of the window that need redrawing.

If nothing requires the Update/Layout/Paint phases we will stay in the Events phase forever, as we don't want to redraw if nothing changes. Each phase can request further processing in the following phases (e.g. the Update phase will cause there to be layout work, and layout changes cause repaints).

There are multiple ways to drive the clock, at the lowest level you can request a particular phase with `gdk_frame_clock_request_phase()` which will schedule a clock beat as needed so that it eventually reaches the requested phase. However, in practice most things happen at higher levels:

- If you are doing an animation, you can use `gtk_widget_add_tick_callback()` which will cause a regular beating of the clock with a callback in the Update phase until you stop the tick.
- If some state changes that causes the size of your widget to change you call `gtk_widget_queue_resize()` which will request a Layout phase and mark your widget as needing relayout.
- If some state changes so you need to redraw some area of your widget you use the normal `gtk_widget_queue_draw()` set of functions. These will request a Paint phase and mark the region as needing redraw.

There are also a lot of implicit triggers of these from the CSS layer (which does animations, resizes and repaints as needed).

Hierarchical drawing

During the Paint phase we will send a single expose event to the toplevel window. The event handler will create a cairo context for the window and emit a `GtkWidget::draw()` signal on it, which will propagate down the entire widget hierarchy in back-to-front order, using the clipping and transform of the cairo context. This lets each widget draw its content at the right place and time, correctly handling things like partial transparencies and overlapping widgets.

When generating the event, GDK also sets up double buffering to avoid the flickering that would result from each widget drawing itself in turn. [the section called “Double buffering”](#) describes the double buffering mechanism in detail.

Normally, there is only a single cairo context which is used in the entire repaint, rather than one per `GdkWindow`. This means you have to respect (and not reset) existing clip and transformations set on it.

Most widgets, including those that create their own `GdkWindows` have a transparent background, so they draw on top of whatever widgets are below them. This was not the case in GTK+ 2 where the theme set the background of most widgets to the default background color. (In fact, transparent `GdkWindows` used to be impossible.)

The whole rendering hierarchy is captured in the call stack, rather than having multiple separate draw emissions, so you can use effects like e.g. `cairo_push/pop_group()` which will affect all the widgets below you in the hierarchy. This makes it possible to have e.g. partially transparent containers.

Scrolling

Traditionally, GTK+ has used self-copy operations to implement scrolling with native windows. With transparent backgrounds, this no longer works. Instead, we just mark the entire affected area for repainting when these operations are used. This allows (partially) transparent backgrounds, and it also more closely models modern hardware where self-copy operations are problematic (they break the rendering pipeline).

Since the above causes some overhead, we introduce a caching mechanism. Containers that scroll a lot (`GtkViewport`, `GtkTextView`, `GtkTreeView`, etc) allocate an offscreen image during scrolling and render their children to it (which is possible since drawing is fully hierarchical). The offscreen image is a bit larger than the visible area, so most of the time when scrolling it just needs to draw the offscreen in a different position. This matches contemporary graphics hardware much better, as well as allowing efficient transparent backgrounds. In order for this to work such containers need to detect when child widgets are redrawn so that it can update the offscreen. This can be done with the new `gdk_window_set_invalidate_handler()` function.

Double buffering

If each of the drawing calls made by each subwidget's draw handler were sent directly to the windowing system, flicker could result. This is because areas may get redrawn repeatedly: the background, then decorative frames, then text labels, etc. To avoid flicker, GTK+ employs a *double buffering* system at the GDK level. Widgets normally don't know that they are drawing to an off-screen buffer; they just issue their normal drawing commands, and the buffer gets sent to the windowing system when all drawing operations are done.

Two basic functions in GDK form the core of the double-buffering mechanism:

`gdk_window_begin_paint_region()` and `gdk_window_end_paint()`. The first function tells a `GdkWindow` to create a temporary off-screen buffer for drawing. All subsequent drawing operations to this window get automatically redirected to that buffer. The second function actually paints the buffer onto the on-screen window, and frees the buffer.

Automatic double buffering

It would be inconvenient for all widgets to call `gdk_window_begin_paint_region()` and `gdk_window_end_paint()` at the beginning and end of their draw handlers.

To make this easier, GTK+ normally calls `gdk_window_begin_paint_region()` before emitting the `#GtkWidget::draw` signal, and then it calls `gdk_window_end_paint()` after the signal has been emitted. This is convenient for most widgets, as they do not need to worry about creating their own temporary drawing buffers or about calling those functions.

However, some widgets may prefer to disable this kind of automatic double buffering and do things on their own. To do this, call the `gtk_widget_set_double_buffered()` function in your widget's constructor. Double buffering can only be turned off for widgets that have a native window.

Example 5. Disabling automatic double buffering

```
1  static void
2  my_widget_init (MyWidget *widget)
3  {
4      ...
5
6      gtk_widget_set_double_buffered (widget, FALSE);
7
8      ...
9  }
```

When is it convenient to disable double buffering? Generally, this is the case only if your widget gets drawn in such a way that the different drawing operations do not overlap each other. For example, this may be the case for a simple image viewer: it can just draw the image in a single operation. This would *not* be the case with a word processor, since it will need to draw and over-draw the page's background, then the background for highlighted text, and then the text itself.

Even if you turn off double buffering on a widget, you can still call `gdk_window_begin_paint_region()` and `gdk_window_end_paint()` by hand to use temporary drawing buffers.

App-paintable widgets

Generally, applications use the pre-defined widgets in GTK+ and they do not draw extra things on top of them (the exception being `GtkDrawingArea`). However, applications may sometimes find it convenient to draw directly on certain widgets like toplevel windows or event boxes. When this is the case, GTK+ needs to be told not to overwrite your drawing afterwards, when the window gets to drawing its default contents.

`GtkWindow` and `GtkEventBox` are the two widgets that allow turning off drawing of default contents by calling `gtk_widget_set_app_paintable()`. If you call this function, they will not draw their contents and let you do it instead.

Since the `#GtkWidget::draw` signal runs user-connected handlers *before* the widget's default handler, what usually happens is this:

1. Your own draw handler gets run. It paints something on the window or the event box.
2. The widget's default draw handler gets run. If `gtk_widget_set_app_paintable()` has not been called to turn off widget drawing (this is the default), *your drawing will be overwritten*. An app paintable widget will not draw its default contents however and preserve your drawing instead.
3. The draw handler for the parent class gets run. Since both `GtkWindow` and `GtkEventBox` are descendants of `GtkContainer`, their no-window children will be asked to draw themselves recursively, as described in [the section called “Hierarchical drawing”](#).

Summary of app-paintable widgets. Call `gtk_widget_set_app_paintable()` if you intend to draw your own content directly on a `GtkWindow` and `GtkEventBox`. You seldom need to draw on top of other widgets, and `GtkDrawingArea` ignores this flag, as it *is* intended to be drawn on.

The GTK+ Input and Event Handling Model

The GTK+ Input and Event Handling Model — GTK+ input and event handling in detail

Overview of GTK+ input and event handling

This chapter describes in detail how GTK+ handles input. If you are interested in what happens to translate a key press or mouse motion of the users into a change of a GTK+ widget, you should read this chapter. This knowledge will also be useful if you decide to implement your own widgets.

Devices and events

The most basic input devices that every computer user has interacted with are keyboards and mice; beyond these, GTK+ supports touchpads, touchscreens and more exotic input devices such as graphics tablets. Inside GTK+, every such input device is represented by a [GdkDevice](#) object.

To simplify dealing with the variability between these input devices, GTK+ has a concept of master and slave devices. The concrete physical devices that have many different characteristics (mice may have 2 or 3 or 8 buttons, keyboards have different layouts and may or may not have a separate number block, etc) are represented as slave devices. Each slave device is associated with a virtual master device. Master devices always come in pointer/keyboard pairs - you can think of such a pair as a 'seat'.

GTK+ widgets generally deal with the master devices, and thus can be used with any pointing device or keyboard.

When a user interacts with an input device (e.g. moves a mouse or presses a key on the keyboard), GTK+ receives events from the windowing system. These are typically directed at a specific window - for pointer

events, the window under the pointer (grabs complicate this), for keyboard events, the window with the keyboard focus.

GDK translates these raw windowing system events into `GdkEvents`. Typical input events are:

`GdkEventButton`
`GdkEventMotion`
`GdkEventCrossing`
`GdkEventKey`
`GdkEventFocus`
`GdkEventTouch`

Additionally, GDK/GTK synthesizes other signals to let know whether grabs (system-wide or in-app) are taking input away:

`GdkEventGrabBroken`
[“grab-notify”](#)

When GTK+ is initialized, it sets up an event handler function with `gdk_event_handler_set()`, which receives all of these input events (as well as others, for instance window management related events).

Event propagation

For widgets which have a `GdkWindow` set, events are received from the windowing system and passed to [`gtk_main_do_event\(\)`](#). See its documentation for details of what it does: compression of enter/leave events, identification of the widget receiving the event, pushing the event onto a stack for [`gtk_get_current_event\(\)`](#), and propagating the event to the widget.

When a GDK backend produces an input event, it is tied to a [`GdkDevice`](#) and a `GdkWindow`, which in turn represents a windowing system surface in the backend. If a widget has grabbed the current input device, or all input devices, the event is propagated to that [`GtkWidget`](#). Otherwise, it is propagated to the the [`GtkWidget`](#) which called [`gtk_widget_register_window\(\)`](#) on the `GdkWindow` receiving the event.

Grabs are implemented for each input device, and globally. A grab for a specific input device ([`gtk_device_grab_add\(\)`](#)), is sent events in preference to a global grab ([`gtk_grab_add\(\)`](#)). Input grabs only have effect within the [`GtkWindowGroup`](#) containing the [`GtkWidget`](#) which registered the event's `GdkWindow`. If this [`GtkWidget`](#) is a child of the grab widget, the event is propagated to the child — this is the basis for propagating events within modal dialogs.

An event is propagated to a widget using [`gtk_propagate_event\(\)`](#). Propagation differs between event types: key events (`GDK_KEY_PRESS`, `GDK_KEY_RELEASE`) are delivered to the top-level [`GtkWindow`](#); other events are propagated down and up the widget hierarchy in three phases (see [`GtkPropagationPhase`](#)).

For key events, the top-level window's default “key-press-event” and “key-release-event” signal handlers handle mnemonics and accelerators first. Other key presses are then passed to [`gtk_window_propagate_key_event\(\)`](#) which propagates the event upwards from the window's current focus widget ([`gtk_window_get_focus\(\)`](#)) to the top-level.

For other events, in the first phase (the “capture” phase) the event is delivered to each widget from the top-most (for example, the top-level [`GtkWindow`](#) or grab widget) down to the target [`GtkWidget`](#). [`Gestures`](#) that are attached with `GTK_PHASE_CAPTURE` get a chance to react to the event.

After the “capture” phase, the widget that was intended to be the destination of the event will run gestures attached to it with `GTK_PHASE_TARGET`. This is known as the “target” phase, and only happens on that widget.

Next, the “event” signal is emitted, then the appropriate signal for the event in question, for example “[`motion-notify-event`](#)”. Handling these signals was the primary way to handle input in GTK+ widgets before gestures

were introduced. If the widget is realized, the “[event-after](#)” signal is emitted. The signals are emitted from the target widget up to the top-level, as part of the “bubble” phase.

The default handlers for the event signals send the event to gestures that are attached with [GTK_PHASE_BUBBLE](#). Therefore, gestures in the “bubble” phase are only used if the widget does not have its own event handlers, or takes care to chain up to the default [GtkWidget](#) handlers.

Events are not delivered to a widget which is insensitive or unmapped.

Any time during the propagation phase, a widget may indicate that a received event was consumed and propagation should therefore be stopped. In traditional event handlers, this is hinted by returning [GDK_EVENT_STOP](#). If gestures are used, this may happen when the widget tells the gesture to claim the event touch sequence (or the pointer events) for its own. See the “gesture states” section below to know more of the latter.

Event masks

Each widget instance has a basic event mask and another per input device, which determine the types of input event it receives. Each event mask set on a widget is added to the corresponding (basic or per-device) event mask for the widget’s `GdkWindow`, and all child `GdkWindows`.

If a widget is windowless ([gtk_widget_get_has_window\(\)](#) returns `FALSE`) and an application wants to receive custom events on it, it must be placed inside a [GtkEventBox](#) to receive the events, and an appropriate event mask must be set on the box. When implementing a widget, use a `GDK_INPUT_ONLY` `GdkWindow` to receive the events instead.

Filtering events against event masks happens inside `GdkWindow`, which exposes event masks to the windowing system to reduce the number of events GDK receives from it. On receiving an event, it is filtered against the `GdkWindow`’s mask for the input device, if set. Otherwise, it is filtered against the `GdkWindow`’s basic event mask.

This means that widgets must add to the event mask for each event type they expect to receive, using [gtk_widget_set_events\(\)](#) or [gtk_widget_add_events\(\)](#) to preserve the existing mask. Widgets which are aware of floating devices should use [gtk_widget_set_device_events\(\)](#) or [gtk_widget_add_device_events\(\)](#), and must explicitly enable the device using [gtk_widget_set_device_enabled\(\)](#). See the `GdkDeviceManager` documentation for more information.

All standard widgets set the event mask for all events they expect to receive, and it is not necessary to modify this. Masks should be set when implementing a new widget.

Touch events

Touch events are emitted as events of type [GDK_TOUCH_BEGIN](#), [GDK_TOUCH_UPDATE](#) or [GDK_TOUCH_END](#), those events contain an “event sequence” that univocally identifies the physical touch until it is lifted from the device.

On some windowing platforms, multitouch devices perform pointer emulation, this works by granting a “pointer emulating” hint to one of the currently interacting touch sequences, which will be reported on every `GdkEventTouch` event from that sequence. By default, if a widget didn't request touch events by setting [GDK_TOUCH_MASK](#) on its event mask and didn't override “[touch-event](#)”, GTK+ will transform these “pointer emulating” events into semantically similar `GdkEventButton` and `GdkEventMotion` events. Depending on [GDK_TOUCH_MASK](#) being in the event mask or not, non-pointer-emulating sequences could still trigger gestures or just get filtered out, regardless of the widget not handling those directly.

If the widget sets [GDK_TOUCH_MASK](#) on its event mask and doesn't chain up on [“touch-event”](#), only touch events will be received, and no pointer emulation will be performed.

Grabs

Grabs are a method to claim all input events from a device, they happen either implicitly on pointer and touch devices, or explicitly. Implicit grabs happen on user interaction, when a `GdkEventButtonPress` happens, all events from then on, until after the corresponding `GdkEventButtonRelease`, will be reported to the widget that got the first event. Likewise, on touch events, every [GdkEventSequence](#) will deliver only events to the widget that received its [GDK_TOUCH_BEGIN](#) event.

Explicit grabs happen programatically (both activation and deactivation), and can be either system-wide (GDK grabs) or application-wide (GTK grabs). On the windowing platforms that support it, GDK grabs will prevent any interaction with any other application/window/widget than the grabbing one, whereas GTK grabs will be effective only within the application (across all its windows), still allowing for interaction with other applications.

But one important aspect of grabs is that they may potentially happen at any point somewhere else, even while the pointer/touch device is already grabbed. This makes it necessary for widgets to handle the cancellation of any ongoing interaction. Depending on whether a GTK or GDK grab is causing this, the widget will respectively receive a [“grab-notify”](#) signal, or a `GdkEventGrabBroken` event.

On gestures, these signals are handled automatically, causing the gesture to cancel all tracked pointer/touch events, and signal the end of recognition.

Keyboard input

Event controllers and gestures

Event controllers are standalone objects that can perform specific actions upon received `GdkEvents`. These are tied to a [GtkWidget](#), and can be told of the event propagation phase at which they will manage the events.

Gestures are a set of specific controllers that are prepared to handle pointer and/or touch events, each gestures implementation attempts to recognize specific actions out the received events, notifying of the state/progress accordingly to let the widget react to those. On multi-touch gestures, every interacting touch sequence will be tracked independently.

Being gestures “simple” units, it is not uncommon to tie several together to perform higher level actions, grouped gestures handle the same event sequences simultaneously, and those sequences share a same state across all grouped gestures. Some examples of grouping may be:

A “drag” and a “swipe” gestures may want grouping. The former will report events as the dragging happens, the latter will tell the swipe X/Y velocities only after gesture has finished.

Grouping a “drag” gesture with a “pan” gesture will only effectively allow dragging in the panning orientation, as both gestures share state.

If “press” and “long press” are wanted simultaneously, those would need grouping.

Gesture states

Gestures have a notion of “state” for each individual touch sequence. When events from a touch sequence are first received, the touch sequence will have “none” state, this means the touch sequence is being handled by the gesture to possibly trigger actions, but the event propagation will not be stopped.

When the gesture enters recognition, or at a later point in time, the widget may choose to claim the touch sequences (individually or as a group), hence stopping event propagation after the event is run through every gesture in that widget and propagation phase. Anytime this happens, the touch sequences are cancelled downwards the propagation chain, to let these know that no further events will be sent.

Alternatively, or at a later point in time, the widget may choose to deny the touch sequences, thus letting those go through again in event propagation. When this happens in the capture phase, and if there are no other claiming gestures in the widget, a [GDK_TOUCH_BEGIN/GDK_BUTTON_PRESS](#) event will be emulated and propagated downwards, in order to preserve consistency.

Grouped gestures always share the same state for a given touch sequence, so setting the state on one does transfer the state to the others. They also are mutually exclusive, within a widget there may be only one gesture group claiming a given sequence. If another gesture group claims later that same sequence, the first group will deny the sequence.
